Resolution of exercises

Juan García Garland

December 16, 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, since when checking the context the only interesting thing is if typing judgements belong to it or not.
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. (Weakening).
Let's be more precise:

Lemma 8 (Weakening in simply typed λ-calculus)**.** *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:

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

  applying the induction hypothesis to the term b.

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

  If $t = uv$ then the root of the typing proof is:

  $$\frac{\Gamma \vdash a : A \Rightarrow B \qquad \Gamma \vdash b : A}{\Gamma \vdash a\, b : B}$$

  Then the following derivation is well formed:

  $$\frac{\Gamma, x : T \vdash a : A \Rightarrow B \qquad \Gamma, x : T \vdash b : A}{\Gamma \vdash a\, b : B}$$

  applying induction hypothesis to a and b.

□

Lemma 9 (Weakening in HOL)**.** *If* $\gamma; \Gamma \vdash t : A$ *is derivable then* $\gamma; \Gamma, \Delta \vdash t : A$ *is derivable, assuming* Γ *and* Δ *with disjoint domains.*

Proof. I'll be more sloppy here since otherwise this is very verbose.
This lemma is trivial by induction in the size of Δ once we prove that we can add a proposition to a context. (Note that we are assuming that contexts are finite).
To prove this, we can use induction over the rules. For example for the Axiom rule we only check that $A \in \Gamma$, so this is true when we extend Γ.
For inductive rules it is the same, in every rule either we do not touch the context or we check if something belongs to it, which is conservative when we extend it.

□

Theorem 2 (Exercise 21)**.** *If* $\gamma; \Gamma \vdash A$ *is derivable then* $\gamma, \delta; \Gamma, \Delta \vdash A$ *is derivable.*

If $\gamma \vdash A : o$ is derivable then we have $\gamma, \delta \vdash A : o$ derivable by weakening in λ-calculus. 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 $\gamma; \Gamma \vdash A$ derivable implies $\gamma, \delta; \Gamma \vdash A$ derivable. Applying weakening for HOL then $\gamma, \delta; \Gamma, \Delta \vdash A$ can be derived, as we wanted.
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. Note that when applying a rule, the context is decreasing, by induction this is true for a proof, which contradicts the possibility of derive this rule (At least if δ and Δ are empty).

3 Exercise 24

Let's start for what we can prove. The second rule is ok even without weakening.

Theorem 3. *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 $\exists_s xA \equiv_{\text{def}} (\lambda X.\forall_o Y.(\forall_s x.Xx \Rightarrow Y) \Rightarrow Y)A \succ_\beta \forall_o Y.(\forall_s x.Ax \Rightarrow Y) \Rightarrow Y$

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

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

□

To derive a left rule, we need to manipulate context:

Theorem 4 (Left $\exists$-elimination)**.** *The following rule is admissible:*

$$\frac{\gamma;\Gamma, \exists_s x.A \vdash B}{\gamma;\Gamma, A[x := e] \vdash B}\ L\exists_e$$

Proof. We write a derivation, note that leaves are premises. Also note that we use weakening and a rule that was not derivable without it ($\exists_i$). So we are proving admissibility:

$$\dfrac{\dfrac{\dfrac{\gamma;\Gamma, \exists_s x.A \vdash B}{\gamma;\Gamma \vdash \exists_s x.A \Rightarrow B}\ \forall_i}{\gamma;\Gamma, A[x := e] \vdash \exists_s x.A \Rightarrow B}\ wk \qquad \dfrac{\gamma \vdash e : s \qquad \dfrac{}{\gamma, \Gamma, A[x := e] \vdash A[x := e]}\ \text{Ax}}{\gamma;\Gamma, A[x := e] \vdash \exists_s x.A}\ \exists_i}{\gamma;\Gamma, A[x := e] \vdash B}\ \Rightarrow_e$$

□

The first rule given is also admissible. Again, now that we showed some formal proofs, let's be more sloppy here. (And also because actually we will be using admissible but not derivable rules everywhere, to be really formal we need some boring work before).

