

Generic Conversions of Abstract Syntax Representations

Steven Keuchel

Johan Jeuring

Technical Report UU-CS-2012-009

Department of Information and Computing Sciences

Utrecht University, Utrecht, The Netherlands

www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Generic Conversions of Abstract Syntax Representations

Steven Keuchel

Universiteit Gent
steven.keuchel@ugent.be

Johan Jeuring

Universiteit Utrecht & Open Universiteit Nederland
j.t.jeuring@uu.nl

Abstract

In this paper we present a datatype-generic approach to syntax with variable binding. A universe specifies the binding and scoping structure of object languages, including binders that bind multiple variables as well as sequential and recursive scoping. Two interpretations of the universe are given: one based on parametric higher-order abstract syntax and one on well-typed de Bruijn indices. The former provides convenient interfaces to embedded domain-specific languages, but is awkward to analyse and manipulate directly, while the latter is a convenient representation in implementations, but is unusable as a surface language. We show how to generically convert from the parametric HOAS interpretation to the de Bruijn interpretation thereby taking the pain from DSL developer to write the conversion themselves.

Categories and Subject Descriptors D.2.13 [Reusable Software]: Reusable libraries

General Terms Languages

Keywords datatype-generic programming, fixed points, abstract syntax

1. Introduction

Datatype-generic techniques allow the definition of functions that can be applied to an entire class of datatypes. A generic function only depends on the structure of datatypes for which it is defined.

In order to define a generic function, we must thus provide access to the underlying structure of datatypes.

A good choice of representation is needed to include the relevant structure and to make it easy to define generic functions. An elegant way available in a dependently-typed setting is the use of a universe construction to model the structure of datatypes. A universe is a collection of types given by a set of codes and an interpretation function that maps codes to proper types. The code reflects the structure of its type, and thus a generic function can learn about the structure of a type by inspecting its code.

Variable binding is a domain that exhibits a lot of generic functionality. Datatype-generic treatment of abstract syntax has already been addressed in the literature [4, 7, 8, 12] using different approaches to abstract syntax with different trade-offs in terms of elegance, usability and expressivity.

Well-typed de Bruijn indices and higher-order abstract syntax (HOAS) are two approaches that statically guarantee that functions defined over these representation respect scoping. However, both come with their own trade-offs.

Well-typed de Bruijn indices provide an easy and elegant way to adequately encode the scoping and typing structure of simply-typed languages, but are virtually impossible to use by humans.

The HOAS approach uses the meta-language's bindings to model binding of a language, which gives the user a pleasant interface to write terms directly, but does not allow direct definitions of many desired operations.

Especially in the field of embedded domain-specific languages (EDSLs), we would like to combine the advantages of both. In fact, people do use both approaches simultaneously and convert between them when necessary [2, 3, 5]. However, to the best of our knowledge, a datatype-generic treatment of such a conversion has not been addressed before.

In this paper, we show a particular approach that captures the scoping structure of abstract syntax types with variable binding. Specifically, our contributions are the following:

- We develop a universe for generic programming with syntax types that is very expressive, in the sense that it allows the specification of rich binding structures such as patterns, declarations including sequential and recursive scoping, and in the sense that it captures the scoping and typing structure of simply-typed languages.
- We show how to extend the MultiRec [13] approach to incorporate scoping. This makes it possible to define an interpretation of the universe in terms of well-typed de Bruijn representations.
- We give an interpretation of the universe using the parametric higher-order abstract syntax (PHOAS) variant proposed by [1] and show how to construct a novel generic conversion function from PHOAS to de Bruijn representations.

The rest of this paper is organized as follows: we introduce the well-scoped de Bruijn approach to variable binding in Section 2, by demonstrating how to encode a few representative languages. In Section 3 we develop a first version of our universe for mutually recursive families of datatypes representing syntax with unary binding. Section 4 introduces the PHOAS approach, and Section 5 gives an alternative interpretation of the universe in terms of PHOAS. In Section 6 we revisit the well-scoped de Bruijn approach and show how to express rich binding forms and recursive as well as sequential scoping. An enriched universe, which supports these features, and a new generic conversion function are developed in Section 7. We conclude in Section 8, pointing to related research and giving possible directions for future research.

We perform our development in the dependently-typed programming language Agda [9] and assume the reader is familiar with Agda. Norell [10] provides an introduction, including a universe for generic programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WGP'12, September 9, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1576-0/12/09...\$10.00

2. Well-scoped de Bruijn representations

This section introduces essential elements for well-scoped de Bruijn representations of syntax with binding. We restrict ourselves to binding forms that bind a single variable. Later we will introduce other forms of scoping and binding forms that are found in many programming languages.

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 to restrict variable occurrences to variables from the context given by the index of the family. Throughout this paper we will use the type Ctx defined below as this index set.

The syntax of a language may involve terms of different syntactic sorts, so we define Ctx to be a list of Sort for a given type Sort that represents all syntactic sorts with variables of a specific language.

```

module Context (Sort : Set) where
  infixl 5  $\_ \triangleright \_$ 
  data Ctx : Set where
     $\epsilon$  : Ctx
     $\_ \triangleright \_$  : ( $\Gamma$  : Ctx) (s : Sort)  $\rightarrow$  Ctx

```

The type of a variable is then a membership predicate on a context, i.e. a variable of sort (s : Sort) in a context (Γ : Ctx) is a proof that s is contained in the list Γ .

```

infix 4  $\_ \ni \_$ 
data  $\_ \ni \_$  : ( $\Gamma$  : Ctx)  $\rightarrow$  Sort  $\rightarrow$  Set where
   $vz$  : { $\Gamma$  : Ctx} {s : Sort}  $\rightarrow$   $\Gamma \triangleright s \ni s$ 
   $vs$  : { $\Gamma$  : Ctx} {s t : Sort}  $\rightarrow$   $\Gamma \ni s \rightarrow \Gamma \triangleright t \ni s$ 

```

2.1 Single-sorted syntax

We define a well-scoped de Bruijn representation for the untyped λ -terms. Terms are the only syntactic sort, so we can use a singleton type for Sort . Note that in the case of a λ -abstraction exactly one variable is bound, so the context has to be extended by appending a new element.

```

data Sort : Set where
  term : Sort
open Context Sort
data Lam ( $\Gamma$  : Ctx) : Set where
   $var_{DB}$  :  $\Gamma \ni term \rightarrow$  Lam  $\Gamma$ 
   $app_{DB}$  : Lam  $\Gamma \rightarrow$  Lam  $\Gamma \rightarrow$  Lam  $\Gamma$ 
   $abs_{DB}$  : Lam ( $\Gamma \triangleright term$ )  $\rightarrow$  Lam  $\Gamma$ 

```

2.2 Multi-sorted syntax

Object languages may deal with multiple sorts of variables. System F for example has two syntactic sorts: terms and types. We use a two-valued datatype to represent these sorts.

```

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

data Type ( $\Gamma$  : Ctx) : Set where
  var :  $\Gamma \ni type \rightarrow$  Type  $\Gamma$ 
  arr : Type  $\Gamma \rightarrow$  Type  $\Gamma \rightarrow$  Type  $\Gamma$ 
  for : Type ( $\Gamma \triangleright type$ )  $\rightarrow$  Type  $\Gamma$ 

data Term ( $\Gamma$  : Ctx) : Set where
   $var_{DB}$  :  $\Gamma \ni term \rightarrow$  Term  $\Gamma$ 
   $app_{DB}$  : Term  $\Gamma \rightarrow$  Term  $\Gamma \rightarrow$  Term  $\Gamma$ 
   $abs_{DB}$  : Type  $\Gamma \rightarrow$  Term ( $\Gamma \triangleright term$ )  $\rightarrow$  Term  $\Gamma$ 
   $\tau app_{DB}$  : Term  $\Gamma \rightarrow$  Type  $\Gamma \rightarrow$  Term  $\Gamma$ 
   $\tau abs_{DB}$  : Term ( $\Gamma \triangleright type$ )  $\rightarrow$  Term  $\Gamma$ 

```

In the case of type binders τabs and for we extend the context with the sort value representing types and in the case of λ -abstraction we extend the context with the sort value representing terms. In System F, the constructor for λ -abstraction has a type annotation, so this is a family of two syntax types that are however not mutually recursive. Note that this representation only guarantees that terms are well-scoped, but it does not guarantee that they obey the System F typing rules.

