DEPENDENT POLYVARIADIC FUNCTIONS

Jan Malakhovski

graduate student at NRU ITMO; trojan@rain.ifmo.ru

Abstract: The proposition that polyvariadic functions could be defined within dependently typed setting is a folklore in the functional programming community. We could not, however, find any explicit evidence of this fact.

Here we present an explicit implementation of this folklore problem and propose a novel approach for implementing programs usually defined using templates with dependently typed polyvariadic functions.

It is impossible to replace conventional term generation techniques altogether with our approach, because it does not provide any means for reflection. However, it covers a considerable amount of code usually generated with them, yet does not require any special support from the compiler and is completely orthogonal to other type-system extensions.

Our approach is formalized in *Agda* programming language.

Introduction

In functional programming languages, such as Haskell [1], polymorphic data types and functions are usually defined by the abstraction over type variables. In mainstream imperative languages, such as C++ [2], generic programming is done with overloading and different types of templates. Functions that are generic not only by types of variables they receive, but by the *structure* of data they process (e.g. `show` function from `Show` type class in Haskell) as well, are usually defined directly by the compiler/interpreter (e.g. *deriving* construction in Haskell, `str` and `repr` functions in Python [3]) or with templates and reflection.

From type theoretic point of view, templates are functions from *non-ground* type universe, and so their execution is done at compile time. Indeed, templating engines of all mainstream programming languages use methods that are very close to β-reduction from lambda calculus. This property means that conventional generic programming leads to code duplication at different levels, because every pure function has at least two representations: an imperative one for runtime, and a functional one for template engine.

Reflection machinery like Template Haskell [4], that represents terms of language with a description that does not describe itself, usually does not allow to reflect terms that operate on the description language level and above, because each new level of such reflection needs a new set of operational primitives. The work [5] proposes a possible self-supporting representation for datatypes, which in effect allows to describe description language in terms of base language without any external primitives.

... Data-generic programming thus becomes ordinary programming. ...

There is, however, a special case of generic programming, which apparently could not be described by templates with simple substitution model, yet does not need any extensions of conventional dependent type systems: polyvariadic programming [6], i.e. programming generic by the number and/or types of function arguments.

Classical examples of this kind of functions are:

- `printf` function from *C* programming language, which could be type-safely defined with Template Haskell or Variadic Templates from the new C++11 standard [7];
- numerous *lifting* functions, which lift plain *n*-ary functions into *n*-ary functions that preprocess their arguments before applying them to a given plain function.

Here, we propose and study typing schemes for functions polymorphic by the number of their arguments. We directly formalize our definitions in universe polymorphic way within dependently typed Agda programming language [8]. We define base combinators, which allow us to describe desired functions in close to *point-free* style, deliberately avoiding definitions that need anything more powerful than natural induction. In doing so, we pinpoint any subtle properties of type systems that complicate our definitions. Once we defined all obvious combinators arising from our typing scheme, we show how one can use them to define lifting functions on the concrete example of embedded domain specific language (*eDSL*) for logical expressions over arbitrary data types.

We aim to provide simple usable combinators for implementing functions commonly written with templates and reflection, yet without using this machinery. Every generalization (universe polymorphism, indexing) we introduce is due to real use cases we have implemented with our approach.

Each section of this document is *Literate Agda* program with module and import declarations stripped for space-saving. Everything missing on listings is defined in the standard library.

Basic Combinators

Consider the simplest possible universe polymorphic definition of the function composition:

```
infixr 9 _∘′_
_∘′_ : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁}
         {B : Set ℓ₂} {C : Set ℓ₃}
     → (B → C) → (A → B) → (A → C)
f ∘′ g = λ x -> f (g x)
```

From the point of view of its first argument (function `f`) function composition changes the domain of `f`. Dually, from the point of view of function `g` it changes its the codomain. In other words, we could argue that function composition is actually a specialized form of two different functions (pseudo-Agda code):

```
changeDomain : (X → A) → (A ~> B) → (X ~> B)
changeCodomain : (B → X) → (A ~> B) → (A ~> X)
```

where the second argument of each of these functions is an element (arrow) of some category. I.e. `changeCodomain` is a `fmap` (functor map) between coslice categories of the form `(A ~>)` and `changeDomain` is a dual to `fmap` for slice categories `(~> B)`.

