Resolution of exercises

Juan García Garland

November 23, 2017

Some Remarks

Notation

We write $\lambda fg.t$ instead of $\lambda f.\lambda g.t$ as usual. we write $\lambda\_.t$ to construct a term that "eats an argument". So _ is a variable such that $\_ \notin FV(t)$

Exercise 5

A good idea to think how to construct the terms is to reason over the types that they should have. Even if the system is untyped, this strategy works. Type systems usually restrict which terms can be constructed, but they do not affect reduction. If we construct a term assuming a type system and it reduces as we want, it will work in the untyped calculus.
In this exersise whe should encode the universal property of coproduct. Σ_i are the constructors (usually *inl*, *inr* in functional programming literature), so they have type $A_i \to A_1 + A_2$.
Since we do not have a primitive way to construct a sum type (in the way we defined our calculus), we have to encode it in the same way we encode, for example Booleans with the type $X \to X \to X$ (given two terms of type X, in which one you choose you have a bool of information).
Using this reasoning, a good way to type the terms is:

$$\Sigma_i : A_i \to (A_1 \to B) \to (A_2 \to B) \to B$$

So

$$\Sigma_1 = \lambda tfg.ft$$

$$\Sigma_2 = \lambda tfg.gt$$

are good definitions.
C is the destructor of this case analysis, it should have type

$$C : A_1 + A_2 \to (A_1 \to B) \to (A_2 \to B) \to B$$

So

$$C = \lambda cfg.cfg$$

is a possible definition (recall: unfold the definition of $A_1 + A_2$ and this type checks).
Finally we can verify:

$$\underline{C}(\Sigma_1 t)f_1 f_2 \succ^3 (\Sigma_1 t)f_1 f_2 = (\underline{\lambda tfg.ft})tf_1 f_2 \succ^3 f_1 t$$

$$\underline{C}(\Sigma_2 t)f_1 f_2 \succ^3 (\Sigma_2 t)f_1 f_2 = (\underline{\lambda tfg.gt})tf_1 f_2 \succ^3 f_2 t$$

note: Actually $C = Id$ works fine and we need less reductions, I preferred this presentation because is inmediate from some program calculation from types, I think also that this technique will be useful to derive harder λ-programs.
note: This is probably not the unique way to code coproducts.
note: C is actually a (prefix) `case` construction, $(C\ u\ f\ g)$ actually models:

```
case u of
    inl t → f t
    inr t → g t
```

Exercise 7

A great advantage of Scott numerals is that the predecessor function can be encoded efficiently. On the other hand, functions like sum, that are computed in constant time in Church's implementation, are linear in this case.
Let $\bar{0} := \lambda fx.x$ and $S := \lambda nfx.fn$
It is not hard to show that
$\bar{n} = \lambda fx.f(\lambda fx.f(...(\lambda fx.f(\lambda fx.x))))$ (each successor application is a double abstraction, applying the first argument, where $\bar{0}$ is the second double abstraction using the second argument).

Zero test

$Zero := \lambda mtf.m(\lambda\_.f)t$ where
$Zero$ applied to a numeral m must be a boolean, so there are two extra arguments (recall: $True = \lambda tf.t$, $False = \lambda tf.f$).
Then apply m to a function that eats an argument and returns f, and to t. If m is $\bar{0}$ we use the second argument (t), else the result is apply f to the predecessor of m, eating it and returning f.

$$Zero\ \bar{0} = Zero\ (\lambda fx.x) = (\lambda mtf.m(\lambda\_.f)t)(\lambda fx.x) \succ \lambda tf.(\lambda fx.x)(\lambda\_.f)t \succ^2 \lambda tf.t$$

$$Zero\ \bar{0} = Zero\ (\lambda fx.fN) = (\lambda mtf.m(\lambda\_.f)t)(\lambda fx.fN) \succ \lambda tf.(\lambda fx.fN)(\lambda\_.f)t \succ^2 \lambda tf.(\lambda\_.f)N \succ \lambda tf.t$$

