De Bruijn Monads

A high-level perspective on de Bruijn encoding

Marco Maggesi (joint work with André Hirschowitz)

Foundations for the practical formalization of mathematics (FPFM)
April 26–27, 2017, Nantes, France

Plan of this talk

- *Classical* dB encoding
- *Functional* dB encoding
- Theory: modules over dB-monads

Computing vs Reasoning

Good for *reasoning*

```
fact 0       := 1
fact (suc n) := n * fact n
```

Good for *computing*

```
fact n    := facti 1 n
facti a 0 := a
facti a n := facti (n*a) (n-1)
```

Motivating example

De Bruijn encoding of Lambda Calculus

De Bruijn encoding: example

Nominal encoding:

$$\lambda x.(\lambda y.yx)x$$

De Bruijn encoding:

$$\lambda.(\lambda.01)0$$

De Bruijn encoding

Example: Datatype for λ-calculus in OCaml

```
type term = Ref of int
          | Abs of term
          | App of term * term;;
```

Classical approach (good for computing)

```
value rec lifti n t k  = match t with
  [ Ref i   -> if i<k then Ref(i)
                      else Ref(n+i)
  | Abs t   -> Abs (lifti n t (k+1))
  | App t u -> App (lifti n t k) (lifti n u k)
  ]
and lift n t = lifti n t 0;
```

[G. Huet, CCT, Section 1.4.2, p. 15]

Classical approach (good for computing)

```
value rec substi w t n = match t with
  [ Ref k   -> if k=n then lift n w
               else if k<n then Ref k
                    else Ref (pred k)
  | Abs t   -> Abs (substi w t (n+1))
  | App t u -> App (substi w t n) (substi w u n)
  ]
and subst u t = substi u t 0;
```

[G. Huet, CCT, Section 1.4.2, p. 15]

Problem: associativity of substitution

```
subst w (subst u t) =
    subst (subst w u) (substi w t 1)
```

Classical approach (not so good for reasoning)

$$lifti\ k\ (lifti\ j\ t\ i)\ (j+i) = lifti\ (j+k)\ t\ i$$

$$i \leq n \Rightarrow lifti\ k\ (lifti\ j\ t\ i)\ (j+n) = lifti\ j\ (lifti\ k\ t\ n)\ i$$

$$i \leq k \leq (i+n) \Rightarrow lifti\ j\ (lifti\ n\ t\ i)\ k = lifti\ (j+n)\ t\ i$$

$$lifti\ k\ (substi\ u\ t\ j)\ (j+i) = substi\ (lifti\ k\ u\ i)\ (lifti\ k\ t\ (j+i+1)\ j)$$

$$i \leq n \Rightarrow substi\ u\ (lifti\ j\ t\ i)\ (j+n) = lifti\ j\ (substi\ u\ t\ n)\ i$$

$$i \leq k \leq (i+n) \Rightarrow substi\ u\ (lifti\ (n+1)\ t\ i)\ k = lifti\ n\ t\ i$$

$$substi\ w\ (substi\ u\ t\ i)\ (i+j) = substi\ (substi\ w\ u\ j)\ (substi\ w\ t\ (i+j+1))\ i.$$

[Huet, CCT, Section 1.4.3, p.15]

Associativity of substitution

What you want:

```
subst w (subst u t) = subst (subst w u) (substi w t 1)

      substi w (substi u t 0) 0 =
          substi (substi w u 0) (substi w t 1) 0
```

What you have to prove by induction:

```
substi w (substi u t i) (i + j) =
    substi (substi w u j) (substi w t (i + j + 1)) i
```

Even worst: fusion law subs-lift

What you want:

```
subst u (lift (n + 1) t) = lift n t
```

What you have to prove by induction:

```
i ≤ k ≤ i + n
⇒ substi u (lifti (n + 1) t i) k =
    lifti n t i
```

We seek for a more elegant solution

Key ideas

- Use *parallel substitution*
- *Functional* approach

“La supériorité de l'ordre supérieur”

```
subst : (nat -> term) -> (term -> term)
deriv : (nat -> term) -> (nat  -> term)
map   : (nat ->  nat) -> (term -> term)

subst f (Ref i)      := f i
subst f (App (t,u)) := App (subst f t,subst f u)
subst f (Abs t)      := Abs (subst (deriv f) t)

deriv f 0            := Ref 0
deriv f (Suc i)      := map suc (f i)

map   f x            := subst (Ref o f) x
```

How to recover the linear substitution

```
push u f 0        := u
push u f (suc i) := f i

subst1 u v        := subst (push u ref) v
```

Advantages

- No auxiliary functions/parameters (i.e., no k).
- High-level view on each line of code.
- Nice fusion laws