3. Universes of de Bruijn syntax types

A generic function depends on the structure of the datatype on which it is used. To define a generic function we need a uniform view and a uniform representation of the structure of datatypes. In a dependently-typed setting, we obtain a uniform representation by using a universe construction. A universe is a set of codes together with an interpretation that maps codes to types. To define a generic function, the structure it depends on needs to be reflected in the codes.

The generic function then inspects the code to learn about the structure of a type and can apply its generic functionality.

In this section we develop a universe for generic programming with de Bruijn types, i.e. types indexed by a context:

```

DeBruijn : Set1
DeBruijn = Ctx  $\rightarrow$  Set

```

A regular datatype is a datatype representable as a fixed point of a polynomial functor of type $\text{Set} \rightarrow \text{Set}$. In Section 3.1 we establish a generic view of single de Bruijn types related to regular datatypes, i.e. de Bruijn types representable as fixed points of functors of type $\text{DeBruijn} \rightarrow \text{DeBruijn}$. In Section 3.2 we extend our view to families of possibly mutually recursive de Bruijn types and in Section 3.3 define a universe that enables us to program generically.

3.1 Fixed points for regular de Bruijn types

Datatypes whose structure can be represented by polynomial functors – consisting of sums, products and constants – are often called *regular datatypes*. In Haskell or ML-like datatype definitions these are directly recursive datatypes. We now look at corresponding well-typed de Bruijn representations of type $\text{Ctx} \rightarrow \text{Set}$ that are directly recursive, and represent their structure by polynomial functors extended with a notion of binding.

We can represent the datatype for the first representation of the untyped λ -calculus of Section 2.1 by its pattern functor where we abstract from the recursive positions using a new parameter Φ . It is straightforward to verify that LamF indeed is a functor between syntax types by defining a functorial mapping map-lamf .

```

data LamF ( $\Phi$  : DeBruijn) ( $\Gamma$  : Ctx) : Set where
   $var_{DB}$  :  $\Gamma \ni term \rightarrow$  LamF  $\Phi$   $\Gamma$ 
   $app_{DB}$  :  $\Phi$   $\Gamma \rightarrow$   $\Phi$   $\Gamma \rightarrow$  LamF  $\Phi$   $\Gamma$ 
   $abs_{DB}$  :  $\Phi$  ( $\Gamma \triangleright term$ )  $\rightarrow$  LamF  $\Phi$   $\Gamma$ 
   $map\text{-lamf}$  :  $\forall \{ \Phi \Psi \} \rightarrow (\Phi \rightsquigarrow \Psi) \rightarrow$  LamF  $\Phi \rightsquigarrow$  LamF  $\Psi$ 

```

We use $_ \rightsquigarrow _$ to denote morphisms between syntax types, i.e. functions that are natural in the context.

```

 $\_ \rightsquigarrow \_$  : DeBruijn  $\rightarrow$  DeBruijn  $\rightarrow$  Set
 $\Phi \rightsquigarrow \Psi$  = { $\Gamma$  : Ctx}  $\rightarrow$   $\Phi$   $\Gamma \rightarrow$   $\Psi$   $\Gamma$ 

```

Furthermore, we define conversion functions

```

to : LamF Lam  $\rightsquigarrow$  Lam
to ( $var_{DB}$  x) =  $var_{DB}$  x
to ( $app_{DB}$  s t) =  $app_{DB}$  s t
to ( $abs_{DB}$  s) =  $abs_{DB}$  s
from : Lam  $\rightsquigarrow$  LamF Lam
from ( $var_{DB}$  x) =  $var_{DB}$  x
from ( $app_{DB}$  s t) =  $app_{DB}$  s t
from ( $abs_{DB}$  s) =  $abs_{DB}$  s

```

which trivially form inverses of each other. It follows that Lam and LamF Lam are isomorphic and Lam forms a fixed point of LamF . In the rest of this paper we call types like LamF Lam a one-level unfolding of a fixed point.

3.2 Fixed points for inductive families of syntax types

Types representing syntax with binding are often large families of mutually recursive datatypes. In this section we extend the generic view from Section 3.1 to families. Our approach is based on `MultiRec` [13]. `MultiRec` views families of datatypes as fixed points of indexed functors of type $(I \rightarrow \text{Set}) \rightarrow (I \rightarrow \text{Set})$ for some index set I specific to the family. In a similar vein, we view families of de Bruijn types as fixed points of indexed functors of type $(I \rightarrow \text{DeBruijn}) \rightarrow (I \rightarrow \text{DeBruijn})$.

As an example, consider the following alternative representation of the untyped λ -calculus, where application is defined with a non-empty list of arguments.

```
mutual
data Lam (Γ : Ctx) : Set where
  varDB : Γ ∋ term → Lam Γ
  appDB : Lam Γ → Args Γ → Lam Γ
  absDB : Lam (Γ ▷ term) → Lam Γ
data Args (Γ : Ctx) : Set where
  [ ]DB : Lam Γ → Args Γ
  ::DB : Lam Γ → Args Γ → Args Γ
```

We have a family of two datatypes that are mutually recursive and Lam is the only type in the family with variables. Let us model this family more explicitly. The type index forms an index set, i.e. it has one value for each de Bruijn type in the family. LamI maps each index to the appropriate type.

```
data Index : Set where
  term args : Index
LamI : Index → DeBruijn
LamI term = Lam
LamI args = Args
```

LamF below is a *pattern functor* for this family. Each constructor of the functor corresponds to a specific constructor of a type in the family. Recursive occurrences are replaced by the parameter Φ at the appropriate index.

```
data LamF (Φ : Index → DeBruijn) : Index → DeBruijn where
  varDB : ∀ {Γ} → Γ ∋ term → LamF Φ term Γ
  appDB : ∀ {Γ} → Φ term Γ → Φ args Γ → LamF Φ term Γ
  absDB : ∀ {Γ} → Φ term (Γ ▷ term) → LamF Φ term Γ
  [ ]DB : ∀ {Γ} → Φ term Γ → LamF Φ args Γ
  ::DB : ∀ {Γ} → Φ term Γ → Φ args Γ → LamF Φ args Γ
```

As in the regular case it is easy to define conversion functions

```
fromLamI : ∀ {i} → LamI i ∼ LamF LamI i
toLamI : ∀ {i} → LamF LamI i ∼ LamI i
```

and prove them inverses of each other. Consequently, LamI is a fixed point of LamF .

3.3 A universe of indexed functors

In the following, we develop a universe that can be used to construct pattern functors for families such as LamI from Section 3.2 systematically. Our universe is parameterized over two types I and S . S represents the syntactic sorts with variables in a language and I is an index set for a family of types representing that language. In general S is a subset of I . However, we do not enforce that in the implementation.

```
module Universe (I S : Set) where
```

The codes of the universe are given by three different types `ProdCode`, `TagCode` and `DataCode` that closely follow the struc-

ture of Haskell datatype definitions of pattern functors. A `ProdCode` describes one or more fields of a constructor.

```
data ProdCode : Set where
  one : ProdCode
  rec : (i : I) → ProdCode
  ⊗ : (c1 c2 : ProdCode) → ProdCode
  abs : (s : S) (i : I) → ProdCode
```

The `one` code represents a field of a unit type and the code `rec i` represent a field that references the i th member of the parameter family of the functor. The `abs s i` code also references the parameter, and additionally introduces a new abstraction over a variable of sort s . Using `⊗` we can combine different fields of a constructor. Note that we do not include information about variables in the codes. Variables are handled by the interpretation.