There are two most simple (without introducing new type variables) possible generalizations of the type `A → B` within the function space:

- `A → A → ··· → A → B` where `A` is repeated *n* times. We'll write types like that as `A ^ n → B` from here on.
- `A → B → B → ··· → B` where `B` is repeated *m* times.

Note, however, that the latter type is equivalent to `A → (B ^ (m - 1) → B)`, which is a function from `A` to an element of the special case of the former type. That means there is only one simple general form: `A ^ n → B`.

We would like to define this form directly in Agda in universe polymorphic way:

```
_^_→_ : ∀ {a b} → Set a → N → Set b → Set (a ⊔ b)
A ^ zero → B = B
A ^ (suc n) → B = A → (A ^ n → B)
```

but Agda's universe hierarchy is not cumulative and this code wouldn't type check. We could fix that by lifting `B` with `Lift` type from Agda standard library's `Level` module. However, in this case, lifting and lowering would significally complicate all our following definitions. That's why we'll make our definition a bit less general by disallowing types without at least one arrow.

```
_^_→_ : ∀ {a b} → Set a → N → Set b → Set (a ⊔ b)
A ^ zero → B = A → B
A ^ (suc n) → B = A → (A ^ n → B)
```

Note that we could generalize this type more by introducing more complicated type schemes. E.g., in the most obvious case, by allowing indexing by `Vec`tors of elements of `Set`s to generate types like `A → B → A → B → ··· → C` and alike and/or making `A`s and `B`s dependent on each other. In that follows we show that at least these simple generalizations in conjunction with simple base combinators are just useful syntax sugar, not the theoretical generalization.

We can now define our `change(Co)Domain` functions:

```
cdom : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁}
         {B : Set ℓ₂} {C : Set ℓ₃} n
     → (A → B) → (B ^ n → C) → (A ^ n → C)
cdom zero g f = f °′ g
cdom (suc n) g f = λ x -> cdom n g $ f °′ g $ x

ccodom : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁}
           {B : Set ℓ₂} {C : Set ℓ₃} n
       → (B → C) → (A ^ n → B) → A ^ n → C
ccodom zero f g = f °′ g
ccodom (suc n) f g = λ x -> ccodom n f (g x)
```

Note that because of our decision on evading lifting we have to deal with "off by one" definitions.

If universe hierarchy of our base language were cumulative (or lifting were automatic), we would be able to give simpler definitions for the first (`zero`) cases to both of these and for every following function. On the other hand, repetition of subterms in definitions above shows relation between those two functions and function composition more directly.

Functions `_°_` (usual dependent function composition) and `_°′_` (its special form) could be used interchangeably in all our definitions from this section. Starting from the next listing we'll use `_°_` instead of `_°′_` for simplicity and ease of future abstractions.

Let us utter the programming way to describe transformations (for all $n > 0$) `cdom (n-1)` and `ccodom (n-1)` functions do:

- `cdom (n-1)` takes two functions `g` and `f` (first is one argument function, later has *n* arguments) and returns an *n*-ary function that applies `g` to every argument before passing it to `f`;
- `ccodom (n-1)` takes two functions `f` and `g` (as before, 1-ary and *n*-ary) and returns an *n*-ary function that applies its *n* arguments to `f` and then gives the result to `g`.

