Builtin Types viewed as Inductive Families

Guillaume Allais

University of Strathclyde
Glasgow, UK

ESOP'23
24^{th} April 2023

Table of Contents

Safe Lookup

```
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a

data Fin : Nat -> Type where
  Z : Fin (S n)
  S : Fin n -> Fin (S n)

lookup : Vect n a -> Fin n -> a
lookup (x :: _) Z = x
lookup (_ :: xs) (S k) = lookup xs k
```

Compiling Safe Lookup

Brady, McBride, and McKinna [BMM03]

```
partial
lookup : (n : Nat) -> List a -> Nat -> a
lookup (S n) (x :: _) Z = x
lookup (S n) (_ :: xs) (S k) = lookup n xs k
```

Tejiščák [Tej20]

```
partial
lookup : List a -> Nat -> a
lookup (x :: _) Z = x
lookup (_ :: xs) (S k) = lookup xs k
```

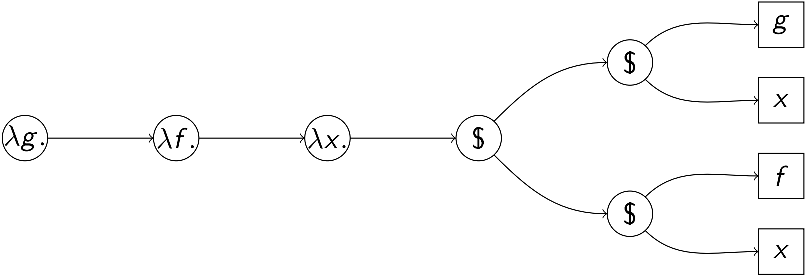
No Magic Solution

Quoting from the Coq reference manual:

> *Translating an inductive type to an arbitrary ML type does* **not** *magically improve the asymptotic complexity of functions, even if the ML type is an efficient representation. For instance, when extracting* `nat` *to OCaml* `int`*, the function* `Nat.mul` *stays quadratic.*

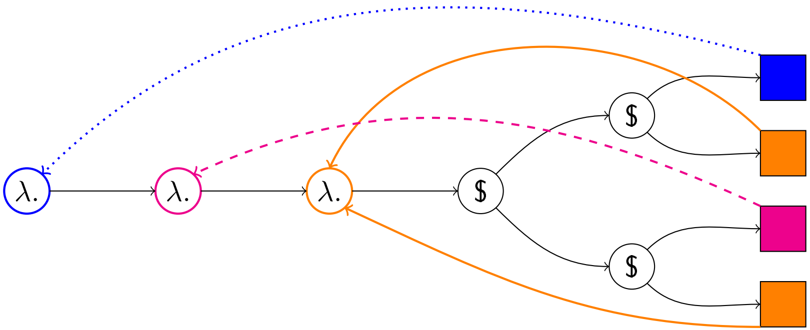
Table of Contents

Named: $\lambda g.\lambda f.\lambda x.g\,x\,(f\,x)$



Hard: α-equivalence

De Bruijn: $\lambda g.\lambda f.\lambda x.g\,x\,(f\,x)$



Hard: thickening

Co-De Bruijn [McB18]: $\lambda g.\lambda f.\lambda x.g\, x\, (f\, x)$

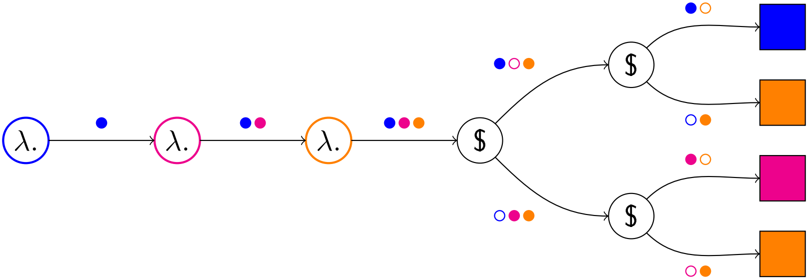


Table of Contents

Safe: Inductive Family

```
data Thinning : (sx, sy : SnocList a) -> Type where
  Done : Thinning [<] [<]
  Keep : Thinning sx sy -> (0 x : a) ->
         Thinning (sx :< x) (sy :< x)
  Drop : Thinning sx sy -> (0 x : a) ->
         Thinning sx (sy :< x)
```