A `TagCode` represents a single constructor. It consists of a `ProdCode` for the fields together with an index $i : I$ that designates the family member that is targeted by the constructor.

```
data TagCode : Set where
  ▶ : (c : ProdCode) (i : I) → TagCode
```

Finally the type `DataCode` represents a complete data declaration as a list of `TagCodes`.

```
DataCode : Set
DataCode = List TagCode
```

The well-scoped representation of the alternative untyped λ -calculus from Section 3.2 has the following `DataCode` in our universe:

```
data I : Set where
  term args : I
data S : Set where
  term : S
LamCode : DataCode
LamCode = (rec term ⊗ rec args ▶ term)
          :: (abs term term ▶ term)
          :: (rec term ▶ args)
          :: (rec term ⊗ rec args ▶ args)
          :: [ ]
```

3.4 Interpretation of the universe

We now come to the definition of an interpretation for codes. So far we have been working on the sorts with variables S and on the indices I of the family without having any information which datatype in the family belongs to a specific sort $(s : S)$. We assume that the user provides a function `embed : S → I` that establishes this missing link. We do not require `embed` to be injective. However, in general S is a subset of I and `embed` is the injection.

An interpretation of a code is a functor that takes a parameter $(\Phi : I \rightarrow \text{DeBruijn})$. For convenience we use a parameterized module and inside this module define functions of type $I \rightarrow \text{DeBruijn}$.

For the interpretation of `one` we provide a single constructor in every context, for `rec i` we reference the member i of the parameter family Φ , in the case of a product we pack the interpretation of the components together and finally for `abs s i` we reference the parameter Φ in a context extended by s .

```
module Interpretation (embed : S → I) (Φ : I → DeBruijn) where
data P[ ] : ProdCode → DeBruijn where
  one : ∀ {Γ} → P[ one ] Γ
  rec : ∀ {Γ i} → Φ i Γ → P[ rec i ] Γ
  ⊗ : ∀ {Γ c1 c2} → P[ c1 ] Γ → P[ c2 ] Γ → P[ c1 ⊗ c2 ] Γ
  abs : ∀ {Γ s i} → Φ i (Γ ▷ s) → P[ abs s i ] Γ
```

For a tagged code $c \blacktriangleright o$ 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 T[ ] : TagCode → I → DeBruijn where
  tag : ∀ {o Γ c} → P[ c ] Γ → T[ c ▶ o ] o Γ
```

A `DataCode` is interpreted in two steps. The interpretation function `D[]` covers the sum structure of a `DataCode`. Finally, `[]` adds variables for functors. For each `s : S`, `var` constructs a value of the embed `s` member.

```
data D[ ] : DataCode → I → DeBruijn where
  top : ∀ {o Γ c cs} → T[ c ] o Γ → D[ c :: cs ] o Γ
  pop : ∀ {o Γ c cs} → D[ cs ] o Γ → D[ c :: cs ] o Γ
data [ ] : DataCode → I → DeBruijn where
  var : ∀ {s Γ c} → Γ ∋ s → [ c ] (embed s) Γ
  ◁ : ∀ {o Γ c} → D[ c ] o Γ → [ c ] o Γ
```

We pack the information for a specific language syntax representation together in a record. This includes the family representing the syntax, its code in the universe, the embed function and finally conversions from and to the one level unfolding of the syntax family.

```
↳ : (I → DeBruijn) → (I → DeBruijn) → Set
F ↳ G = {i : I} {Γ : Ctx} → F i Γ → G i Γ
record DeBruijnFamily : Set1 where
  field
    syntaxFamily : I → DeBruijn
    syntaxCode : DataCode
    syntaxEmbed : S → I
  open Interpretation syntaxEmbed syntaxFamily
  field
    from : syntaxFamily ↳ [ syntaxCode ]
    to : [ syntaxCode ] ↳ syntaxFamily
```

3.5 Quantified constructors

Variables in the de Bruijn representations are enriched with information about syntactic sorts. This idea can be taken further to include type information about variables. Types are a refinement of syntactic sorts and can replace sorts in the context representation leading to well-typed representations.

A well-typed representation for the simply-typed λ -calculus (STLC) is given below.

```
infixr 6 _→_
data Ty : Set where
  unit : Ty
  _→_ : (τ1 τ2 : Ty) → Ty
open Context Ty
infix 2 _┌_
data _┌_ (Γ : Ctx) : Ty → Set where
  varDB : ∀ {τ} → Γ ∋ τ → Γ ┌ τ
  appDB : ∀ {τ1 τ2} → Γ ┌ τ1 → τ2 → Γ ┌ τ1 → Γ ┌ τ2
  absDB : ∀ {τ1 τ2} → Γ ▷ τ1 ┌ τ2 → Γ ┌ τ1 → τ2
  ttDB : Γ ┌ unit
```

The family of STLC terms is indexed over simple types with one base type unit. Note that the application and abstraction constructors are universally quantified over two simple-types and the variable constructor is quantified over a single simple-type. In this section we extend the universe to include this kind of quantification.

In the STLC example the quantification is over simple types, i.e. the whole index of the family. In general quantifications over a subset of the index set are useful. Consider for example adding declarations to the STLC example. In this case, we quantify over the subset that corresponds to the types of expressions.

For the sake of presentation, however, we restrict ourselves to the case of the full index set. It is possible to alter the universe to quantify over other sets. We redefine `TagCode` to include an

other alternative σ , which takes a function `cf` mapping `I` to a new `TagCode`.

```
data TagCode : Set where
  _┌_ : (c : ProdCode) (i : I) → TagCode
  σ : (cf : I → TagCode) → TagCode
```

We also redefine the interpretation of `TagCodes` and add a new alternative `some` that is universally quantified over a value `i : I`. It uses the interpretation of the new `TagCode` obtained by applying the `cf` function.

```
data T[ ] : TagCode → I → DeBruijn where
  tag : ∀ {o Γ c} → P[ c ] Γ → T[ c ▶ o ] o Γ
  some : ∀ {o Γ cf} {i : I} → T[ cf i ] o Γ → T[ σ cf ] o Γ
```

This extended universe is rich enough to include the STLC example from above. Both the index set `I` of the family and the set `S` of sorts with variables are `Ty`. The `embed` function is the identity in this case.

```
Stlc : Ty → DeBruijn
Stlc α Γ = Γ ┌ α
open Interpretation id Stlc
```

The codes for the `Stlc` family are shown below.

```
StlcApp = σ (λ α → σ (λ β → rec (α → β) ⊗ rec α ▶ β))
StlcAbs = σ (λ α → σ (λ β → abs α β ▶ α → β))
StlcUnit = one ▶ unit
StlcCode : DataCode
StlcCode = StlcApp :: StlcAbs :: StlcUnit :: []
```

We complete this example by showing how to fully instantiate the record type from Section 3.4 for the `Stlc` family including the conversion functions from and to the one-level unfolding.

