Parallel hereditary substitutions

Andreas Abel

30 October 2019

1 Introduction

During the TYPES 2018 conference in Braga, Conor McBride posed me with the riddle how to implement parallel hereditary substitutions. At the time of TYPES 2019 in Oslo I had not solved the riddle yet. He gave me the additional hint that *active and passive parts need to be separated* (I don't remember the actual wording).

In October 2019, after many tries to implement his hint, and after leaving a *field of corpses* (McBride, 20??), I realized I had already had the main idea needed here in my implementation of single hereditary substitutions, in the type of `Res`. This type distinguishes (the only) active substitution entry `tm` contained in the `sg` singleton substitution from a passive entry `var` introduced by `lift`.

Turning my realization into a solution was not hard from there, except that I used sized types to implement the common upper bound on the order of the types of all active substitution entries, and it took some patience to get all the sizings correct.

The following is a detailed sketch of the solution in prose, but see also the Agda implementation.

2 Syntax

Types are simple types over a set of uninterpreted base types denoted by `o`.

```
a,b,c ::= o | a ⇒ b
```

The order of a type.

```
ord(o)     = 0
ord(a ⇒ b) = max (ord(a)+1, ord(b))
```

Terms are in -normal form:

- Variables (de Bruijn indices)

```
x ∈ ℕ
```

- Normal forms

```
t, u ::= λ t | x s
```

- Spines

```
s ::= ε | u, s
```

2.1 Typing

Variable typing `Γ ∋ x : a`

```
Γ.a ∋ 0 : a

Γ ∋ n : a
-------------
Γ.b ∋ n+1 : a
```

Normal form typing `Γ ⊢ t : a`

```
Γ.a ⊢ t : b
---------------
Γ ⊢ λ t : a ⇒ b

Γ ∋ x : a    Γ | a ⊢ s : b
--------------------------
Γ ⊢ x s : b
```

Spine typing `Γ | a ⊢ s : b`

```
Γ | a ⊢ ε : a

Γ ⊢ u : a    Γ | b ⊢ s : c
--------------------------
Γ | (a ⇒ b) ⊢ (u,s) : c
```

Weakening `t + 1`, to be defined as usual via generalization to context extensions `Γ ⊆ Δ`.

```
Γ ⊢ t : a
---------------
Γ.b ⊢ t + 1 : a
```

3 Substitutions

Substitutions

```
σ ::= id | σ.u | lift σ
```

Looking up in a substitution `σ(x) = t`

```
id(x)          = x
(σ.u)   (0)    = u
(σ.u)   (x+1)  = σ(x)
(lift σ)(0)    = 0
(lift σ)(x+1)  = σ(x) + 1
```

3.1 Bounded substitution typing

Bounded typing of non-variables $\Gamma \vdash^n t : a$:

```
Γ ∋ x : a
---------
Γ ⊢ⁿ x : a

Γ ⊢ t : a
---------- ord(a) ≤ n
Γ ⊢ⁿ t : a
```

Bounded substitution typing $\Gamma \vdash^n \sigma : \Delta$:

```
Γ ⊢ⁿ id : Γ

Γ ⊢ⁿ σ : Δ    Γ ⊢ⁿ u : a
-------------------------
Γ ⊢ⁿ σ.u : Δ.a

Γ ⊢ⁿ σ : Δ
-------------------
Γ.a ⊢ⁿ lift σ : Δ.a
```

Lemma (substitution lookup). If $\Gamma \vdash^n \sigma : \Delta$ and $\Delta \ni x : a$ then $\Gamma \vdash^n \sigma(x) : a$.

Proof. In the interesting case `(σ.u)(0) = u`, we have by substitution typing $\Gamma \vdash^n u : a$.

4 Hereditary parallel substitution

We define four mutually recursive functions by a lexicographic measure `(n,m,l)` $\in \mathbb{N}^3$ where

- `n` is a bound on the order of a type
- `m` is the function index:

0. active application: `0`
1. hereditary substitution into normal forms or spines: `1`
2. accumulating application: `2`

- `l` is the height or a term or spine (if viewed as a mixed tree of terms and spines)

The following definition is well-founded, i.e., the functions are terminating by virtue of the measure. We justify each recursive call by checking that the measure goes strictly down.