Monadic fusion laws

`Monadic' fusion laws

```
  map f (map g t)   = map (f o g) t
  map f (subst g t) = subst (map f o g) t
subst f (map g t)   = subst (f o g) t
subst f (subst g t) = subst (subst f o g) t
```

`Classical' fusion laws [Huet]

$$lifti\ k\ (lifti\ j\ t\ i)\ (j+i) = lifti\ (j+k)\ t\ i$$

$$i \leq n \Rightarrow lifti\ k\ (lifti\ j\ t\ i)\ (j+n) = lifti\ j\ (lifti\ k\ t\ n)\ i$$

$$i \leq k \leq (i+n) \Rightarrow lifti\ j\ (lifti\ n\ t\ i)\ k = lifti\ (j+n)\ t\ i$$

$$lifti\ k\ (substi\ u\ t\ j)\ (j+i) = substi\ (lifti\ k\ u\ i)\ (lifti\ k\ t\ (j+i+1)\ j)$$

$$i \leq n \Rightarrow substi\ u\ (lifti\ j\ t\ i)\ (j+n) = lifti\ j\ (substi\ u\ t\ n)\ i$$

$$i \leq k \leq (i+n) \Rightarrow substi\ u\ (lifti\ (n+1)\ t\ i)\ k = lifti\ n\ t\ i$$

$$substi\ w\ (substi\ u\ t\ i)\ (i+j) = substi\ (substi\ w\ u\ j)\ (substi\ w\ t\ (i+j+1))\ i.$$

De Bruijn monads

From monads to dB-monads

- De Bruijn functor: $\mathsf{dB} : \langle \mathbb{N} \rangle \longrightarrow \mathbf{Set}$
- De Bruijn monads **:=** monads relative to dB
- Functor

$$\text{monads}/\mathbf{Set} \longrightarrow \mathsf{dB}\text{-monads}$$

$$R \longmapsto R(\mathbb{N})$$

Axiomatic presentation of dB-monads

Structure

Carrier	T : type
Substitution	subst : $(\mathbb{N} \longrightarrow \mathsf{T}) \longrightarrow (\mathsf{T} \longrightarrow \mathsf{T})$
Reference	ref : $\mathbb{N} \longrightarrow \mathsf{T}$

Axioms

Associativity	subst f (subst g x) = subst (subst f o g) x
Right unit	subst f (ref i) = f i
Left unit	subst ref x = x

Accessory definitions

```
map f t              := subst (ref o f) t

lift n t             := map (i ⟼ i+n) t

push u f 0           := u
push u f (suc i)     := f i

subst1 u t           := subst (push u ref) t
```

(dB-)Modules

dB-modules

Structure

Base	`T` dB-monad
Carrier	`M` : `type`
Action	`msubst` : $(\mathbb{N} \longrightarrow$ `T`$) \longrightarrow ($`M` $\longrightarrow$ `M`$)$

Axioms

Associativity	`msubst f (msubst g x) = msubst (subst f o g) x`
Left unit	`msubst ref x = x`

dB-linear morphisms

`M1`, `M2` modules over the same dB-monad `T`.

A linear *morphism* is

```
phi : M1 -> M2
```

such that

```
phi (msubst1 f x) = msubst2 f (phi x)
```

Basic examples of dB-modules

- The tautological module
- Initial module $\mathbb{N}$ and final module $\star$
- Products

Derivation

- `M` module over a dB-monad `T`
- `M'` is the derived module with

```
M'                  := M
msubst' f x         := msubst (deriv f) x
deriv f 0           := ref 0
deriv f (suc i)     := bump (f i)
```

- Caveat: $\mathbb{N} \simeq \mathbb{N} + \star$

λ-calculus revised

```
ref :      nat       -> term    --  unit
app : term * term -> term    -- linear
abs :     term'      -> term    -- linear
```

High level point of view

```
subst f (Ref i)    := f i                              right unit
subst f (App(t,u)) := App(subst f t,subst f u)         linearity App
subst f (Abs t)    := Abs(subst (deriv f) t)           linearity Abs

deriv f 0          := Ref 0                            derivation
deriv f (Suc i)    := map suc (f i)

map   f x          := subst (Ref o f) x                functoriality
```

Initial syntax and semantics

Syntax and semantics with dB-monads

- Just use the functor
  $$\text{monads/Set} \longrightarrow \text{dB-monads}$$
- Can easily translate:
  - Signatures (high-order, algebraic)
  - Representations
  - Initiality
  - Equations — e.g., λ-calculus / αβη

Future work

- Expand our computer formalization (generalize to algebraic signatures).
- Add types [Ahrens].
- More general signatures (strengthened signatures: [Matthes-Uustalu].

Thank you!