Dependently Typed Programming in Cayenne

or

Why does Agda look so strange?

Lennart Augustsson

lennart@augustsson.net

www.dependent-types.org

An example

We want to write a small program that does bracket abstraction for λ-calculus.

```
data Exp = App Exp Exp
         | Lam Sym Exp
         | Var Sym
type Sym = String
```

The function we want will remove all λ-expressions and replace them with the S, K, and I combinators. We could give it this type:

```
abstractVars :: Exp -> Exp
```

This does not reflect that all `Lam` constructors are gone.

Bracket abstraction

Remove all λ-expressions by using combinators.

I x = x

K x y = x

S f g x = (f x) (g x)

Every lambda term is replaced by its bracket abstraction:

λ x.e = [x]e

[x]x = I

[x]y = K y

[x](f e) = S ([x]f) ([x]e)

An example

Use a different result type.

```
abstractVars :: Exp -> LamFreeExp
```

Use a "subtype"

```
type LamFreeExp =
     sig exp :: Exp
         lf  :: LamFree exp
```

An example, a little logic

```
data Absurd =

absurd :: (a :: #) |-> Absurd -> a
absurd i = case i of { }

data Truth = truth

data (/\) a b = (&) a b
```

An example

Describe what it means to be `LamFree`:

```
LamFree :: Exp -> #
LamFree (App f a) = LamFree f /\ LamFree a
LamFree (Lam _ _) = Absurd
LamFree (Var _) = Truth
```

An example

We are all set, just proceed as usual:

```
abstractVars :: Exp -> LamFreeExp
abstractVars e@(Var _) =  struct { exp = e; lf = truth }
abstractVars (App f a) =
     let f' = abstractVars f
         a' = abstractVars a
     in  struct exp = App f'.exp a'.exp
                lf  = f'.lf & a'.lf
abstractVars (Lam x e) =
     let e' = abstractVars e
     in  abstractVar x e'.exp e'.lf
```

An example

```
abstractVar :: Sym -> (e :: Exp) -> LamFree e -> LamFreeExp
abstractVar s (App f a) (lf & la) =
    let f' = abstractVar s f lf
        a' = abstractVar s a la
    in  struct exp = App (App S f'.exp) a'.exp
               lf  = (truth & f'.lf) & a'.lf
abstractVar s (Lam _ _) l = absurd l
abstractVar s e@(Var x) l =
    if (s == x)
       ( struct {exp = I; lf :: LamFree exp = truth})
       ( struct {exp = App K e; lf :: LamFree exp = truth & l})

S = Var "S"
K = Var "K"
I = Var "I"
```

Cayenne design goals

- A **programming** language with dependent types.
- "First class" types.
- Few basic concepts.
- No *top level*.
- "Pure", i.e., the β-rule is valid.
- Uniform way to define and name things.
- Staged execution, i.e., compiled.
- All used variables must be explicitly bound.

Cayenne design goals

Lesser goals:

- No silly case restrictions on names.
- Compiled with same efficiency as Haskell.
- Proofs possible.
- Haskell like.

No top level

Many languages have a *top level* that is different. E.g., C only allows function definitions on the top level, Haskell only allows type definitions on the top level.

I want to take any program fragment and move it to where it belongs.

Example:

```
data BT a = Leaf | Node (BT a) a (BT a)
sortBy :: (a -> a -> Ordering) -> List a -> List a
sortBy cmp xs = ...
```

If the binary tree type is only used in `sortBy` it should be like this.

```
sortBy :: (a -> a -> Ordering) -> List a -> List a
sortBy cmp xs =
    let data BT a = Leaf | Node (BT a) a (BT a)
    in  ...
```

Staged execution

I want a phase distinction; execution has two phases:

- Compile time: type checking and maybe more.
- Run time: actual program execution.

Any (closed) expression should be possible to compile.

The type of an expression, A, should be the only thing needed to compile an expression, B, that is using A.

The function type

The function type is easy, we only need some syntax for dependent functions. Most of the syntax comes from Haskell.

```
\ (x :: t) -> e  ::  (x :: t) -> s
```

Application looks as usual.

```
f e  ::  s
```

If the function is not actually dependent on `x` we can use the usual Haskell syntax.

```
\ (x :: t) -> e  ::  t -> s
```

We can also usually leave out the type in the term.

```
\ x -> e  ::  t -> s
```

The function type, hidden arguments

Cayenne has the ability to "hide" arguments. This means that they need not be given when a function is applied, if the type checker can deduce them.

```
id :: (a :: #) |-> a -> a
id |a x = x

... id 5 ...
```

The sum type

We need sum types. Can we take Haskell's data type definitions?