Next, we shall define a generalization of `flip` function. Conventional `flip` swaps the order of first two arguments of a function. Our generalization `cflip n`[1] flips the first argument of a given function with *(n+1)* following arguments. I.e. it transforms a function of type `E → A ^ n → B` into a function of type `A ^ n → (E → B)`.

```
cflip : ∀ {ℓ₁ ℓ₂ ℓ₃} {E : Set ℓ₁}
          {A : Set ℓ₂} {B : Set ℓ₃} n
```

[1] In "cdom" and "ccodom" prefix "c" stands for "change", here we would like to use prefix "g" for "generalized", but we've chosen to spell it as "counted" for uniformness.

```
      → (E → A ^ n → B) → (A ^ n → (E → B))
cflip zero f = λ x e -> f e x
cflip (suc n) f = λ x -> cflip n (λ e -> f e x)
```

Last base function combinator we need is the parallel composition, `cparcomp`, of two $(n+1)$-ary functions. I. e. a function that transforms two functions into one function operating on pairs.

```
cparcomp : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄}
             {A : Set ℓ₁} {B : Set ℓ₂}
             {A′ : Set ℓ₃} {B′ : Set ℓ₄} n
         → (A ^ n → B) → (A′ ^ n → B′)
         → (A × A′) ^ n → (B × B′)
cparcomp zero f g = λ p
  -> (f ° proj₁ $ p) ,′ (g ° proj₂ $ p)
cparcomp (suc n) f g = λ p
  -> cparcomp n (f ° proj₁ $ p) (g ° proj₂ $ p)
```

Type `A × B` is a direct product of types `A` and `B`, constructor `_,′_` is a constructor for `×`, $\texttt{proj}_1$ and $\texttt{proj}_2$ are the eleminators for `×`.

An eDSL Example

Suppose, we're writing a system which has a lot of logic in the subject domain. With `if`, `with` (`case` in *Haskell*) and *guard* constructions from usual functional programming languages, even dependent ones, there is no easy way to check that the given program is total, has no not contradictions and doesn't have dead branches in the presence of arbitrary logical expressions. In Turing-complete language like *Haskell* it's completely impossible in general case. With total languages like *Agda* it's hard to prove first two properties because every input should be examined case by case and it is impossible to prove the last one because there is no notion of reachability. In the end, programmer is forced to write either unchecked code or boiler-plate, that checks every possible input.

Note however, that if our subject domain allows us to restrict ourselves to `Bool`ean variables only, we can use any SAT-solver to check for totality, contradictions and dead branches explicitly. For such use cases we would like to have an eDSL with flat construction similar to `with` (or `case`) which allows us to write code like (pseudo-Agda):

```
example0 = select $
  X1 ·∧ X2      —> ...
  X1 ·∧ ·not X2 —> ...
  ·not X1       —> ...
```

To implement that, we shall define data type for logic expressions:

```
data Logic (V : Set) : Set where
  Latom : V → Logic V            -- Variable
  Ltrue Lfalse : Logic V
  Lnot : Logic V → Logic V
  Land Lor : Logic V → Logic V → Logic V
```

which could be used with arbitrary type of variables:

```
data Var : Set where
  X1 X2 X3 : Var

example1 : Logic Var
example1 = Lor
  (Land (Latom X1) (Latom X2))
  (Lor (Latom X3) (Lnot (Latom X1)))

example2 : Logic Var
example2 = (Land (Latom X1) (Latom X3))
```

Correctness checking

Amusingly, if we overlook horrible syntax, that simple definition is already usable for our purposes. Given *environment*, a function from variables to values, we could compute the value of a given logical expression:

```
compute : ∀ {V} → Logic V → (V → Bool) → Bool
compute (Latom v) e = e v
compute (Ltrue) _ = true
compute (Lfalse) _ = false
compute (Lnot x) e = not $ compute x e
compute (Land x1 x2) e =
  (compute x1 e) ∧ (compute x2 e)
compute (Lor x1 x2) e =
  (compute x1 e) ∨ (compute x2 e)
```

and check satisfability of a formula with simple SAT-solving algorithm[1]:

```
-- Boolean algebra of satisfying sets
SSet : Set → Set
```