Note that we can use zero test as a case analysis. EXPLAIN

Predecessor

Erase the first two abstractions and the application of the first parameter. So, apply the numeral to the identity and somewhat else.
$Pred := \lambda m.m\ (id)\ y$
Thus:

$$Pred\ \bar{0} = (\lambda m.m\ (id)\ y)(\lambda fx.x) \succ (\lambda fx.x)(id)y \succ^2 y$$

So y is the value of $Pred\ \bar{0}$, we can redefine $Pred := \lambda m.m\ (id)\ \bar{0}$ if we want that $Pred\ \bar{0} \succ^* \bar{0}$, wich is a reasonable implementation.

$$Pred\ (SN) = (\lambda m.m\ (id)\ y)(\lambda fx.fN) \succ (\lambda fx.fN)(id)y \succ^2 (id)N \succ N$$

Recursor

It is programmed exactly in the same way than for Church numerals, since we do not touch the "low level implementation" (of course, all advantages of modularization are also true in this level).
R must take a function to apply in the recursive case, and value to return in the zero case, it must satisfy this equations:

$$Rfa\bar{0} \equiv_\beta a \tag{1}$$
$$Rfa(SN) \equiv_\beta f(SN)(RfaN) \tag{2}$$

The second equation can be rewtited as

$$RfaN \equiv_\beta fN(Rfa(PredN)) \tag{3}$$

To avoid pattern matching in the numeral argument. Then, combine both (1) and (3) as:

$$R\ f\ a\ N \equiv_\beta Zero\ N\ a(fN(R\ f\ a(Pred\ N))) \tag{4}$$

Finally we use the usual trick to write recursive functions. Given the Y fixpoint combinator, let $R := Y\ R'$ where

$$R' = \lambda R\ f\ a\ n.Zero\ n\ a\ (f\ n(R\ f\ a\ (pred\ n)))$$

Adition, Multiplication, Exponentiation

Since the $Pred$ function reduces in constant time, these functions can be implemented using the recursor without much overhead.
$Add := \lambda\ m\ n.R\ (\lambda\_.S)\ m\ n$
$Mul := \lambda\ m\ n.R\ (\lambda\_.Add\ m)\ \bar{0}\ n$
$Exp := \lambda\ m\ n.R\ (\lambda\_.Mul\ m)\ \bar{1}\ n$

Equality test and Comparison

First define
$Sub := \lambda\ m\ n.R\ (\lambda\_.Pred)\ m\ n$
The substraction function (with the implementation od $Pred$ such that $Pred\ \bar{0} = \bar{0}$).
Then:
$Leq = \lambda m\ n.Zero(Sub\ m\ n)$
$Eq = \lambda m\ n.And(Leq\ m\ n)(Leq\ n\ m)$
where $And = \lambda a\ b\ t\ f.a\ b(\lambda t\ f.f)$
(Can be implemented more efficiently, of course)

1 Exercise 10 - Church-Rosser Theorem

We must show that $\succ^*$ is confluent.
Define $D : \Lambda \to \Lambda$ as:

$D(x) = x$

$D(\lambda x.t) = \lambda x.D(t)$

$D((\lambda x.t)u) = D(t)[x := D(u)]$

$D(tu) = D(t)D(u)$ if t is not an abstraction

Lemma 1. *If $t \succ^* t'$ and $u \succ^* u'$ then $t[x := u] \succ^* t'[x := u']$.*

Proof. Well known, it follows by induction over t using the substitution lemma. □

We prove the following lemmas by induction over a term. We separate the application in cases depending if the first term is an abstraction or not, since the definition of D is different in each case.

Lemma 2. $t \succ^* D(t)$

Proof. By induction over t.

- If $t = x$ this is trivial since $D(t) = x$, so $t \succ^0 D(t)$.
- If $t = \lambda x.a$, by induction hypothesis $a \succ^* D(a)$, since a is smaller than t, so $\lambda x.a \succ^* \lambda x.D(a) = D(\lambda x.a)$
- If $t = (\lambda x.a)b$, then $(\lambda x.a)b \succ a[x := b] \succ^*_{hip} D(a)[x := D(b)] = D((\lambda x.a)b)$ using Lemma 1 and induction hypothesis.
- Finally if $t = ab$ where a is not an abstraction,

  $ab \quad \succ^*$ { by induction hypothesis, twice }
  $D(a)D(b) \quad =$ { by definition of D }
  $D(ab)$

□

Lemma 3. *If $t \succ t'$ then $t' \succ^* D(t)$*

Proof. By induction over t (considering the rules for beta reductions).