We want to prove $\gamma;\Gamma \vdash \exists_s x.A$, if we unfold the definition of $\exists$ and do a beta reduction on the goal this is $\gamma;\Gamma \vdash \forall Y.(\forall_s x.A(x) \Rightarrow Y) \Rightarrow Y$. Introducing the universal quantifier and the first paremeter of the arrow, the goal is $\gamma;\Gamma, Y : o, \forall_s x.A(x) \Rightarrow Y \vdash Y$. Since we can derive $A[x := e]$, applying $\forall_s x.A(x) \Rightarrow Y$ to e and $A[x := e]$ we have what we want.

Note that we are using a lot of non-primitive rules here.

4 Exercise 26

Again we will not show a strict formal proof, but they can be implemented trivially. We assume implicitly that we previously derived some usual rules such as RAA or $\iff_{intro}$. So to prove equivalence we prove double implication. Also in the second equivalence we make a proof by contradiction, and use $\neg\exists_s x.\neg A \iff \forall_s x.A$.

Theorem 5. $A[x := \epsilon_s(\lambda x.A)] \Leftrightarrow \exists_s x.A$

Proof. The left to right implication is trivial applying the rule $\exists_i$ of the exercise 24. We only need that $\gamma \vdash \epsilon_s(\lambda x.A) : s$ which is well typed.
For the opposite we apply the $\exists_e$ rule of the exercise 24.

□

Theorem 6. $A[x := \epsilon_s(\lambda x.\neg A)] \Leftrightarrow \forall_s x.A$

Proof. We are using classical logic so assume by contradiction $\exists_s x.\neg A$. Then applying $\exists_e$ with $\neg A \Rightarrow \neg A[x := \epsilon_s(\lambda x.\neg A)]$ (which is provable) we prove $\neg A[x := \epsilon_s(\lambda x.\neg A)]$. By contraposition, we proved $A[x := \epsilon_s(\lambda x.\neg A)] \Rightarrow \forall_s x.A$ (using De Morgan's law and double negation elimination).
The opposite is trivial, apply the ϵ rule. □

5 Exercise 30

This was solved in Coq (`Arith.v`).

6 Exercise 31

Also implemented in `Arith.v`.

7 Exercise 37 - Arithmetic functions in SystemF

One way to solve this exercise is to do in the same way then in Exercise 14, constructing a proof tree and collecting equations to compose a system and then find a unifier.
This could work by hand in this small examples, as a guide to figure out types, but not as a general rule as SystemF typing is not decidable. Proofs are no longer syntax directed and unification is no longer first order. [1]

Since we have to write **a** type, and it is very intuitive which one we want, We'll show a typing derivation for that type instead of trying to infer one from scratch. To define arithmetic operations we use the very well known terms, we will not prove here that they reduce to the desired value.
Remark: Note that each time we apply a forall introduction, the binded variable does not occurs free in the context.
We will use the notation $\mathbb{N} := \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X$, folding and unfolding this definition when we need it.

Addition

Theorem 7. *Let* $Add = \lambda mnfx.mf(nfx)$*, then:* $\emptyset \vdash Add : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$

Proof. We construct a proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash\ m : \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X}\text{Ax}}{\Gamma \vdash\ m : (X \Rightarrow X) \Rightarrow X \Rightarrow X}\forall_e \quad \dfrac{}{\Gamma \vdash f : X \Rightarrow X}\text{Ax}}{\Gamma \vdash\ mf : X \Rightarrow X}\Rightarrow_e \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash n : \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X}\text{Ax}}{\Gamma \vdash n : (X \Rightarrow X) \Rightarrow X \Rightarrow X}\forall_e \quad \dfrac{}{\Gamma \vdash\ f : X \Rightarrow X}\text{Ax}}{\Gamma \vdash\ nf : X \Rightarrow X}\Rightarrow_e \quad \dfrac{}{\Gamma \vdash\ x : X}\text{Ax}}{\Gamma \vdash\ nfx : X}\Rightarrow_e}{(\Gamma \equiv) m, n : \mathbb{N}, f : X \Rightarrow X, x : X \vdash\ mf(nfx) : X}\Rightarrow_i}{m, n : \mathbb{N}, f : X \Rightarrow X \vdash\ \lambda x.mf(nfx) :\Rightarrow X \Rightarrow X}\Rightarrow_i}{m, n : \mathbb{N} \vdash\ \lambda fx.mf(nfx) : (X \Rightarrow X) \Rightarrow X \Rightarrow X}\forall_i}{m, n : \mathbb{N} \vdash\ \lambda fx.mf(nfx) : \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X (\equiv \mathbb{N})}\Rightarrow_i}{m : \mathbb{N} \vdash\ \lambda nfx.mf(nfx) : \mathbb{N} \Rightarrow \mathbb{N}}\Rightarrow_i}{\vdash \lambda m\, n\, f\, x.mf(nfx) : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}}\Rightarrow_i$$