[1] The idea of this algorithm is based on “sequent calculus” [9] derivation search algorithm. The original method is defined imperatively and can’t be directly translated into Agda program passing the termination checker. We reformulated it in terms of Boolean algebras and implemented in less efficient, but well-formed recursive way.

```
   SSet V = List (List V)

   -- "0" element
   szero : ∀ {V} → SSet V
   szero = []

   -- "1" element
   sone : ∀ {V} → SSet V
   sone = [ [] ]

   -- Lift from V to SSet
   ssingle : ∀ {V} → V → SSet V
   ssingle v = [ [ v ] ]

   -- Union
   _sor_ : ∀ {V} → SSet V → SSet V → SSet V
   x sor y = x ++ y

   -- Intersection
   _sand_ : ∀ {V} → SSet V → SSet V → SSet V
   x sand y = concatMap (λ xel -> map (λ yel -> (xel ++
yel)) y) x

   -- Now we define calculation of satisfying set in a
well-defined
   -- recursive manner.
   -- First Bool argument of a function: whenever ex-
pression we analize

   -- should be true of false.
   calc : ∀ {V} → Bool → Logic V → SSet (Bool × V)
   calc any (Latom x) = ssingle (any , x) --
   calc true   Ltrue = sone
   calc false  Ltrue = szero
   calc true  Lfalse = szero
   calc false Lfalse = sone
   calc any (Lnot x) = calc (not any) x
   calc true  (Land x1 x2) =
     (calc true x1)  sand (calc true x2)
   calc false (Land x1 x2) =
```

```
    (calc false x1) sor  (calc false x2)
  calc true  (Lor x1 x2) =
    (calc true x1)  sor  (calc true x2)
  calc false (Lor x1 x2) =
    (calc false x1) sand (calc false x2)

  -- Given satisfying set transform
  -- it into two sets:
  -- 1. variables that should be true
  -- 2. variables that should be false
  trans : ∀ {V : Set} → List (Bool × V)
        → List V × List V
  trans xs = go xs [] [] where
    go : ∀ {V} → List (Bool × V) → List V
       → List V → List V × List V
    go [] ts fs = ts , fs
    go ((true  , x) ∷ xs) ts fs = go xs (x ∷ ts) fs
    go ((false , x) ∷ xs) ts fs = go xs ts (x ∷ fs)

  private
    module Dummy {V : Set}
                 (_==_ : Decidable _≡_) where
      _===_ : V → V → Bool
      a === b with a == b
      ... | (yes _) = true
      ... | (no _) = false

      _elem_ : V → List V → Bool
      x elem ls = or $ map (_===_ x) ls

      _intersects_ : List V → List V → Bool
      ls intersects rs =
        or (map (flip _elem_ rs) ls)

      -- Given logical expression calc it and filter
      -- out contradictive satisfying sets (same
      -- variable should be true and false).
      solve : Logic V → List (List V × List V)
      solve = filter (not ° uncurry' _intersects_)
            ° map trans ° calc true

  open Dummy public using (solve)
```

Now, given a list of pairs `Logic V × E` (`E` is arbitrary, `V` has decidable propositional equality), where type `E` is a type of result expressions, we could implement the following functions (they are trivial and we don't give their definitions for space-saving):

```
AreTotal : ∀ {ℓ} {E : Set ℓ} {V : Set}
         → List (Logic V × E) → Set
AreTotal = {!!}

AreSatisfyable : ∀ {ℓ} {V : Set} {E : Set ℓ}
               → List (Logic V × E) → Set
AreSatisfyable = {!!}

AreNonContradictional : ∀ {ℓ} {V : Set}{E : Set ℓ}
                      → List (Logic V × E) → Set
AreNonContradictional = {!!}


select : ∀ {ℓ} {V : Set} {E : Set ℓ}
  → (l : List (Logic V × E))
  → {_ : AreTotal l} → {_ : AreSatisfyable l}
  → {_ : AreNonContradictional l}
  → (V → Bool) → E
select = {!!}
```