- If $t = x$ trivial (actually not possible, since t has no redex).
- If $t = \lambda x.a$, then necessarly $t' = \lambda x.a'$ where $a \succ a'$ (there is only one redex rule for abstraction, we reason by inversion). Then:

  $t' \quad =$
  $\lambda x.a' \quad \succ^*${ by induction hypothesis in a}
  $\lambda x.D(a) \quad =${ by definition of D}
  $D(\lambda x.a) = D(t)$

- If $t = (\lambda x.a)b$ it can reduce in one step to $(\lambda x.a')b$, $(\lambda x.a)b'$ or $a[x := b]$ (where $a \succ a'$, $b \succ b'$). Applying one more reduction to the first two cases (in the outhermost redex), it is enough to show that $a'[x := b]$, $a[x := b']$, $a[x := b]$ all reduces (in $*$ steps) to $D(t) = D(a)[x := D(b)]$. This is trivial applying Lemma 1, and Lemma 2 and induction hypothesis to the instances of t and u (induction to the primes, lemma to the untouched substerms). For instance $a[x := b'] \succ^* D(a)[x := D(b)]$ follows from applying Lemma 2 to a, inductive hypothesis to b', and Lemma 1 outside.

- If $t = ab$ with a not being an abstraction, t reduces to either $a'b$ or ab' (where $a \succ a'$, $b \succ b'$). $D(t) = D(a)D(b)$, Apply Lemma 2 and induction hypothesis to the suitable subterm again, and we are done.

□

Lemma 4. *It $t \succ u$ then $D(t) \succ^* D(u)$*

Proof. Again, by induction over t, considering the possible beta reductions.

- $t = x$ is trivial, no reduction.
- $t = \lambda x.a$ can only reduce to $u = \lambda x.a'$ with $a \succ a'$, then $D(t) = \lambda x.D(a)$ and $D(u) = \lambda x.D(a')$, from induction hypothesis (a is smaller than t) $D(a) \succ^* D(a')$ and the result follows.
- If $t = ab$ where a is not an abstraction, it is inmediate again applying the induction hypothesis in the term that is reduced (this is allways the easy -inductive- case).
- If $t = (\lambda x.a)b$, t can reduce to $(\lambda x.a')b$, $(\lambda x.a)b'$ or $a[x := b]$ (where $a \succ a'$, $b \succ b'$). In the last two cases it is easy to apply the induction hypothesis, in the former one, when t reduces to $u = a[x := b]$, so we must show that $D(t) = D((\lambda x.a)b) = D(a)[x := D(b)]$ reduces (in $*$ steps) to $D(a[x := b])$. We demonstrate this in the next lemma.

□

Lemma 5. $D(t)[x := D(u)] \succ^* D(t[x := u])$

Proof. By induction over t.

- If $t = y$, again it is trivial. If y = x both sides are $D(u)$, if $y \neq x$ both sides are $D(y)$.
- if $t = \lambda y.a$ then

$D(t)[x := D(u)]$	=
$D(\lambda y.a)[x := D(u)]$	= { by definition of D}
$(\lambda y.D(a))[x := D(u)]$	={by definition of substitution, assuming $y \neq x$, otherwise rename y }
$\lambda y.(D(a)[x := D(u)])$	$\succ^*$ {by inductive hypothesis}
$\lambda y.D(a[x := u])$	={by definition of D}
$D(\lambda y.(a[x := u]))$	={by definition of substitution}
$D((\lambda y.a)[x := u])$	=
$D(t[x := u])$	

- The application case is tricky. We must study separate cases, one more than before, We will see why.

