Simply Typed Lambda Calculus

Mathias Vorreiter Pedersen

November 13, 2015

1 Recalling the untyped lambda calculus

1.1 Syntax

```
t ::= x
    | λ x . t
    | t t
```

1.2 Evaluation

$$\frac{}{x \Downarrow x} \qquad \frac{t \Downarrow t'}{\lambda x.t \Downarrow \lambda x.t'} \qquad \frac{t_1 \Downarrow t_1' \quad t_2 \Downarrow t_2'}{t_1 t_2 \Downarrow t_1' t_2'} \qquad \frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow t_2' \quad t[t_2'/x] \Downarrow t'}{t_1 t_2 \Downarrow t'}$$

1.3 Examples

We define natural numbers with the following data-type

```
Nat ::= 0
      | S Nat
```

and we can write this in untyped lambda calculus as

```
zero ≡ λ s z . z
succ ≡ λ n s z . s (n s z)
```

and define operations on this data-type. For instance

```
plus ≡ λ n m . n succ m
```

Let's try a simple example

```
plus 2 1 = 2 succ 1
         = (λ s z . s (s z)) succ (λ s z . s z)
         = succ (succ (λ s z . s z))
         = succ (λ s z . (s (s z)))
         = λ s z . s (s (s z))
         = 3
```

Alright. Now we have natural numbers. Let's have some lists as well! We model the following data-type (since we are still untyped we view * as the type of every term).

```
List ::= Nil
       | Cons * List
```

using the following lambda terms corresponding to the constructors.

```
nil = λ oncons onnil . onnil
cons = λ hd tl oncons onnil .
           oncons hd (tl oncons onnil)
```

For instance the list consisting of the numbers 2 and 3 will be

```
cons 2 (cons 3 nil)
  = λ oncons onnil .
       oncons 2 ((cons 3 nil) oncons onnil)
  = λ oncons onnil .
       oncons 2 (oncons 3 (nil oncons onnil))
  = λ oncons onnil .
       oncons 2 (oncons 3 onnil)
```

Notice that we can now write

```
cons 2 (cons 3 nil) plus 0
  = (λ oncons onnil .
        oncons 2 (oncons 3 onnil)) plus 0
  = plus 2 (plus 3 0)
  = 5
```

With this in mind, it now totally makes sense to define the summation of the elements in a list as

```
sum_list ≡ λ xs . xs plus 0
```

But there's nothing stopping us from doing

```
sum_list 3 = 3 plus 0
           = (λ s z . s (s (s z))) plus 0
           = plus (plus (plus 0))
           = plus (plus (λ m . 0 succ m))
           = plus (plus (λ m . m))
           = plus (λ m . (λ m . m) succ m)
           = plus (λ m . succ m)
           = λ m . (λ m . succ m) succ m
           = λ m . (succ succ) m
           = λ m . (λ s z . s (succ s z)) m
           = λ m . (λ s z . s (λ y . z (s z y))) m
           = λ m . λ z . m (λ y . z (m z y))
```

Such a term makes no sense within the interpretation we've made of the terms! Clearly we'd like to somehow prevent such silly mistakes.

2 Introducing types

```
t ::= x
    | λ x . t
    | λ x : τ . t
    | t t
    | t : τ
```

where

```
τ ::= A
    | τ → τ
```

2.1 Type well-formedness

$$\frac{}{\Gamma \vdash_{\mathsf{wf}} A} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \tau_1 \quad \Gamma \vdash_{\mathsf{wf}} \tau_2}{\Gamma \vdash_{\mathsf{wf}} \tau_1 \to \tau_2}$$

2.2 Type inference and type checking

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Uparrow \tau} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \tau \quad \Gamma \vdash t \Downarrow \tau}{\Gamma \vdash t : \tau \Uparrow \tau} \qquad \frac{\Gamma \vdash t_1 \Uparrow \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 \Downarrow \tau_1}{\Gamma \vdash t_1 t_2 \Uparrow \tau_2}$$

$$\frac{\Gamma \vdash_{\mathsf{wf}} \tau_1 \quad \Gamma, x : \tau_1 \vdash t \Uparrow \tau_2}{G \vdash \lambda x.\tau_1.t \Uparrow \tau_1 \to \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t \Downarrow \tau_2}{\Gamma \vdash \lambda x.\tau \Downarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t \Uparrow \tau}{\Gamma \vdash t \Downarrow \tau}$$

2.3 Examples

Define the identity function of values of type A as

```
id ≡ (λ x : A . x) : A → A
```

Note that the following does NOT type check

```
id id
```

And definining the well-known combinator

```
K ≡ (λ x : A . λ y : A . x) : A → A → A
```

Neither of the following type checks

```
id K
K id
```

But the following type checks

```
(λ x : A . K (id x) (K x (id x))) : A → A
```

2.4 Revisiting our example

Let's add types to the constructors of the natural numbers. Write

```
Nat ≡ (A → A) → A → A
```

```
zero = (λ s : (A → A) . λ z : A . z) : Nat
succ = (λ n : Nat .
          λ s : A → A . λ z : A .
            s (n s z)) : Nat → Nat
plus = (λ n : Nat .
          λ m : Nat .
            n succ m) : Nat → Nat → Nat
```

and similarly for lists we write

```
List ≡ (A → A → A) → A → A
```

And the typed versions of our list data-type constructors

```
nil = (λ oncons : A → A → A .
         λ onnil : A . onnil) : List
cons = (λ hd : A . λ tl : List .
          λ oncons : A → A → A . λ onnil : A .
            oncons hd (tl oncons onnil))
         : A → List → List
```

Note that now we cannot type

```
cons 2 (cons 3 nil)
```

since 2 has type Nat (that is, $(A \to A) \to A \to A$), and not A as expected. Thus we'd need a separate data-type (and data-type constructor terms) for terms representing lists of natural numbers. In other words: Our data-type is monomorphic. For polymorphic (more specifically: parametric polymorphic) lists we'd need a more sophisticated type system.

3 The cost of adding types

By adding simple types to our language we've severly limited its computational power. To see this, consider the famous Ω term

```
Ω ≡ (λ x . x x) (λ x . x x)
```

What is the type of this term? Let's assume we can type this program and let's write

$$\Omega \equiv t_1 \;\; t_2$$

Since Ω is well-typed it must follow that t_1 is a function type $\tau_1 \to \tau_2$, and that t_2 is a term of type τ_1. But note that the terms t_1 and t_2 are equivalent! So we must have $\tau_1 \to \tau_2 \cong \tau_1$. It is, however, a well-known fact that such an isomorphism cannot exist in the standard set theoretic notion.[1]

We've reached a contradiction and can conclude that Ω does not have a type in simply typed lambda calculus!

The Ω term has other remarkable properties. To see this, let's try to find its normal-form:

```
Ω = (λ x . x x) (λ x . x x)
  = (λ x . x x) (λ x . x x)
  = (λ x . x x) (λ x . x x)
  = (λ x . x x) (λ x . x x)
  = (λ x . x x) (λ x . x x)
  ...
```

Okay that was fairly unsuccessful. It turns out that Ω does not have a normal-form. In other words: It's a diverging computation! This is precisely why we couldn't construct a type for Ω: Terms in the simply typed lambda calculus are guaranteed to terminate. Does that mean we've solved the halting problem? Well no, it's provably undecidable for Turing complete models of computation. But that must imply that simply typed lambda calculus is not Turing complete.

Summing up: In the process of adding types we've removed Turing completeness from our language. What a bummer!

[1] Yes, category theory is lurking just below the surface.