`AreTotal` checks that a disjunction of all first elements of pairs of a list is a tautology (negation is not solvable). If it is a tautology, `AreTotal` returns ⊤ (the type with only one element), otherwise it returns ⊥ (the type without elements). For ⊤ (top) compiler infers the only element and substitutes it, for ⊥ (bottom) it generates an unsolved constraint. Moreover, we could return not just the bottom type, but a type isomorphic to bottom, which has an error description packed into its term, if we wish. This description would be then printed in an unsolved constraint message generated by the compiler. Respectively, `AreSatisfyable` checks that every element could be satisfied (no dead branches) and `isNonContradictional` checks that a pairwise conjunction of arbitrary two `Logic V` expressions is not satisfiable (each input defines exactly one output). `select` linearly scans through the list and returns the second element of the first (and the only, by the definition) pair which has `true` as a result of `compute`.

We might also consider sugaring the syntax for defining lists of these pairs. To obtain a syntax similar to the described above, it's enough to define a new constructor for pairs:

```
infix 10 _—>_
_—>_ : ∀ {ℓ} {V : Set} {E : Set ℓ} → Logic V
     → E → Logic V × E
a —> b = a ,′ b
```

Example usage:

```
exampleDef = select $
  example1 —> 0 ::
  example2 —> 1 ::
  []
```

This would look scary if we substitute the definitions of `example1` and `example2`, but it already checks all the desired properties at compile time.

Syntax sugar

In order to rewrite logical expressions with the following desired syntax sugar:

```
example1′ : Logic Var
example1′ = X1 ·∧ X2 ·∨ X3 ·∨ ·not X1

example2′ : Logic Var
example2′ = ·not X1 ·∧ X3
```

we should write infix functions `·∧`, `·∨` and `·not` which are able to receive arguments of different types. For example, in the code above `·∧` receives the value of type `Var` at the first occurrence and the value of type `Logic Var` at the second.

In Haskell this problem could easily be solved by typeclasses. In Agda, a type class becomes a dependent tuple of elements `A : Set` and `A × I A`, i.e. the type `A`, the element of this type and the implementation of the interface `I` for this type. With that knowledge we could define needed functions right away, but let us check what else we don't like in our definitions first.

Clearly, we don't like our definitions around `select` because they interpret logical expressions with `compute`, also we don't like `select` that makes a linear search. For the later problem we don't know any simple solutions. For the former problem we could do better by lifting native Agda's or Haskell's `Bool`ean functions into functions of type `(V → Bool) → Bool`.

For this purpose we need to define the interface:

```
record Liftable (V : Set) (F : Set) : Set where
  field
    llift : F → Logic V
    flift : F → (V → Bool) → Bool
```

`llift` lifts (transforms) F into its `Logic`al representation, `flift` lifts `F` into a function from any environment to `Bool`.

We still wish to analyze our expressions, thus we shall store their logical and functional representations at the same time:

```
AlreadyLifted : Set → Set
AlreadyLifted V = (Logic V) × ((V → Bool) → Bool)
```

We are now able to define our type class in Agda:

```
ELiftable : Set → Set → Set
ELiftable V L = L × Liftable V L

CLiftable : Set → Set1
CLiftable V = Σ Set (ELiftable V)

llift : ∀ {V} → CLiftable V  → Logic V
llift (t , a) = uncurry (flip $
  Liftable.llift {F = t}) a

flift : ∀ {V} → CLiftable V → (V → Bool) → Bool
flift (t , a) = uncurry (flip $
  Liftable.flift {F = t}) a
```

and the default implementations:

```
mkAnyLiftable : ∀ V → Liftable V V
mkAnyLiftable _ = record
  { llift = Latom
  ; flift = λ v e → e v }

mkLogicLiftable : ∀ V → Liftable V (Logic V)
mkLogicLiftable _ = record
  { llift = id
  ; flift = compute }

mkAlreadyLiftable : ∀ V
                  → Liftable V (AlreadyLifted V)
mkAlreadyLiftable _ = record
  { llift = proj₁
  ; flift = proj₂ }
```

Lifting $(n+1)$-ary Boolean function `f` into a function from an environment to `Bool` is simple: given an environment `e`, we need to lift every argument into a

function, apply `e` to each of them, and then apply results to `f`. That is exactly what `cdom` function does. After this transformation we end up with a function of the type `(V → Bool) → A → ··· → A → Bool` (where each `A` is a specialization of `CLiftable`).