```
StlcSyntaxFamily : SyntaxFamily
StlcSyntaxFamily = record {
  syntaxCode = StlcCode;
  syntaxFamily = Stlc;
  from = from;
  to = to }
where
  from : Stlc ↳ [ StlcCode ]
  from (varDB x) = var x
  from (appDB s t) = < top (some (some (tag (rec s ⊗ rec t)))) >
  from (absDB s) = < pop (top (some (some (tag (abs s)))) >
  from ttDB = < pop (pop (top (tag one))) >
  to : [ StlcCode ] ↳ Stlc
  to (var x) = varDB x
  to < top (some (some (tag (rec s ⊗ rec t)))) > = appDB s t
  to < pop (top (some (some (tag (abs s)))) > = absDB s
  to < pop (pop (top (tag one))) > = ttDB
  to < pop (pop (pop ())) >
```

4. Parametric higher-order abstract syntax

This section introduces another approach to represent syntax with binding called higher-order abstract syntax (HOAS). It uses the function space of the meta-language to model binding in an object-language.

We will begin by giving a general outline of the ideas behind HOAS and then introduce a variant called parametric HOAS which we will use. In the last part of this section we show how to write a conversion of a parametric HOAS representation of the untyped λ -calculus to a de Bruijn representation. This serves as a template for a generic conversion that we will develop later.

The HOAS approach uses the function space of the meta-language to model variable binding. So the λ -expression $\lambda x \rightarrow e$ binds the variable `x` in `e`. We can use this in the definition of a datatype representation by using function types where variable


```

size : Cλ → ℕ
size t = t {sizeAlg}

```

4.2 Conversion to de Bruijn terms

In this section we develop a conversion function from the PHOAS representation of the untyped λ -calculus of Section 4.1 to the well-scoped de Bruijn representation given in Section 2.1. The conversion will be written as a fold by constructing a suitable LamAlgebra. Instead of using the Lam 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 constructs a term which is closed in Γ . Abstracting from the context allows us to instantiate a term in the correct context.

```

Exp : Set
Exp = (Γ : Ctx) → Lam Γ
conversionAlgebra : LamAlgebra Exp
conversionAlgebra = record
  { appPH = λ a b Γ → appDB (a Γ) (b Γ)
  ; absPH = λ f Γ → absDB (f (makeExpVar Γ) (Γ ▷ term)) }
  where makeExpVar : (Γ : Ctx) → Exp
        makeExpVar Γ Δ = ...
convert : Cλ → Exp
convert t = t {conversionAlgebra}

```

To construct a de Bruijn term of an application we instantiate both subterms in the given context and combine the results. In case of a λ -abstraction the body is represented by a meta-language function f . The function `makeExpVar` will create an Exp value representing the new variable which is passed to f . The body of the λ -abstraction will then be constructed in the bigger context $\Gamma \triangleright \text{term}$.

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 a de Bruijn 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. [2] point out, Agda's type system unfortunately does not provide us with enough information to obtain such a proof. Atkey [1] provides a meta-theoretical proof of this property for an encoding in System F, however, his proof relies on the parametricity of the universal quantification in C_λ . We define an almost well-typed `makeExpVar` function at the end of the following subsection. For a completely well-typed solution, we could alter the PHOAS types to include information about the current term context or we could incorporate parametricity principles into the type system.

Using the conversion algebra with the K and S combinator examples produces the following desired results. We make use of a function $v : \{\Gamma : \text{Ctx}\} (n : \mathbb{N}) \{- : \dots\} \rightarrow \text{Lam } \Gamma$ as a convenient way to write de Bruijn indices using natural numbers, i.e. $v \ 2$ stands for `var (vs (vs vz))`.

```

convert K ε ≡ absDB (absDB (v 1))
convert S ε ≡ absDB (absDB (absDB
  (appDB (appDB (v 2) (v 0))
  (appDB (v 1) (v 0))))))

```

4.3 Prefix relation

We develop a predicate `_⊆_` for the sub-context relation and implement a deciding function for it. Using it we can reconstruct the necessary information for `makeExpVar`, i.e. decide the relation $\Gamma \triangleright \text{term} \sqsubseteq \Delta$. 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.

We assume that we operate on a language with syntactic sorts given by $S : \text{Set}$ and that we have a function `_≐?S_` : $\forall (x y : \text{Sort}) \rightarrow \text{Dec } (x \equiv y)$ which decides equality on Sort. The result of `_≐?S_` 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

```

For the conversion we will need to relate a context Γ outside of an abstraction to contexts inside the abstraction. Consider the case of a single variable abstraction where a new variable for the sort s is introduced in an outer context Γ . As we only append new variables to a context, clearly the context $\Gamma \triangleright s$ will be a prefix of any context Δ inside the abstraction. The following predicate `_⊆_` encodes the prefix relation between contexts. Furthermore this predicate is decidable and `_≐?_` 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) (s : Sort) → Γ ⊆ Δ ▷ s
infix 4 _≐?_
_≐?_ : ∀ (x y : Sort) → Dec (x ≡ y)

```

Given a proof for $\Gamma \triangleright s \sqsubseteq \Delta$ we can create an index in Δ for the variable introduced in 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 Section 4.2. As mentioned before we have to postulate the result in the negative case.

```

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

```

5. PHOAS interpretation of syntax families

We develop 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. We assume that we are given a complete representation of a syntax family:

```

module PHOAS {I S : Set} (SF : SyntaxFamily I S) where
  open SyntaxFamily SF
  open Interpretation syntaxEmbed syntaxFamily

```

5.1 Church encodings

The Church encodings of syntax families are the types of a fold operator that expects an algebra for some abstract carrier $A : I \rightarrow \text{Set}$. We use the module Church that is parameterized over such a carrier and inside define functions that interpret codes as algebra types with carrier A .

As before, an algebra will be a product of algebra functions, one per non-variable constructor. Generally an algebra function has the type $\forall \{i\} \rightarrow F A i \rightarrow A i$ for some functor F . The function `Arg` computes this functor for a `ProdCode`. In the interesting case of an abstraction, we produce a function type that provides us with one meta-language variable.


```

module Church (A : I → Set) where
  Arg : ProdCode → I → Set
  Arg one   j = T
  Arg (rec i) j = A i
  Arg (c1 ⊗ c2) j = Arg c1 j × Arg c2 j
  Arg (abs s i) j = A (syntaxEmbed s) → A i

```

Note that the index parameter $j : I$ is never used, except in recursive calls of `Arg`, and can thus be dropped. The fields of a constructor corresponding to a `ProdCode` do not depend or change the output index. To get more flexibility we make the result type into a parameter $R : Set$ and calculate $F A \rightarrow R$ in a curried style, to eliminate the unit type and the products. This is implemented by `AlgProd`.

```

AlgProd : ProdCode → Set → Set
AlgProd one   R = R
AlgProd (rec i) R = A i → R
AlgProd (c1 ⊗ c2) R = AlgProd c1 (AlgProd c2 R)
AlgProd (abs s i) R = (A (syntaxEmbed s) → A i) → R

```

`AlgTag` computes the algebra type for a constructor described by a `TagCode`. In case of a σ cf we add a universal quantification and recurse over the produced `TagCode`. For a tag, we select the appropriate member for the result family and call `AlgProd`. We again parameterize over the result family instead of fixing it to A .

```

AlgTag : (c : TagCode) (R : I → Set) → Set
AlgTag (c ▶ i) R = AlgProd c (R i)
AlgTag ( $\sigma$  cf) R =  $\forall \{i\} \rightarrow$  AlgTag (cf i) R

```

For a `DataCode` the `AlgData` function calculates the product of the algebra function types.

```

AlgData : (c : DataCode) (R : I → Set) → Set
AlgData [] R = T
AlgData (c :: cs) R = AlgTag c R × AlgData cs R

```

The `AlgProd`, `AlgTag` and `AlgData` functions are functorial as witnessed by the following mappings. These will be used in the definition of 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 i) f x = f ○ x
map-prod (c1 ⊗ c2) f x = map-prod c1 (map-prod c2 f) x
map-prod (abs s c) f x = f ○ x

map-tag : (c : TagCode) {R S : I → Set} →
  ( $\forall \{i\} \rightarrow$  R i → S i) → AlgTag c R → AlgTag c S
map-tag (c ▶ i) f x = map-prod c f x
map-tag ( $\sigma$  cf) f x =  $\lambda \{i\} \rightarrow$  map-tag (cf i) f x

map-data : (c : DataCode) {R S : I → Set} →
  ( $\forall \{i\} \rightarrow$  R i → S i) → AlgData c R → AlgData c S
map-data [] f tt = tt
map-data (c :: cs) f (x, y) = map-tag c f x, map-data cs f y

```

The `AlgebraF` and `Algebra` functions are variants of the `AlgTag` and `AlgData` functions, in which the result type is fixed to A .

```

AlgebraF : TagCode → Set
AlgebraF c = AlgTag c A

Algebra : Set
Algebra = AlgData syntaxCode A