– If $t = (\lambda y.a)b$:

$D(t)[x := D(u)]$ =

$D((\lambda y.a)b)[x := D(u)]$ =

$D((\lambda y.a)b)[x := D(u)]$ = { by definition of D}

$(D(a)[y := D(b)])[x := D(u)]$ $\succ^*$ { by substitution lemma}

$(D(a)[x := D(u)])[y := D(b)[x := D(u)]]$ $\succ^*$ { by inductive hypothesis }

$(D(a[x := u])[y := D(b)[x := D(u)]]$ $\succ^*$ by inductive hypothesis

$(D(a[x := u])[y := D(b[x := u]]$ = by definition of D

$D((\lambda y.a[x := u])(b[x := u]))$ = by definition of substitution

$D(((\lambda y.a)[x := u])(b[x := u]))$ = by definition of substitution

$D((\lambda y.a)b[x := u])$ =

$D(t[x := u])$

– If $t = ab$, and a is not a variable, then:

$D(t)[x := D(u)]$ =

$D(ab)[x := D(u)]$ = { by definition of D}

$(D(a)D(b))[x := D(u)]$ = { by definition of substitution}

$(D(a)[x := D(u)])(D(b)[x := D(u)])$ ={by hypothesis, twice}

$D(a[x := u])D(b[x := u])$ = { by definition of D}

$D(a[x := u]b[x := u])$ = { by definition of substitution}

$D((ab)[x := u])$ =

$D(t[x := u])$

– If $t = xb$ this case is different to the previous one since the substitution could generate a new redex. Note that if $t = yb$ where $y \neq x$ the proof is like in the former case.

$D(t)[x := D(u)]$ =

$D(xb)[x := D(u)]$ = { by definition of D}

$(D(x)D(b))[x := D(u)]$ = { by definition of substitution}

$(D(x)[x := D(u)])(D(b)[x := D(u)])$ = { by definition of substitution and D }

$(D(u))(D(b)[x := D(u)])$ = { applying inductive hypothesis}

$(D(u))(D(b[x := u]))$ = { applying substitution backwards}

If u is not an abstraction there is no problem (rewrite the definition of D backwards and so on). If $u = \lambda y.c$ then:

$(D(u))(D(b[x := u]))$ =

$(D(\lambda y.c))(D(b[x := u]))$ = { by definition of D}

$(\lambda y.D(c))(D(b[x := u]))$ $\succ$

$D(c)[y := (D(b[x := u]))]$ = { rewriting D}

$D((\lambda y.c)(b[x := u]))$ =

$D(u(b[x := u]))$ = { by definition of substitution}

$D((x[x := u])(b[x := u]))$ = { by definition of substitution}

$D(xu[x := u])$

□

Lemma 6. *$t \succ^* u$ implies $D(t) \succ^* D(u)$*

Proof. By induction over the number of steps of the $t \succ^* u$ reduction. If $n = 1$ then this is an instance of Lemma 4.
If $t \succ^n u$ then consider the first reduction, so $t \succ t'$, $t' \succ^{n-1} u$. By induction hypothesis $D(t') \succ^* D(u)$, and by Lemma 4 $D(t) \succ^* D(t')$. By transitivity of $\succ^*$ follows that $D(t) \succ^* D(u)$. □

Lemma 7. *$t \succ^n u$ implies $u \succ^* D^n(t)$*

Proof. Again, by induction over the length of reduction.
For the base case, this is exactly Lemma 3.
For $t \succ^n u$ with $n > 1$ consider again the first reduction, so $t \succ t'$ and $t' \succ^{n-1} u$. $t' \succ^* D(t)$ by Lemma 3, and by induction hypothesis $u \succ^* D^{n-1}(t')$, Applying Lemma 6 $D^{n-1}(t') \succ^* D^{n-1}(D(t)) = D^n(t)$. By transitivity of $\succ^*$ then $u \succ^* D^n(t)$ as we claim. □

Theorem 1 (Church-Rosser Theorem)**.** *The relation $\succ^*$ is confluent. This is, if $t \succ^* t_1$ and $t \succ^* t_2$ then there is a term v such that $t_1 \succ^* v$ and $t_2 \succ^* v$.*