`cflip`ping will convert it into a function with the desired type `··· → (V → Bool) → Bool`:

```
lift2Func : ∀ {V} n → (Bool ^ n → Bool)
          → (CLiftable V) ^ n → ((V → Bool) → Bool)
lift2Func n f = cflip n
  (\e -> cdom n (flip flift e) f)
```

Lifting into `Logic`al expression is a bit more complicated. For a single argument function we'll write the resulting logic expression explicitly. Functions of bigger arity are obtained by:

- abstracting by the first argument of the resulting expression (`a`),
- fixing first argument of `f`, lifting resulting expression, then `cflip`ping (to move `Bool` argument behind arguments of the lifted version of a function),
- applying all other arguments (here we end up with a function of the type `Bool → Logic V`)
- and then constructing resulting expression from two possible values of `a`.

The last two steps are exactly what `ccodom` function does. The resulting expression is defined by:

```
lift2Logic : ∀ {V} n → (Bool ^ n → Bool)
           → (CLiftable V) ^ n → Logic V
lift2Logic zero f = repr f where
  repr : ∀ {V} → (Bool → Bool) → CLiftable V →
Logic V
  repr f a with f true | f false
  ... | true  | true  = Ltrue
  ... | true  | false = llift a
  ... | false | true  = Lnot $ llift a
  ... | false | false = Lfalse
lift2Logic (suc n) f = λ a →
  ccodom n (lift2L a) ° cflip n $
 λx -> lift2Logic n (f x) where
  lift2L : ∀ {V} → CLiftable V
         → (Bool → Logic V) → Logic V
  lift2L a liftN = Lor
    (Land (llift a)         (liftN true))
    (Land (Lnot $ llift a) (liftN false))
```

Note again, that this definition would be less complicated if we had cumulative universe hierarchy or automatic lifting between universes.

Now, lifting any Boolean function to `AlreadyLifted` is just a parallel composition of `lift2Logic` and `lift2Func`:

```
lift2Already : ∀ {V} n → (Bool ^ n → Bool)
             → (CLiftable V) ^ n → AlreadyLifted V
lift2Already n f = cdom n (λ x → (x ,' x)) $
  cparcomp n (lift2Logic n f) (lift2Func n f)
```

That `lift2Already` still won't give us our desired syntax because we'll have to wrap every value into `CLiftable` by hand. Implicit arguments and instance arguments should solve this problem, but there is no way to use them with our current type scheme.

Therefore, we shall define a type scheme for functions which take $(n+1)$ arguments of arbitrary type, where each of them has an implementation of interface somewhere in the context:

```
_^I_→_ : ∀ {ℓ₁ ℓ₂ ℓ₃} → (Set ℓ₁ → Set ℓ₂)
       → N → Set ℓ₃ → Set (Level.suc ℓ₁ ⊔ ℓ₂ ⊔ ℓ₃)
_^I_→_ {ℓ₁} I zero B =
  {A : Set ℓ₁} → (a : A) → {{i : I A}} → B
_^I_→_ {ℓ₁} I (suc n) B =
  {A : Set ℓ₁} → (a : A) → {{i : I A}}
  → (I ^I n → B)
```

We are able to define a definition similar to `cdom` to adapt this type scheme to our previous type scheme:

```
cidom : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄} {B : Set ℓ₂}
          {C : Set ℓ₃} {I : Set ℓ₁ → Set ℓ₄}
      → (n : N) → (Σ (Set ℓ₁) (λ A → A × I A) → B)
      → (B ^ n → C) → (I ^I n → C)
cidom zero g f = λ {A} a {{I}}
  -> f ° g $ (A , (a , i))
cidom (suc n) g f = λ {A} a {{i}}
  -> cidom n g $ f ° g $ (A , (a , i))
```

We could define a backward transformation in a similar way, but we shall skip this definition for space-saving. From a philosophical standpoint of view this bijection means that our type scheme is orthogonal to type system extensions which infer elements from a context. It's even orthogonal to obvious extensions which generate types like `A → B → A → B → ⋯ → C` because we can pack them into tuples in the same way.