```

5.2 A generic conversion algebra

We now come to the definition of a generic conversion algebra. As in the non-generic example, we specify a family $\text{Exp} : I \rightarrow \text{Set}$ 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`, `ExpT` and `ExpD` where the de Bruijn syntax family is wrapped in interpretations of corresponding

codes. The intention is that these will represent partial constructions of the one-level unfolding of the syntax family.

```

Exp : I → Set
Exp i = ( $\Gamma : \text{Ctx}$ ) → syntaxFamily i  $\Gamma$ 
ExpP : ProdCode → Set
ExpP c = ( $\Gamma : \text{Ctx}$ ) → P[[c]]  $\Gamma$ 
ExpT : TagCode → I → Set
ExpT c i = ( $\Gamma : \text{Ctx}$ ) → T[[c]] i  $\Gamma$ 
ExpD : DataCode → I → Set
ExpD c i = ( $\Gamma : \text{Ctx}$ ) → D[[c]] i  $\Gamma$ 
open Church Exp

```

As in Section 4.3 we define a helper function `makeExpVar` that creates an `Exp` value, which calculates a de Bruijn index for a variable that is introduced in a context Γ . Note the use of `to` to perform the one-level folding of the fixed point

```

makeExpVar : ( $\Gamma : \text{Ctx}$ ) (s : S) → Exp (syntaxEmbed s)
makeExpVar  $\Gamma$  s  $\Delta$  with  $\Gamma \triangleright s \sqsubseteq \Delta$ 
makeExpVar  $\Gamma$  s  $\Delta$  | yes  $\Gamma \triangleright s \sqsubseteq \Delta$  = to (var (makeVar  $\Gamma \triangleright s \sqsubseteq \Delta$ ))
makeExpVar  $\Gamma$  s  $\Delta$  | no  $\neg p$  = whatever
where postulate whatever :  $\_$ 

```

The `conv-prod` function produces the part of the conversion algebra corresponding to a `ProdCode` c . The result is a function of type `AlgProd c (ExpP c)` that takes as arguments the fields described by the `ProdCode` and builds the part of the one level unfolding of the syntax family that corresponds to c abstracted over a context Γ .

```

conv-prod : (c : ProdCode) → AlgProd c (ExpP c)
conv-prod one =  $\lambda \Gamma \rightarrow$  one
conv-prod (rec i) =  $\lambda x \Gamma \rightarrow$  rec (x  $\Gamma$ )
conv-prod (abs s i) =  $\lambda f \Gamma \rightarrow$  abs (f (makeExpVar  $\Gamma$  s) ( $\Gamma \triangleright s$ ))
conv-prod (c1 ⊗ c2) =
  map-prod c1 ( $\lambda x \Gamma \rightarrow$  map-prod c2 ( $\lambda y \Gamma \rightarrow$  x  $\Gamma$  ⊗ y  $\Gamma$ ) a2) a1
where a1 = conv-prod c1; a2 = conv-prod c2

```

For a recursive position `rec i` we take the field $(x : \text{Exp } i)$ and wrap the instantiation in the de Bruijn interpretation of `rec i`. In case of an `abs s i`, we get a meta-binding $(f : \text{Exp } (\text{syntaxEmbed } s) \rightarrow \text{Exp } i)$ as argument. A variable is created using `makeExpVar` and the body is instantiated in the extended context $\Gamma \triangleright s$. For a product $c_1 \otimes c_2$ the algebras of the recursive calls have to be combined. For this the functorial mapping of `AlgProd` is used to access the results $(x : \text{ExpP } c_1)$ and $(y : \text{ExpP } c_2)$.

The functions `conv-tag` and `conv-data` calculate the conversion algebra for `TagCodes` and `DataCodes`. They use the functorial mappings and function composition `_o_` to add the appropriate constructors of the de Bruijn interpretation.

```

conv-tag : (c : TagCode) → AlgTag c (ExpT c)
conv-tag (c ▶ i) = map-prod c ( $\_o\_ \text{tag}$ ) (conv-prod c)
conv-tag ( $\sigma$  cf) =  $\lambda \{s\} \rightarrow$  map-tag (cf s) ( $\_o\_ \text{some}$ )
  (conv-tag (cf s))

conv-data : (c : DataCode) → AlgData c (ExpD c)
conv-data [] = tt
conv-data (c :: cs) = map-tag c ( $\_o\_ \text{top}$ ) (conv-tag c),
  map-data cs ( $\_o\_ \text{pop}$ ) (conv-data cs)

```

Finally, `conversionAlgebra` takes care of a one-level folding of the fixed point once the pattern functor is fully constructed.

```

conversionAlgebra : Algebra
conversionAlgebra = map-data syntaxCode
  ( $\lambda x \Gamma \rightarrow$  to  $\langle x \Gamma \rangle$ )
  (conv-data syntaxCode)

```

5.3 Example: Simply-typed lambda calculus

We exemplify the usage of the generic conversion algebra for the simply-typed lambda calculus from section 3.5. The HOAS

interface is determined by the record `StlcAlgebra`. The appropriate types of the fields can easily be calculated using the `AlgebraF` function from the `Church` module. The resulting types are given in comments above the field declarations.

```

open PHOAS StlcSyntaxFamily
record StlcAlgebra (Lam : Ty → Set) : Set where
  open Church Lam
  field
    -- ∀ α β → Lam (α → β) → Lam α → Lam β
    appPH : AlgebraF StlcApp
    -- ∀ α β → (Lam α → Lam β) → Lam (α → β)
    absPH : AlgebraF StlcAbs
    -- Lam unit
    ttPH : AlgebraF StlcUnit
  syntax absPH (λ x → e) = λ x ⇒ e

```

`StlcPHOAS` is the type of the parametric HOAS representation, i.e. the type of fold operators over the syntax. For a convenient way to write HOAS terms, we unpack the record fields using instance arguments.

```

open StlcAlgebra {...}
StlcPHOAS : Ty → Set1
StlcPHOAS τ = ∑ {A} {alg : StlcAlgebra A} → A τ

```

The generic conversion algebra needs to be converted to our language specific `StlcAlgebra` record so that we are able to write the conversion `convert` to the well-typed de Bruijn representation.

```

convertAlg : ∑ {A} → Church.Algebra A → StlcAlgebra A
convertAlg (a, b, c, t) = record { appPH = a; absPH = b; ttPH = c }
convert : ∑ {τ} → StlcPHOAS τ → Exp τ
convert t = t { {convertAlg conversionAlgebra} }

```

Function `convert` converts the apply function

```

apply : StlcPHOAS ((unit → unit) ⇒ unit → unit)
apply = λ f ⇒ λ a ⇒ appPH f a

```

to the following de Bruijn term

```
absDB (absDB (appDB (v 1) (v 0)))
```

6. Binders with embedded terms, sequential and recursive scoping

So far, we have only considered languages with single variable binding. However, modern functional programming languages let users define algebraic datatypes and provide mechanisms to pattern match on them, which allows binding of multiple variables at once. Other rich binding forms allowing multiple bindings are let-expressions, which come in different flavours. Using patterns, non-recursive lets, sequential lets and recursive lets we exemplify how these and other rich binding forms can be encoded using a well-typed de Bruijn representation. In the next section we develop a universe that is rich enough to capture the structure of these binding forms and construct a generic conversion algebra from PHOAS to de Bruijn terms.