Compiled to:

```
data Thinning = Done | Keep Thinning | Drop Thinning
```

i.e. essentially `List Bool`

Unsafe: Tuple

```
record Thinning where
  bitPattern : Integer
  bigEnd : Int
```

$$\text{bitPattern} = \cdots 0000\underbrace{1001\cdots 11}_{\text{bigEnd bits}}$$

Table of Contents

Thinnings

```
record Th {a : Type} (sx, sy : SnocList a) where
  constructor MkTh
  bigEnd : Nat
  encoding : Integer
  0 invariant : Invariant bigEnd encoding sx sy
```

Thinnings

```
record Th {a : Type} (sx, sy : SnocList a) where
  constructor MkTh
  bigEnd : Nat
  encoding : Integer
  0 invariant : Invariant bigEnd encoding sx sy

data Invariant : (i : Nat) -> (bs : Integer) ->
                 (sx, sy : SnocList a) -> Type where
  Done : Invariant Z 0 [<] [<]
  Keep : Invariant i (bs `shiftR` 1) sx sy -> (0 x : a) ->
         {auto 0 b  : So (testBit bs Z)} ->
         Invariant (S i) bs (sx :< x) (sy :< x)
  Drop : Invariant i (bs `shiftR` 1) sx sy -> (0 x : a) ->
         {auto 0 nb : So (not (testBit bs Z))} ->
         Invariant (S i) bs sx (sy :< x)
```

Smart constructors

```
done : Th [<] [<]
done = MkTh { bigEnd = 0, encoding = 0, invariant = Done }
```

Smart constructors

```
keep : Th sx sy -> (0 x : a) -> Th (sx :< x) (sy :< x)
keep th x = MkTh
 { bigEnd = S (th .bigEnd)
 , encoding = cons True (th .encoding)
 , invariant =
    let 0 b = eqToSo $ testBit0Cons True (th .encoding) in
    let 0 eq = consShiftR True (th .encoding) in
    Keep (rewrite eq in th.invariant) x
 }
```

Smart constructors

```
drop : Th sx sy -> (0 x : a) -> Th sx (sy :< x)
drop th x = MkTh
 { bigEnd = S (th .bigEnd)
 , encoding = cons False (th .encoding)
 , invariant =
   let 0 prf = testBit0Cons False (th .encoding) in
   let 0 nb = eqToSo $ cong not prf in
   let 0 eq = consShiftR False (th .encoding) in
   Drop (rewrite eq in th .invariant) x
 }
```

View [Wad87]: “un-applying” the smart constructors

```
data View : Th sx sy -> Type where
  Done : View Smart.done
  Keep : (th : Th sx sy) -> (0 x : a) -> View (keep th x)
  Drop : (th : Th sx sy) -> (0 x : a) -> View (drop th x)

view : (th : Th sx sy) -> View th
```

Using the view

```
kept : Th sx sy -> (n : Nat ** length sx === n)
kept th = case view th of
  Done      => (0 ** Refl)
  Keep th x => let (n ** eq) = kept th in
               (S n ** cong S eq)
  Drop th x => kept th
```

Table of Contents

Results & Limitations

Results:

- Functional, proven correct, library
- Generates good code
- Choose your own abstraction level

Limitations:

- Smart constructors & view are only *provably* inverses
- `Invariant` is proof irrelevant but it may not always be possible
- Inverting proofs with a `0` quantity is currently painful

Bibliography

Edwin C. Brady, Conor McBride, and James McKinna.
Inductive families need not store their indices.
In *TYPES*. Springer, 2003.

Conor McBride.
Everybody's got to be somewhere.
In *MSFP*. EPTCS, 2018.

Matúš Tejiščák.
A dependently typed calculus with pattern matching and erasure inference.
In *ICFP*. ACM, 2020.

Philip Wadler.
Views: A way for pattern matching to cohabit with data abstraction.
In *POPL*. ACM, 1987.