NO

```
data T = A | B | C
```

Haskell's data type definition forces the type to have a name.

Naming things should be uniform, so if a type has a name it should be given in the same way as for anything else, possibly no name at all.

```
T :: #
T = data A | B | C
```

The sum type, constructors

Constructors are written in a peculiar way:

```
C@t  ::  t
```

Example:

```
True@(data False | True)
```

Contructors do not have any scope (just like record labels), they are only meaningful with an `@`.

The sum type, weird stuff

What's the type of this expression?

```
let T = data A | B | C
in  A@T
```

You could give a type in terms of T, but it's not in scope. I find that bizarre.

The Cayenne answer to the question is:

```
let T = data A | B | C
in  A@T                   ::  data A | B | C
```

Structural equivalence

Types are compared with structural equivalence rather than name equivalence (unlike, e.g., Haskell).

Rationale: (A) Types do not have to have names. (B) Name equivalence does not work with the β-rule.

Example:

```
List a = data Nil | Cons a (List a)
```

Unfold `List`

```
List a = data Nil | Cons a ( data Nil | Cons a (List a))
```

According to the β-rule principle these must be considered equal.

Btw, this is also equivalent:

```
List a = Fix (\ l -> data Nil | Cons a l)
```

The sum type, case

The case construct looks mostly familiar:

```
case xs of
(Nil) -> ...
(x : xs) -> ...
```

Contructor patterns must have parenthesis around them. This is to distinguish them from variable patterns. (There is no case distinction like in Haskell.)

The dependent type system shows up in that the case arms can have different types.

```
case b of                      case b of
(False) -> 1            ::     (False) -> Int
(True) -> "Hello"              (True) -> String
```

The sum type, with plenty of sugar

To make life simpler, Cayenne allows

```
data T = C1 | C2 ...
```

which is equivalent to

```
T = data C1 | C2 ...
C1 = C1@T
C2 = C2@T
...
```

Furthermore, function definitions can be written with pattern matching (like in Haskell) instead of λ and case.

The type of types

The type of types is named # (because * is used for multiplication).

```
# :: #1 :: #2 :: ...
```

This isn't the whole story...

The record type

Records with named fields are very, very useful in programming. Their omission from the original Haskell definition is something of a mystery.

```
struct              sig
  x1 = e1      ::     x1 :: t1
  ...                 ...
  xn = en             xn :: tn
```

Record selection uses the ordinary `.´ notation.

```
s.xn  ::  tn
```

The record type

Should the record type be dependent in some way?

YES!

Consider the type theory type:

$\exists\, x \varepsilon A.\, P(x)$

which has elements of the form:

(e, P(e))

We need this in Cayenne records too.

```
sig
   x :: A
   p :: P(x)
```

Generalize: Let all labels be in scope in all types.

The record type

Since the labels have to be bound in the `sig` it's natural to have it the same way in a `struct`.

Example:

```
struct
  x = 5
  y = x + 2     -- i.e., y = 7
```

This interacts well with types too.

```
struct
  Coord = sig { x :: Int; y :: Int }
  origin = struct { x = 0; y = 0 }
  ...
```

let expressions

The `struct` expression is similar to the definition part of `let` expressions in, e.g., Haskell.

Cayenne defines `let` in terms of `struct`. (The label `r` should be fresh.)

<pre>let
    x1 = e1
    ...
    xn = en
in e</pre>
	=	<pre>(struct
		    x1 = e1
		    ...
		    xn = en
		    r = e
		).r</pre>

open expressions

A very convenient feature of Pascal (and other languages) is to "open" a record and bring its labels into scope. Cayenne defines syntactic sugar for this too. (The variable `r` should be fresh.)

```
                                      let r = e
                                          x1 = r.x1
open e use x1, ... xn in e'    =          ...
                                          xn = r.xn
                                      in  e'
```

Example:

```
open coord use x, y, z
in sqrt(x^2 + y^2 + z^2)
```

Modules

In Cayenne the record type has all the power of modules in most languages. The `sig` is used for module signatures, and `struct` for module implementation. Furthermore, ordinary functions can be used instead of (ML) functors.

```
STACK = sig
  Stack   :: # -> #
  empty   :: (a :: #) |-> Stack a
  push    :: (a :: #) |-> a -> Stack a -> Stack a
  pop     :: (a :: #) |-> Stack a -> Stack a
  top     :: (a :: #) |-> Stack a -> a
  isEmpty :: (a :: #) |-> Stack a -> Bool
```

BUT, this doesn't always work as intended...

Modules, abstract and concrete

Consider the following module for booleans.

```
struct
  Bool = data False | True
  not = ...
  ...
```

This module would have the signature

```
sig
  Bool :: #
  not :: Bool -> Bool
  ...
```

That's not right. Where are the constructors?