6.1 Patterns

In a language with patterns, a pattern in an abstract syntax tree will be described by a value that specifies which variables are to be bound. An abstraction then takes a pattern and a term in an appropriately extended context. In a well-scoped de Bruijn representation this means that the term context is extended by a list of new variables, i.e. another context is appended. This is implemented by the following function:

```

_▷▷_ : Ctx → Ctx → Ctx
Γ ▷▷ ε = Γ
Γ ▷▷ (Δ ▷ τ) = Γ ▷▷ Δ ▷ τ

```

We will call the list of new variables the *binder context* and the context by which terms are indexed the *term context*. A pattern can be represented as a family that is indexed by a binder context. Consider the datatype `Pat` for patterns below. The constructor for pattern variables takes no argument and produces a pattern with a singleton binder context. For a product pattern the binder context is the concatenation of the binder contexts of the sub-patterns. `Lam` is a representation of an untyped λ -calculus with a product constructor `_` and pattern matching in λ -abstractions. In the λ -abstraction case the binder context Δ is appended to the outer term context Γ to form the binder context of the body.

```

data Pat : Ctx → Set where
  varDB : Pat (ε ▷ term)
  _ , _ : ∑ {Δ1 Δ2} → Pat Δ1 → Pat Δ2 → Pat (Δ1 ▷▷ Δ2)
data Lam (Γ : Ctx) : Set where
  varDB : Γ ∋ term → Lam Γ
  appDB : Lam Γ → Lam Γ → Lam Γ
  absDB : ∑ {Δ} → Pat Δ → Lam (Γ ▷▷ Δ) → Lam Γ
  _ , _ : Lam Γ → Lam Γ → Lam Γ

```

6.2 Declarations

Many programming languages offer declarations or definitions that associate names with expressions, type signatures or other kinds of annotations. 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, \dots, x_n is the expression e_{n+1} . They are considered to be bound simultaneously. We want to group related things together and not separate a variable x_i from its expression e_i . A declaration and a list of declarations must be seen as both a value containing expressions that reference variables and thus needs a term context, and as a value that binds variables and thus needs a binder context. The following datatypes implement this for a λ -calculus with let-expressions:

```

mutual
data Decl : Ctx → Ctx → Set where
  declDB : ∑ {Γ} → Lam Γ → Decl (ε ▷ term) Γ
data Decls : Ctx → Ctx → Set where
  dnilDB : ∑ {Γ} → Decls ε Γ
  dconsDB : ∑ {Γ Δ1 Δ2} → Decl Δ1 Γ → Decls Δ2 Γ → Decls (Δ1 ▷▷ Δ2) Γ
data Lam (Γ : Ctx) : Set where
  varDB : Γ ∋ term → Lam Γ
  appDB : Lam Γ → Lam Γ → Lam Γ
  absDB : Lam (Γ ▷ term) → Lam Γ
  letDB : ∑ {Δ} → Decls Δ Γ → Lam (Γ ▷▷ Δ) → Lam Γ

```

Note that the binder context in the `dconsDB` 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 distributed to all embedded expressions and not changed by the declarations.

6.3 Recursive scoping

In a mutually recursive let expression the scope of the variables x_1, \dots, x_n are all the right hand sides e_1, \dots, 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 $\Gamma ▷▷ \Delta$ of the declaration list.

```
letrec : ∑ {Δ} → Decls Δ (Γ ▷▷ Δ) → Lam (Γ ▷▷ Δ) → Lam Γ
```

6.4 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 $i < j \leq n+1$. 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 : ∀ {Γ Δ1 Δ2} → Decl Δ1 Γ →
          Decls Δ2 (Γ ▷▷ Δ1) → Decls (Δ1 ▷▷ Δ2) Γ
data Lam (Γ : Ctx) : Set where
  var  : Γ ∋ term → Lam Γ
  app  : Lam Γ → Lam Γ → Lam Γ
  abs  : Lam (Γ ▷ term) → Lam Γ
  letseq : ∀ {Δ} → Decls Δ Γ → Lam (Δ ▷▷ Γ) → Lam Γ

```

7. A universe of binders

In this section we will generalize the universe of families from Section 3 to include binders. For simplicity we use a uniform representation and index all types in a family by both a term context and a binder context. 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. Furthermore we extend the universe by new structure descriptions for richer forms of binding like recursive and sequential scoping.

7.1 Universe codes

As before, the universe is parameterized over two types I and S for the index and sorts. The `ProdCodes` of the previous universe are extended with three new alternatives: `bsng`, `brec` and `bseq`.

```

module Universe (I S : Set) where
data ProdCode : Set where
  one  : ProdCode
  rec  : I → ProdCode
  _⊗_  : (c1 c2 : ProdCode) → ProdCode
  abs  : (c : ProdCode) → I → ProdCode
  bsng : S → ProdCode
  bseq : (c : ProdCode) → I → ProdCode
  brec : I → ProdCode

```

The code `bsng s` represents a value that binds exactly one new variable of the given sort s . The `bseq` code describes sequential scoping. The variables of values of the first code scope over the second. Recursive scoping is introduced by `brec`. A variable bound by a value of the given code scopes over the value itself. The definitions of `TagCode` and `DataCode` are the same as in Section 3.5.

7.2 De Bruijn interpretation

The interpretations of the different codes are functors over families with two contexts. The interpretation of `one` does not bind any variables, so the binder context is empty. `bsng 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 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. 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. For `bseq` the variables of the first code scope over the second. The binder context of the result is the concatenation of the binders of the sub-components. 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 $\Gamma ▷▷ \Delta$.

```

module Interpretation (embed : S → I)
  (Φ : I → Ctx → Ctx → Set) where
data P[ ] : ProdCode → I → Ctx → Ctx → Set where
  one  : ∀ {o Γ} → P[one] o ε 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 c i o Γ} →
        P[c] o Δ1 Γ → Φ i Δ2 (Γ ▷▷ Δ1) →
        P[abs c i] o Δ2 Γ
  bsng : ∀ {s o Γ} → P[bsng s] o (ε ▷ s) Γ
  bseq : ∀ {Δ1 Δ2 c i o Γ} →
        P[c] o Δ1 Γ → Φ i Δ2 (Γ ▷▷ Δ1) →
        P[bseq c i] o (Δ1 ▷▷ Δ2) Γ
  brec : ∀ {Δ i o Γ} → Φ i Δ (Γ ▷▷ Δ) → P[brec i] o Δ Γ

```

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

```

data T[ ] : TagCode → I → Ctx → Ctx → Set where
  tag  : ∀ {o Γ Δ c} →
        P[c] o Δ Γ → T[c ▶ o] o Δ Γ
  some : ∀ {o Γ Δ cf i} →
        T[cf i] o Δ Γ → T[σ cf] o Δ Γ
data D[ ] : DataCode → I → Ctx → Ctx → Set where
  top  : ∀ {o Γ Δ c cs} → T[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) : I → Ctx → Ctx → Set where
  var  : ∀ {o Γ} → Γ ∋ o → [c] (embed o) ε Γ
  <_> : ∀ {o Γ Δ} → D[c] o Δ Γ → [c] o Δ Γ

```

The `SyntaxFamily` record is upgraded to handle a family of binders.

```

_↗_ : (Φ Ψ : I → Ctx → Ctx → Set) → Set
Φ ↗ Ψ = {i : I} {Δ Γ : Ctx} → Φ i Δ Γ → Ψ i Δ Γ
record SyntaxFamily : Set1 where
field
  syntaxFamily : I → Ctx → Ctx → Set
  syntaxCode   : DataCode
  syntaxEmbed  : S → I
open Interpretation syntaxEmbed syntaxFamily
field
  from : syntaxFamily ↗ [syntaxCode]
  to   : [syntaxCode] ↗ syntaxFamily

```

7.3 Church encodings for binders

For the PHOAS representation we need to define Church encodings for the codes in the universe. A church encoding is the type of a fold operator that accepts an abstract algebra. Algebra carriers for binders are indexed by I and the binder context, i.e. algebra carriers have the type $I \rightarrow Ctx \rightarrow Set$. This allows us to access and modify the binder context. The term context is still handled at the meta-level and hidden in the representation. A function of the form

$$\text{Env } D \Delta \rightarrow R$$

represents a higher-order binding that binds multiple variables Δ simultaneously. `Env` represents a heterogeneous list. Given a domain $D : S \rightarrow Set$ an environments holds a value of type $D s$ for each s in Δ .