- Substitution into NFs: `t ∘ⁿ σ` measure `(n,1,t)`

```
λt  ∘ⁿ σ = λ (t ∘ⁿ lift σ)      ok: (n,1,t) < (n,1,λt)
x s ∘ⁿ σ = σ(x) •ⁿ (s ∘ⁿ σ)     ok: (n,1,s) < (n,1,xs)
                                and (n,0,_) < (n,1,_)
```

- Substitution into spines: `s ∘ⁿ σ` measure `(n,1,s)`

```
ε     ∘ⁿ σ = ε
(u,s) ∘ⁿ σ = (u ∘ⁿ σ, s ∘ⁿ σ)   ok: (n,1,u/s) < (n,1,(u,s))
```

- Active application: `t •ⁿ s` measure `(n,0,t)`

```
t   •ⁿ   ε     = t
x s •ⁿ   s'    = x (s,s')
λt  •ⁿ⁺¹ (u,s) = t [id.u]ⁿ s    ok: (n,_,_) < (n+1,_,_)
```

- Accumulating application: `t [σ]ⁿ s` measure `(n,2,t)`

```
t      [σ]ⁿ ε     = t ∘ⁿ σ                    ok: (n,1,t) < (n,2,t)
(x s)  [σ]ⁿ s'    = σ(x) •ⁿ (s ∘ⁿ σ , s')     ok: (n,0,_) < (n,2,xs)
λt     [σ]ⁿ (u,s) = t [σ.u]ⁿ s                ok: (n,2,t) < (n,2,λt)
```

4.1 Typing and totality

Theorem and proof. The four recursive functions are total if restricted to well-typed terms according to the following scheme:

- Substitution into NFs

```
Δ ⊢ t : a    Γ ⊢ⁿ σ : Δ
------------------------
Γ ⊢ t ∘ⁿ σ : a
```

  - Case (ok since `lift` preserves `n`)

```
λt  ∘ⁿ σ = λ (t ∘ⁿ lift σ)
```

  - Case

```
x s ∘ⁿ σ = σ(x) •ⁿ (s ∘ⁿ σ)
```

- Substitution into spines

  ```
  Δ | a ⊢ s : c    Γ ⊢ⁿ σ : Δ
  ------------------------
  Γ | a ⊢ s ∘ⁿ σ : c
  ```

  - Case

    ```
    ε ∘ⁿ σ = ε
    ```

  - Case

    ```
    (u,s) ∘ⁿ σ = (u ∘ⁿ σ, s ∘ⁿ σ)
    ```

- Active application `t •ⁿ s`

  ```
  Γ ⊢ⁿ t : a    Γ | a ⊢ s : c
  ---------------------------
  Γ ⊢ t •ⁿ s : c
  ```

  - Case

    ```
    t   •ⁿ   ε     = t
    ```

  - Case

    ```
    x s •ⁿ   s'    = x (s,s')
    ```

  - Case

    ```
    λt  •ⁿ⁺¹ (u,s) = t [id.u]ⁿ s
    ```

    The order of the type of an abstraction is at least `1`, thus, always of the form `n+1`. We get the following inversion on the bounded typing of `λt`:

    ```
    Γ.a ⊢ⁿ⁺¹ t : b
    ----------------
    Γ ⊢ⁿ⁺¹ λt : a ⇒ b
    ```

    Since `ord(a) ≤ n`, we have

    ```
    Γ ⊢ⁿ u : a
    ```

    which justifies the substitution typing

    ```
    Γ ⊢ⁿ id.u : Γ.a
    ```

    Thus, the accumulating application `t [id.u]ⁿ s` is welltyped with `Δ = Γ.a`.

- Accumulating application `t [σ]ⁿ s`

  ```
  Δ ⊢ⁿ⁺¹ t : b    Γ ⊢ⁿ σ : Δ    Γ | b ⊢ s : c
  ------------------------------------------
  Γ ⊢ t [σ]ⁿ s : b
  ```

- Case

  ```
  t      [σ]ⁿ ε     = t ∘ⁿ σ
  ```

- Case

  ```
  (x s)  [σ]ⁿ s'    = σ(x) •ⁿ (s ∘ⁿ σ , s')
  ```

- Case

  ```
  λt     [σ]ⁿ (u,s) = t [σ.u]ⁿ s
  ```

  The bounded typing of `λt` is:

  ```
  Δ.b ⊢ⁿ⁺¹ t : b'
  -------------------
  Δ ⊢ⁿ⁺¹ λt : b ⇒ b'
  ```

  Spine typing gives us

  ```
  Γ ⊢ u : b    Γ | b' ⊢ s : c
  ---------------------------
  Γ | (b ⇒ b') ⊢ (u,s) : c
  ```

  Since `ord(b) ≤ n`, we get

  ```
  Γ ⊢ⁿ u : b
  ```

  which justifies the substitution typing

  ```
  Γ ⊢ⁿ σ.u : Δ.b
  ```

  and, thus, the typing of the recursive call. □

5 Conclusion

Parallel hereditary solutions allow for more efficient normalization as several β-redexes can be gathered into a single substitution using the equation

```
λt [σ]ⁿ (u,s) = t [σ.u]ⁿ s
```

which corresponds to the usual closure-based evaluation strategy of lambda calculus.