Modules, abstract and concrete

We can export values for them.

```
struct
  Bool = data False | True
  False = False@Bool
  True = True@Bool
  not = ...
  ...
```

This module would have the signature

```
sig
  Bool :: #
  False :: Bool
  True :: Bool
  not :: Bool -> Bool
  ...
```

That's still not right. Pattern matching would not work since there is no indication that `Bool` is actually a data type.

Modules, abstract and concrete

We need something more, we need the signature to actually tell us the *definition* of `Bool`.

Enter **`concrete`** and **`abstract`**!

```
struct
  concrete Bool = data False | True
  abstract not = ...
  ...
```

This module would have the signature

```
sig
  Bool :: # = data False | True
  not :: Bool -> Bool
  ...
```

The **`concrete`** and **`abstract`** qualifiers can be applied to any kind of fields in a record. Sensible defaults are used if they are not given.

Modules, public and private

When making modules you often need auxilliary definitions that should not be part of the visible interface of the module.

So we extend the record syntax even more, with `public` and `private`.

```
struct
  private x = 12
  public y = x * 2
```

This module would have the signature

```
sig
  y :: Integer
```

As usual, sensible defaults are provided.

Modules

Recall

```
STACK = sig
  Stack   :: # -> #
  empty   :: (a :: #) |-> Stack a
  push    :: (a :: #) |-> a -> Stack a -> Stack a
  pop     :: (a :: #) |-> Stack a -> Stack a
  top     :: (a :: #) |-> Stack a -> a
  isEmpty :: (a :: #) |-> Stack a -> Bool
```

We can now give an implementation

```
ListStack :: STACK
ListStack = struct
    abstract Stack = List
    empty = Nil
    push = (:)
    pop = tail
    top = head
    isEmpty = null
```

Modules, functors

A signature for queues

```
QUEUE = sig
  Queue   :: # -> #
  empty   :: (a :: #) |-> Queue a
  enqueue :: (a :: #) |-> a -> Queue a -> Queue a
  dequeue :: (a :: #) |-> Queue a -> Queue a
  first   :: (a :: #) |-> Queue a -> a
```

A "functor" to turn stacks into queues (very badly).

```
SQ :: STACK -> QUEUE
SQ s = struct
  open s use Stack, empty, push, pop, top, isEmpty

  abstract Queue = Stack
  empty = empty
  enqueue x xs = app xs x
  dequeue xs = pop xs
  first xs = top xs
  private
  app :: (a :: #) |-> Stack a -> a -> Stack a
  app xs y =
    if (isEmpty xs)
       (push y empty)
       (push (top xs) (app (pop xs) y))
```

Modules

A signature for stacks more in the style of, e.g., Oberon

```
Stack (a :: #) =  sig
  T         :: #
  empty     :: T
  push      :: a -> T -> T
  pop       :: T -> T
  top       :: T -> a
  isEmpty :: T -> Bool
```

```
mkListStack :: (a :: #) |-> Stack a
mkListStack |a =  struct
  T = List a
  empty = Nil
  push = (:)
  pop = tail
  top = head
  isEmpty = null
```

Modules in the world

To make modules reusable they need to have a name that is actually mapped to some external storage so they can be accessed by different programs.

Cayenne is similar to Java in how this is done.

```
module a$global$identifier = e
```

This defines a "module" in the global name space, named `a$global$identifier`.

Cayenne programs may contain free module identifiers. (They are checked at compile time, of course.)

```
... System$Integer.(+) 30 12 ...
```

A named module is a compilation unit. In fact, any kind of expression can be named, not just a `struct`.

A very simple evaluator

Consider a tiny language of typed expressions:

```
data Expr = EBool Bool | EInt Int
          | EAdd Expr Expr | EAnd Expr Expr | ELE Expr Expr
data Type = TBool | TInt
```

It has the usual typing rules:

$$\frac{}{i : \mathrm{Int}} \qquad \frac{}{b : \mathrm{Bool}} \qquad \frac{x : \mathrm{Int} \quad y : \mathrm{Int}}{x + y : \mathrm{Int}} \qquad \frac{x : \mathrm{Int} \quad y : \mathrm{Int}}{x <= y : \mathrm{Bool}} \qquad \frac{x : \mathrm{Bool} \quad y : \mathrm{Bool}}{x \& y : \mathrm{Bool}}$$

A very simple evaluator

In Haskell (without GADTs) we would have to write an evaluator like this:

```
data Value = VBool Bool | VInt Int
eval :: Expr -> Value
eval (EBool b) = VBool b
eval (EInt i) = VInt i
eval (EAdd x y) =
    case (eval x, eval y)  of
    (VInt x', VInt y') -> VInt (x' + y')
    _ -> error "eval"
...
```