```

data Env (D : S → Set) : Ctx → Set where
  ε : Env D ε
  ▷_ : ∀ {Γ τ} → Env D Γ → D τ → Env D (Γ ▷ τ)

```

We fix the domain to be the subfamily of A corresponding to S where additionally the binder context is empty, as variables do not bind other variables. Of course it is more convenient for a user to write these functions in a curried style.

```

module Church (A : I → Ctx → Set) where
  A' : S → Set
  A' s = A (syntaxEmbed s) ε
  Curried : Ctx → Set → Set
  Curried ε r = r
  Curried (Γ ▷ s) r = Curried Γ (A' s → r)
  uncurry : ∀ {Γ r} → Curried Γ r → Env A' Γ → r
  uncurry f ε = f
  uncurry f (xs ▷ x) = uncurry f xs x

```

We now come to the interpretation of universe codes as algebra types and start with the curried variant AlgProd of the algebra for a ProdCode . These are now functions where all arguments and the result type are indexed by a binder context. For one the algebra type is the result type with an empty context and for $\text{bsng } s$ it is the result type with a single variable context. For a recursive position the algebra function needs to map the syntax type with the given index to the result type. Given an abstraction $\text{abs } c \ i$ we introduce a higher-order binding with variables from the binder context Δ_1 of the interpretation of c . The binder context of the result is the same as the one of the value inside the abstraction. 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. For recursive scoping we use a higher-order binding with the binder context Δ from inside the binding. In general Agda's typechecker will not be able to infer the binder context automatically. The user needs to provide it explicitly.

```

AlgProd : ProdCode → (R : Ctx → Set) → Set
AlgProd one R = R ε
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 c i) R = AlgProd c (λ Δ1 →
  {Δ2 : Ctx} → Curried Δ1 (A i Δ2) → R Δ2)
AlgProd (bsng i) R = R (ε ▷ i)
AlgProd (bseq c i) R = AlgProd c (λ Γ1 →
  {Γ2 : Ctx} → Curried Γ1 (A i Γ2) → R (Γ1 ▷▷ Γ2))
AlgProd (brec i) R =
  (Δ : Ctx) → Curried Δ (A i Δ) → R Δ

```

The definition AlgTag 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 functorial mapping. The implementation of these is omitted.

7.4 A generic conversion algebra

In this section we generalize the conversion algebra for the unary binding case from Section 5.2 to types with binders. We focus on the conversion algebra for ProdCodes . The code for TagCodes and DataCodes is identical to the one from Section 5.2.

The carrier of the conversion algebra is a family indexed by the set I and a binder context. For the conversion algebra we use the syntaxFamily where we abstracts from the term context Γ a term is instantiated in.

```

Exp : I → Ctx → Set
Exp i Δ = (Γ : Ctx) → syntaxFamily i Δ Γ
ExpP : ProdCode → I → Ctx → Set

```

```

ExpP c i Δ = (Γ : Ctx) → P [c] i Δ Γ
open Church Exp

```

The conv-curried function converts a higher-order binding. A list of new variables is created using makeExpVars .

```

makeExpVars : ∀ Γ Δ → Env (λ s → Exp (syntaxEmbed s) ε) Δ
makeExpVars Γ ε = ε
makeExpVars Γ (Δ ▷ τ) =
  makeExpVars Γ Δ ▷ makeExpVar (Γ ▷▷ Δ) τ
conv-curried : {i : I} {Δ2 : Ctx} (Δ1 Γ : Ctx) →
  Curried Δ1 (Exp i Δ2) → syntaxFamily i Δ2 (Γ ▷▷ Δ1)
conv-curried Δ Γ f = uncurry f (makeExpVars Γ Δ) (Γ ▷▷ Δ)

```

```

conv-prod : (c : ProdCode) {i : I} → AlgProd c (ExpP c i)
conv-prod one = λ Γ → one
conv-prod (rec y) = λ x Γ → rec (x Γ)
conv-prod (c1 ⊗ c2) = map-prod c1 (λ x → map-prod c2
  (λ y Γ → x Γ ⊗ y Γ) (conv-prod c2)) (conv-prod c1)
conv-prod (abs c i) = map-prod c (λ {Δ1} x {Δ2} y Γ →
  abs (x Γ) (conv-curried Δ1 Γ y)) (conv-prod c)
conv-prod (bsng y) = λ Γ → bsng
conv-prod (bseq c i) = map-prod c (λ {Δ1} x {Δ2} y Γ →
  bseq (x Γ) (conv-curried Δ1 Γ y)) (conv-prod c)
conv-prod (brec i) = λ Δ f Γ → brec (conv-curried Δ Γ f)

```

7.5 Example: Let bindings

We give smaller examples of the PHOAS interpretation for the λ -calculi with different variants of let-bindings from Section 6. 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 get the following sets as the universe arguments.

```

data S : Set where
  term : S
data I : Set where
  decl decls term : I

```

For the types from Section 6.2 we have the following universe codes.

```

lam-app = rec term ⊗ rec term ▶ term
lam-abs = abs (bsng term) term ▶ term
lam-let = abs (rec decls) term ▶ term
decl-decl = rec term ⊗ bsng term ▶ decl
decls-nil = one ▶ decls
decls-cons = rec decl ⊗ rec decls ▶ decls

```

All calculated PHOAS types are indexed by a binder context, even types for sorts like λ -expressions, which do not bind variables. For these values the binder context is empty. Thus the interface performs unnecessary binder context calculations.

```

record LamAlgebra (A : I → Ctx → Set) : Set where

```

```

open Church A
Lam = A term
Decl = A decl
Decls = A decls
field
  appPH : {Δ1 : Ctx} → Lam Δ1 →
    {Δ2 : Ctx} → Lam Δ2 → Lam (Δ1 ▷▷ Δ2)
  absPH : {Δ : Ctx} → (Lam ε → Lam Δ) → Lam Δ
  letPH-inPH- : {Δ1 : Ctx} → Decls Δ1 →
    {Δ2 : Ctx} → Curried Δ1 (Lam Δ2) → Lam Δ2
  declPH : {Δ : Ctx} → Lam Δ → Decl (Δ ▷ term)
  dnilPH : Decls ε
  dconsPH : {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.

```

infixr 1  $\_|\_$ 
 $\_|\_ : \forall \{\Delta\} \rightarrow \text{Lam } \epsilon \rightarrow \text{Decls } \Delta \rightarrow \text{Decls } (\epsilon \triangleright \text{term } \triangleright \triangleright \Delta)$ 
 $e | ds = \text{dcons}_{\text{PH}} (\text{decl}_{\text{PH}} e) ds$ 
syntax  $\text{abs}_{\text{PH}} (\lambda x \rightarrow e) = \lambda x \Rightarrow e$ 

```

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.

```

test : LamPHOAS
test = letPH
  ( $\lambda x \Rightarrow x$ ) |
  ( $\lambda x \Rightarrow \lambda y \Rightarrow x$ ) |
  ( $\lambda f \Rightarrow \lambda g \Rightarrow \lambda x \Rightarrow$ 
    appPH (appPH f x) (appPH g x)) | dnilPH
inPH ( $\lambda I K S \rightarrow \text{app}_{\text{PH}} I K$ )

```

A downside of the HOAS representation of let bindings is that the variable names are syntactically separated from the expressions of the declarations in the source code. Function `convert` converts the above term to the following de Bruijn representation

```

letDB
  (dconsDB (declDB (absDB (v 0))))
  (dconsDB (declDB (absDB (absDB (v 1)))))
  (dconsDB (declDB (absDB (absDB (absDB
    (appDB (appDB (v 2) (v 0)) (appDB (v 1) (v 0)))))))
  dnilDB))
  (appDB (v 2) (v 1))

```

7.6 Example: Sequential let bindings

For the sequential let expressions of Section 6.4 the code for cons of declarations is

```

decls-cons = bseq (rec decl) decls ▶ decls

