Efficient Elaboration with Controlled Definition Unfolding

András Kovács

University of Gothenburg

20 January 2024, WITS, London

Overview

1. We want to fill holes during elaboration with nice terms.
2. We want to present nice terms to users in interaction.

Unfolding all definitions is usually not nice.

Overview

1. We want to fill holes during elaboration with nice terms.
2. We want to present nice terms to users in interaction.

Unfolding all definitions is usually not nice.

Not unfolding anything can be also bad...

...but we want to preserve unfoldings until we can make a good decision.

Unfolding can be explosive

Classic Hindley-Milner example:

```
let x1 = (True, True) in
let x2 = (x1, x1) in
let x3 = (x2, x2) in
...
xN
```

Unfolding can be explosive

Classic Hindley-Milner example:

```
let x1 = (True, True) in
let x2 = (x1, x1) in
let x3 = (x2, x2) in
...
xN
```

“Who writes code like this?”

Unfolding can be explosive

Classic Hindley-Milner example:

```
let x1 = (True, True) in
let x2 = (x1, x1) in
let x3 = (x2, x2) in
...
xN
```

“Who writes code like this?” (Agda users, incl. me)

Unfolding can be explosive

Classic Hindley-Milner example:

```
let x1 = (True, True) in
let x2 = (x1, x1) in
let x3 = (x2, x2) in
...
xN
```

"Who writes code like this?" (Agda users, incl. me)

Quadratic elaboration is surprisingly common with fancy types!

```
cons : (n : ℕ) → Bool → Vec Bool n → Vec Bool (suc n)
myvec = cons _ True (cons _ True (cons _ True nil))
```

Also in GHC: https://well-typed.com/blog/2021/12/type-level-sharing-now/

Unfolding can be explosive

Classic Hindley-Milner example:

```
let x1 = (True, True) in
let x2 = (x1, x1) in
let x3 = (x2, x2) in
...
xN
```

"Who writes code like this?" (Agda users, incl. me)

Quadratic elaboration is surprisingly common with fancy types!

```
cons : (n : ℕ) → Bool → Vec Bool n → Vec Bool (suc n)
myvec = cons (suc (suc zero)) True (cons (suc zero) True (cons zero Tru
```

Also in GHC: https://well-typed.com/blog/2021/12/type-level-sharing-now/

Hash consing is insufficient

Hash consing doesn't know about beta-expansions.

```
data Nat = Zero | Suc Nat

oneMillion : Nat
oneMillion = ...
```

The `oneMillion` definition may be tiny, but the end result is big and incompressible by hash consing!

Hash consing also doesn't help with unfolding in user interaction.

But: hash-consing/CSE can be crucial to clean up very bad input.

Basic setup

- **Core terms** are viewed as immutable program code.
- Terms get **evaluated** into **semantic values**.
- Expensive computation (reduction, conversion, unification) is implemented only on values.
- There is a **quotation** operation which turns values back to terms.

Interleaving of normalization-by-evaluation and elaboration.[1]

[1] Originally by Thierry Coquand, 1996, "An algorithm for checking dependent types"

Minimal NbE with top-level unfolding control (1)

```haskell
{-# language Strict #-}

data Tm = TopVar TopName
        | LocalVar LocName
        | App Tm Tm
        | Lam LocName Tm

type Spine = [Val]
type Env   = [(LocName, Val)]
data Val   = VNe LocName Spine | VLam LocName (Val -> Val)
           | VUnfold (TopName, Spine) ~Val
```

Minimal NbE with top-level unfolding control (2)

```
eval :: Env -> Tm -> Val
eval env t = case t of
  TopVar x   -> VUnfold (x, []) (lookupTop x)
  LocalVar x -> lookup x env
  App t u    -> vApp (eval env t) (eval env u)
  Lam x t    -> VLam x (λ v -> eval ((x, v):e) t)

vApp :: Val -> Val -> Val
vApp v v' = case v of
  VLam _ f          -> f v'
  VUnfold (x, sp) v -> VUnfold (x, (v':sp)) (vApp v v')
  VNe x sp          -> VNe x (v':sp)
```

Minimal NbE with top-level unfolding control (3)

```
quoteSp :: [LocalName] -> Bool -> Tm -> Spine -> Tm
quoteSp ns unfold t sp =
  foldl' (λ t v -> App t (quote ns unfold v)) t sp

quote :: [LocalName] -> Bool -> Val -> Tm
quote ns unfold v = case v of
  VNe x sp          -> quoteSp ns unfold (LocalVar x) sp
  VLam x f          -> let x' = fresh ns x in
                       Lam x' (quote (x':ns) (f (VNe x' [])))
  VUnfold (x, sp) v -> if unfold then quote ns unfold v
                                 else quoteSp ns unfold (TopVar x)
```