The wrapping and unwrapping of the values is inefficient. We would like to write the following, but it's not well typed.

```
eval (EBool b) = b
eval (EInt i) = i
eval (EAdd x y) = (eval x) + (eval y)
...
```

A very simple evaluator

So we can try something better in Cayenne. How about?

```
eval :: (e :: Expr) -> TypeOf e
eval (EBool b) = b
eval (EInt i) = i
eval (EAdd x y) = (eval x) + (eval y)
...
```

Well, this doesn't work, because not all expressions are well typed. So we need to express the when an expression is well typed.

```
HasType :: Expr -> Type -> #
HasType (EBool _)     (TBool) = Truth
HasType (EInt _)      (TInt)  = Truth
HasType (EAdd e1 e2) (TInt)  = HasType e1 TInt  /\ HasType e2 TInt
HasType (EAnd e1 e2) (TBool) = HasType e1 TBool /\ HasType e2 TBool
HasType (ELE  e1 e2) (TBool) = HasType e1 TInt  /\ HasType e2 TInt
HasType _             _       = Absurd
```

A very simple evaluator

Now we can write an evaluator, given a proof that the term is well typed.

```
eval :: (e :: Expr) -> (t :: Type) -> HasType e t -> Decode t
eval (EBool b)    (TBool) p         = b
eval (EInt  i)    (TInt)  p         = i
eval (EAdd e1 e2) (TInt)  (p1 & p2) = eval e1 TInt  p1 +  eval e2 TInt  p2
eval (EAnd e1 e2) (TBool) (p1 & p2) = eval e1 TBool p1 && eval e2 TBool p2
eval (ELE  e1 e2) (TBool) (p1 & p2) = eval e1 TInt  p1 <= eval e2 TInt  p2
eval _            _       p         = absurd p

Decode :: Type -> #
Decode (TBool) = Bool
Decode (TInt)  = Int
```

Where do we get the proof? Well, from a type checker, of course.

This can be extended to deal with variables.

A small equality proof

Cayenne has special syntax for equality proofs.

```
(++) :: (a :: #) |-> List a -> List a -> List a
(++) (Nil) ys = ys
(++) (x : xs) ys = x : (xs ++ ys)

appendNilP :: (a :: #) |-> (xs :: List a) ->
        xs ++ Nil === xs
appendNilP (Nil) =
        Nil ++ Nil                              ={ DEF }=
        Nil
appendNilP (x : xs) =
        (x:xs) ++ Nil                           ={ DEF }=
        x:(xs ++ Nil)                           ={ appendNilP xs }=
        x:xs                                    ={ DEF }=
        Nil ++ (x:xs)
```

An example with no proofs

In C there is a very useful function, printf, which takes a varying number of arguments of varying types. I want it!

```
-- Haskell version  WRONG
printf fmt = pr fmt "" where
 pr ""            res = res
 pr ('%':'d':s) res = \i -> pr s (res ++ show i)
 pr ('%':'s':s) res = \s -> pr s (res ++ s)
 pr ('%': c :s) res =       pr s (res ++ [c])
 pr (     c :s) res =       pr s (res ++ [c])
```

Using it:

```
printf "%d(%d)" :: Int -> Int -> String
printf "hello %s!" :: String -> String
```

An example with no proofs

```
PrintfType :: String -> #
PrintfType ""            = String
PrintfType ('%':'d':cs) = Int     -> PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%': _ :cs) =            PrintfType cs
PrintfType ( _ :cs)      =            PrintfType cs

printf :: (fmt::String) -> PrintfType fmt
printf fmt = pr fmt ""

pr :: (fmt::String) -> String -> PrintfType fmt
pr ""            res = res
pr ('%':'d':cs) res = \ i -> pr cs (res ++ show i)
pr ('%':'s':cs) res = \ s -> pr cs (res ++ s)
pr ('%': c :cs) res =         pr cs (res ++ (c : Nil))
pr (c:cs)        res =         pr cs (res ++ (c : Nil))
```

Conclusions

- Cayenne was reasonable successful design (in my opinion).
- It needs more work to become a useful programming language.
- Things I would do differently next time:
  - The language should have Agda's `idata`.
  - Stratified universes are just a pain for programming, use `#:: #`. (And use global data flow analysis to get rid of types and proofs.)
  - Type error messages need to be much better.
- I want to see more examples that are totally proof free, but uses dependent types in an essential way (like printf). There are many examples of dependent vector sizes, but they usually require a complicated constraint solver.

Try it!

Cayenne can be found at

`www.dependent-types.org`

All you need is GHC to compile and run programs.