Proof. If $t \succ^n t_1$ and $t \succ^m t_2$, then applying Lemma 7 $t_1 \succ^* D^n(t)$ and $t_2 \succ^* D^m(t)$. Without loss of generality assume that $m \geq n$. It is easy to show that $D^n(t) \succ^* D^m(t)$ (Applying Lemma 2 $m - n$ times, $t \succ^* D^{m-n}(t)$, then Apply Lemma 6 n times, or more formal, we can prove by an easy induction).
Then by transitivity of $\succ^*$ is true that $t_1 \succ^* D^m(t)$, take $u = D^m(t)$. □

2 Exercise 21

In general, in simply typed λ-calculus if we can derive a typing judgement $\Gamma \vdash t : A$ then we can derive it extending the context.
Also, if we can make a derivation in HOL, expanding the context (of assumptions) we can also make the derivation (just ignore the new axioms).
Let's be more precise:

Lemma 8. *If $\Gamma \vdash t : A$ and $x \notin Dom(\Gamma)$ then $\Gamma, x : X \vdash t : A$*

Proof. This is trivial by induction on t. Since the typing derivation is syntax directed, for each shape of t there is only one rule that could be applied in that case.

- If $t = x$ then the only possible derivation is

$$\frac{\Gamma(x) = A}{x : A}$$

  and then the following derivation is well-formed:

$$\frac{(\Gamma, y : X)(x) = A}{x : A}$$

  since the extension of a function does not change the value in a point of the previous domain.

- If $t = \lambda x.b$ then the typing derivation must be like:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \Rightarrow B}$$

  Then the following derivation is well formed:

$$\frac{\Gamma, x : A, y : X \vdash b : B}{\Gamma, y : X \vdash \lambda x.b : A \Rightarrow B}$$

- The application case is even more trivial since there is no manipulation of the context.

□

Lemma 9. *If $\Gamma \vdash t : A$ then $\Gamma, \Delta \vdash t : A$, assuming Γ and Δ with disjoint domains.*

Proof. This is trivial by induction in the size of Δ. Note that we are assuming that contexts are finite. □

Then if $\gamma \vdash A : o$ we have $\gamma, \delta \vdash A : o$. So if Γ is a well formed set of axioms under the context γ of the underlying simply typed lambda calculus, it is well formed under γ, δ.
So $\gamma; \Gamma \vdash A$ implies $\gamma, \delta; \Gamma \vdash A$
.
.
.
.
.
.
This rule is not derivable. We are not doing sequent calculus so there is no way to introduce formulas in the left. Formally this is proved trivially by induction over the rules.

3 Exercise 21

The first rule is admissible but cannot be derived.
For the second rule, let's do a formal proof:

Theorem 2. *The following rule is derivable, assuming $x \notin FV(B)$:*

$$\frac{\gamma;\Gamma \vdash \exists_s x.A \qquad \gamma, x:s;\Gamma \vdash A \Rightarrow B}{\gamma;\Gamma \vdash B}\ \exists_e$$

Proof. Note that $\exists_s xA \equiv_{\text{def}} (\lambda X.\forall_o Y.(\forall_s x.Xx \Rightarrow Y) \Rightarrow Y)A \succ_\beta \forall_o Y.(\forall_s x.Ax \Rightarrow Y) \Rightarrow Y$

$$\dfrac{\dfrac{\gamma, x:s;\Gamma \vdash A \Rightarrow B}{\gamma;\Gamma \vdash \forall x_s.(Ax \Rightarrow B)}\ \forall_i \qquad \dfrac{\gamma \vdash B : o \qquad \dfrac{\dfrac{\gamma \vdash \exists_s xA : o \qquad \forall_o Y.(\forall_s x.Ax \Rightarrow Y) \Rightarrow Y \equiv_\beta \exists_s xA \qquad \gamma;\Gamma \vdash \exists_s xA}{\gamma;\Gamma \vdash \forall_o Y.(\forall_s x.Ax \Rightarrow Y) \Rightarrow Y}\ \beta}{\gamma;\Gamma \vdash (\forall_s x.Ax \Rightarrow B) \Rightarrow B}\ \forall_e}{\gamma;\Gamma \vdash B}\ \Rightarrow_e$$

Note that leaves are axioms, a β-equivalence, and $\gamma \vdash B : o$ which must be true since $\gamma, x : s \vdash (A \Rightarrow B) : o$

□