Glued values

One value in the pair is *minimally* unfolded, the other is *maximally* unfolded.

```
type GVal = (Val, Val)
```

During elaboration/unification, we often force values to whnf, but don't want to throw away the original value!

```
whnf :: Val -> Val
whnf (VUnfold _ v) = whnf v
whnf v             = v

type VTy = Val
type GTy = GVal

check :: Context -> RawTm -> GTy -> M Tm
infer :: Context -> RawTm -> M (Tm, GTy)
unify :: Context -> GVal -> GVal -> M ()
```

When unfolding is needed

Elaboration input:

```
map2 : ∀ {A B C : Type} → (B → C) → (A → B) → List A → List B
map2 f g as = map f (map g as)
```

When unfolding is needed

Elaboration input:

```
map2 : ∀ {A B C : Type} → (B → C) → (A → B) → List A → List B
map2 f g as = map f (map g as)
```

Un-optimized output:

```
m1 = λ A B C f g as. B
m2 = λ A B C f g as. C
m3 = λ A B C f g as. A
m4 = λ A B C f g as. B

map2 : ∀ {A B C : Type} → (B → C) → (A → B) → List A → List B
map2 = λ {A}{B}{C} f g as.
  map {m1 A B C f g as}{m2 A B C f g as} f
    (map {m3 A B C f g as}{m4 A B C f g as} g as)
```

When unfolding is needed

Desired output:

```
map2 : ∀ {A B C : Type} → (B → C) → (A → B) → List A → List B
map2 = λ {A}{B}{C} f g as. map {B}{C} f (map {A}{B} g as)
```

Early code optimization

User-written code should be **preserved**.

Code from elaboration/tactics should be **optimized** soon after it's generated.

- Already in elaboration, we don't want to compute with poor-quality code.
- Simple and fast optimization (basic inlining, let-motion) makes a big difference.

Human-readable elaboration output is important for debugging/optimization!

- A major issue in Agda right now.

Local unfolding control

```
topdef =
  let bigDef1 = ...
      bigDef2 = ...
      bigDef3 = ...
      ...
  in ?
```

We split the local Env to two parts:

1. Visible let-s in elaboration scope.
2. Non-visible bindings arising from evaluation.

Local unfolding control

```
data Tm  = ... | Let LocName Val Val
data Val = ... | VLet LocName Val (Val -> Val) ~Val
               | VUnfoldLoc (LocName, Spine) ~Val

eval :: Env -> Env -> Tm -> Val
eval env env' t = case t of
   LocalVar x -> if x is in env' then lookup x env'
                                 else VUnfoldLet (x, []) (lookup x env)
   Let x t u  -> let v = eval env env' t in
                 VLet x v (λ v -> eval env ((x, v):env') u) v
   ...

vApp v v' = case v of
   VLet x t f v          -> VLet x t ((`vApp` v') . f) (vApp v v')
   VUnfoldLoc (x, sp) v -> VUnfoldLoc (x, (v':sp)) (vApp v v')
   ...
```

Local unfolding control

```
quote ns unfold v = case v of

  VUnfoldLoc (x, sp) v
    | unfold    -> quote ns unfold v
    | otherwise -> quoteSp ns unfold (LocVar x)

  VLet x v f v'
    | unfold    -> quote ns unfold v'
    | otherwise -> let x' = fresh ns x in
                   Let x' (quote ns unfold v)
                          (quote (x':ns) unfold (f (VNe x' [])))
  ...
```

Metavars and local let-s

How can meta solutions refer to local let-s? Two ways:

1. Metas are top-level only, but abstract over let-s too.
   - + Relatively simple.
   - – More overhead during elaboration.
   - + Can be still cleaned up afterwards.
2. Use metas in arbitrary local scope.
   - + More efficient during elaboration.
   - + The only sensible way to do let-generalization.
   - – But we probably just don't want to have let-generalization.
   - – A lot more complex.
   - – We still need to optimize the output.

Non-explosive record construction

What's wrong with this in Agda and elsewhere?

```
foo = record {
    field1 = bigDef1;
    field2 = bigDef2;
    ...
    fieldN = bigDefN }
```

There's no way to share field definitions in elaboration!

Ugly fix: insert a let-binding for each field.

Better fix: *each field definition binds the field name like a local let, in the rest of the record construction*.

Related Implementations

- Sixten: `https://github.com/ollef/sixten`
- Sixty: `https://github.com/ollef/sixty`
- smalltt: `https://github.com/AndrasKovacs/smalltt`
- sett: `https://github.com/AndrasKovacs/sett`
- cctt: `https://github.com/AndrasKovacs/cctt`
- Idris 2: `https://github.com/idris-lang/Idris2`