```

The other codes for the datatypes are the same as in Section 7.5. and the field in the `LamAlgebra` record changes to.

```

record LamAlgebra (A : I → Ctx → Set) : Set where
  field
  ...
  dconsPH : { $\Delta_1 : \text{Ctx}$ } → Decl  $\Delta_1 \rightarrow$ 
    { $\Delta_2 : \text{Ctx}$ } → Curried  $\Delta_1$  (Decls  $\Delta_2$ ) →
    Decls ( $\Delta_1 \triangleright \triangleright \Delta_2$ )
  ...
infixr 1 dcons'PH
dcons'PH : { $\Delta : \text{Ctx}$ } → Lam  $\epsilon \rightarrow$ 
  Curried ( $\epsilon \triangleright \text{lam}$ ) (Decls  $\Delta$ ) → Decls ( $\epsilon \triangleright \text{lam} \triangleright \triangleright \Delta$ )
dcons'PH e ds = dconsPH (declPH e) ds
syntax  $\text{abs}_{\text{PH}} (\lambda x \rightarrow e) = \lambda x \Rightarrow e$ 
syntax  $\text{dcons}'_{\text{PH}} e (\lambda x \rightarrow ds) = x := e | ds$ 

```

We give an example of a sequential let. We first declare a combinator `t` that is used in the second declaration. Note that scoping occurs at every declaration so for each declaration there is immediately a higher-order binding scoping over the subsequent declarations. The abstraction over the body is represented by a separate higher-order binding. The use of two higher-order bindings entails duplication of variable binding and does not enforce equal names for the variables.

```

test : LamPHOAS
test = letPH
  t := ( $\lambda x \Rightarrow x \cdot x$ ) |
   $\omega := t \cdot t$  | dnilPH
inPH ( $\lambda t \omega \rightarrow \omega$ )

```

Function `convert` converts the above term to the following de Bruijn representation

```

convert test  $\epsilon \equiv \text{let}' (\text{dcons} (\text{decl} (\text{abs} (\text{app} (v 0) (v 0))))
  (\text{dcons} (\text{decl} (\text{app} (v 0) (v 0))))
  dnil))
  (v 0)$ 

```

An inconvenience in the chosen PHOAS type for `bseq` is that every `bseq` introduces a function for the complete accumulated binder context of its left component. 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 `init` over the last.

7.7 Example: Recursive let bindings

For recursive lets we modify the representation of the λ -calculus with lets from Section 7.5. The binder context of a let group is reintroduced in the term context of the declarations.

```

data Lam : Ctx → Ctx → Set where
  ...
  letDB :  $\forall \{\Gamma \Gamma^+ \Delta\} \rightarrow \text{Decls } \Gamma^+ (\Gamma \triangleright \triangleright \Gamma^+) \rightarrow$ 
    Lam  $\Delta (\Gamma \triangleright \triangleright \Gamma^+) \rightarrow \text{Lam } \Gamma$ 

```

And we get the following new code for the let constructor.

```

lam-let = abs (brec decls) term ▶ term

```

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.

```

letPH : ( $\Delta : \text{Ctx}$ ) → Curried A  $\Delta$  (Decls  $\Delta$ ) →
  { $\Gamma_2 : \text{Ctx}$ } → Curried A  $\Delta$  (Lam  $\Gamma_2$ ) → Lam  $\Gamma_2$ 

```

We give an example of a recursive let with two mutually recursive functions. Note that in general it is not possible to determine a binder context automatically. We have to provide it explicitly to the algebra function of let. As for the sequential let example we have two higher-order bindings. One binding introduces all the variables for the declaration, and one introduces the variables for the body.

```

test : LamPHOAS
test = letPH ( $\epsilon \triangleright \text{term} \triangleright \text{term}$ )
  ( $\lambda a b \rightarrow (\lambda x \Rightarrow b)$  |
    ( $\lambda x \Rightarrow \text{app}_{\text{PH}} x a$ ) | dnilPH)
  ( $\lambda a b \rightarrow \text{app}_{\text{PH}} b a$ )

```

Function `convert` converts the above PHOAS term to the following de Bruijn term

```

letDB (dconsDB (declDB (absDB (v 1))))
  (dconsDB (declDB (absDB (appDB (v 0) (v 2)))))
  dnilDB))
  (appDB (v 0) (v 1))

```

Again names are not enforced to coincide. In this case it is possible to add another constructor `absrec i1 i2` to `ProdCode`, that is equivalent to `abs (brec i1) i2`, i.e. variables from the first code scope recursively over values defined by the index `i1`, and over values defined by the second index `i2`. The interpretation of `absrec` uses a single higher-order binding for both the declarations, and the let body.

8. Conclusion

In this paper we have presented a universe for dependently typed datatype-generic programming with abstract syntax representations that is very expressive, in the sense that it supports well-scoped representations and also allows rich binding forms.

Two interpretations have been given and a generic conversion function has been implemented that simplifies the implementation of EDLSs with binding.

8.1 Related work

A datatype-generic treatment of syntax with binding has been addressed before in the literature.

Cheney [4] describes the implementation of a Haskell library called *FreshLib* for generic programming with abstract syntax. Cheney also 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, but he does not delve into a datatype-generic treatment of binders.

Oliveira et al. [7] are working on a datatype-generic framework GMeta for the mechanization of formal meta-theory of first-order representations. They make use of a universe construction to represent abstract syntax types and give different first-order interpretations. However, they only cover the case of single variable bindings.

Weirich, Yorgey and Sheard [12] present a domain-specific language UNBOUND and an 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 6. In fact they provide a set of type combinators very similar to the primitive codes of the binders. Internally a locally nameless style is used. The indices for bound variables are however not statically ensured to be well-scoped.

Licata and Harper [8] 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, but they only handle the single variable case.

8.2 Future work

Opportunities for future work include the automatic derivation of the structure representation from datatype declarations and corresponding conversion functions to universe types. However, doing this for well-scoped de Bruijn terms seems very difficult, since changes to the term and binder context have to be translated to structure descriptions.

Agda proved itself invaluable for the development of universes and the implementation of type computations and generic functions. However, it is not as widely used as other functional programming languages like for example Haskell. We hope that we can use the usual techniques for faking dependent types in Haskell together with the enhancements to the kind system added in GHC 7.4 to encode universes and their interpretations in Haskell and implement generic conversions for them.

References

- [1] R. Atkey. Syntax for free: Representing syntax with binding using parametricity. In P.-L. Curien, editor, *TLCA*, volume 5608 of *LNCS*, pages 35–49. Springer Berlin / Heidelberg, 2009.
- [2] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, pages 37–48. ACM, 2009.
- [3] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *DAMP '11*, pages 3–14. ACM, 2011.
- [4] J. Cheney. Scrap your nameplate: (functional pearl). In *ICFP '05*, pages 180–191. ACM, 2005.
- [5] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP '08*, pages 143–156. ACM, 2008.
- [6] D. Devriese and F. Piessens. On the bright side of type classes: instance arguments in agda. *ICFP '11*, pages 143–155. ACM, 2011.
- [7] G. Lee, B. C. d. S. Oliveira, S. Cho, and K. Yi. Gmeta: A generic formal metatheory framework for first-order representations. In H. Seidl, editor, *ESOP '12*, volume 7211 of *LNCS*. Springer, 2012.
- [8] D. R. Licata and R. Harper. A universe of binding and computation. In *ICFP '09*, pages 123–134. ACM, 2009.
- [9] U. Norell. *Towards a practical programming language based on dependent type theory*. Chalmers University of Technology, 2007.
- [10] U. Norell. Dependently typed programming in agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2009.
- [11] G. Washburn and S. Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *ICFP '03*, pages 249–262. ACM, 2003.
- [12] S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *ICFP '11*, pages 333–345. ACM, 2011.
- [13] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09*, pages 233–244. ACM, 2009.