We are now able to define `lift` function with a desired syntax:

```
NLiftable : Set → N → Set1
NLiftable V n = (Liftable V) ^I n
  → (AlreadyLifted V)

lift : ∀ {V} n → (Bool ^ n → Bool) → NLiftable V n
lift n f = cidom n id (lift2Already n f)
```

and test it:

```
infixl 80 _·∧_
_·∧_ : ∀ {V} → NLiftable V 1
_·∧_ = lift 1 _∧_

infixl 70 _·∨_
_·∨_ : ∀ {V} → NLiftable V 1
_·∨_ = lift 1 _∨_

·not : ∀ {V} → NLiftable V 0
·not  = lift 0 not

-- Everything before, except Var data type
-- definition, should be considered a library
-- code.
-- Everything from here and till the end of this
-- listing (and Var) is that a user of
-- this library might write.

-- We have to do the following for every type
-- like Var
-- because Agda doesn't specialize functions which
-- generate instances for instance arguments,
-- even if arguments are implicit.

tmpx = mkAnyLiftable Var
tmpy = mkLogicLiftable Var
tmpz = mkAlreadyLiftable Var

example1' : AlreadyLifted Var
example1' = X1 ·∧ X2 ·∨ X3 ·∨ ·not X1

test1 : proj₁ (X1 ·∧ X2) ≡ Lor
  (Land (Latom X1) (Latom X2))
```

```
  (Land (Lnot (Latom X1)) Lfalse)
test1 = refl

example1'env : Var → Bool
example1'env x with x
... | X1 = false
... | X2 = true
... | X3 = false

test2 : proj₂ example1' example1'env ≡ true
test2 = refl
```

Conclusion

This work has provided a simple explicit handling for polyvariadic functions in dependently typed setting. The combinators presented here make it possible to describe them with expressions in a point-free style. The example usage of the combinators is shown on the moderately sized example of logical eDSL.

The latter shows that presented approach covers most usages of conventional template engines: our `lift` function simultaneously generates `Logic`al expressions that are to be checked at compile time and the code to be executed at runtime.

The type scheme for dependent polyvariadic functions can be straightforwardly enhanced by type indexing, which allows to describe even more rich set of functions. Research in this area would be an important next step.

References

1. *Hudak P., Peterson J., Fasel J.* A Gentle Introduction To Haskell, Version 98. 2000. http://www.haskell.org/tutorial/ [date: 07.04.2012].
2. *Stroustrup B.* The C++ Programming Language (Third ed.). Boston: Addison–Wesley, 1997.
3. Python Programming Language. http://www.python.org/ [date: 07.04.2012].
4. Template Haskell. http://www.haskell.org/haskellwiki/Template_Haskell [date: 07.04.2012].
5. *Chapman J., Dagand P., Mcbride C., Morris P.* The Gentle Art of Levitation // ICFP '10 Proceedings of the 15th ACM SIGPLAN international conference on Functional programming. 2010. P. 3–14.
6. *Kiselov O.* Polyvariadic functions and keyword arguments. http://okmij.org/ftp/Haskell/polyvariadic.html [date: 07.04.2012].
7. C++0x/C++11 Support in GCC. http://gcc.gnu.org/projects/cxx0x.html [date: 07.04.2012].
8. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php [date: 07.04.2012].
9. *Sørensen M. H. B., Urzyczyn P.* Lectures on the Curry-Howard Isomorphism. Elsevier Science, 2006.