□

Mutiplication

Theorem 8. *Let* $Mult = \lambda mnfx.m(nf)x$*, then:* $\emptyset \vdash Mult : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$

Proof. Again, we construct the derivation. To have a smaller proof we apply multiple elimination rules in one step (the usual intros tactic).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash m : \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X}\text{Ax}}{\Gamma \vdash m : (X \Rightarrow X) \Rightarrow X \Rightarrow X}\forall_i \quad \dfrac{\dfrac{}{\Gamma \vdash n : \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X}\text{Ax}}{\Gamma \vdash n : (X \Rightarrow X) \Rightarrow X \Rightarrow X}\forall_i \quad \dfrac{}{\Gamma \vdash f : X \Rightarrow X}\text{Ax}}{\Gamma \vdash (nf) : X \Rightarrow X}\Rightarrow_e}{\Gamma \vdash m(nf) : X \Rightarrow X}\Rightarrow_e \quad \dfrac{}{\Gamma \vdash x : X}\text{Ax}}{(\Gamma :=) m, n : \mathbb{N}, f : X \Rightarrow X, x : X \vdash m(nf)x : X}\Rightarrow_e}{\vdash \lambda mnfx.m(nf)x : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}}\Rightarrow_i, \Rightarrow_i, \forall_i, \Rightarrow_i, \Rightarrow_i$$

□

[1] We saw in the course an interesting way to (if I understood it well) move all complexity to the equations doing subtyping, which at least greatly improves error messages in an implementation when the algorithm stucks

Exponentiation

Theorem 9. *Let* $Exp = \lambda mn.nm$*, then:* $\emptyset \vdash Exp : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$

Proof. This is by far the most interesting example. Recall that this function did not have this desired type in simply typed λ-calculus. Now it is possible to type it as we wish. This shows the power of forall quantification. Note that when we eliminate the $\forall$ quantifier (reading the proof bottom up), the instantiation is not syntax directed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{m,n:\mathbb{N} \vdash n:\mathbb{N}}\,\mathrm{Ax}}{m,n:\mathbb{N} \vdash n:((X \Rightarrow X) \Rightarrow (X \Rightarrow X)) \Rightarrow (X \Rightarrow X) \Rightarrow (X \Rightarrow X)}\,\forall_e\,[X/X \Rightarrow X] \quad \dfrac{\dfrac{}{m,n:\mathbb{N} \vdash m:\mathbb{N}}\,\mathrm{Ax}}{m,n:\mathbb{N} \vdash m:(X \Rightarrow X) \Rightarrow (X \Rightarrow X)}\,\forall_e\,[X/X]}{m,n:\mathbb{N} \vdash nm:(X \Rightarrow X) \Rightarrow (X \Rightarrow X)}}{m,n:\mathbb{N} \vdash nm:\forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X}\,\forall_i}{\vdash \lambda mn.nm:\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}}\,\Rightarrow_i,\Rightarrow_i$$

□

8 Exercise 38 - Product and Coproduct in System F

Terms are the same as in simply typed lambda calculus. Again we will ommit computing reductions (obviowsly we must prove that this terms reduces to what we want) since this was well studied in class and at IMERL's hall, in previous exercises, (and actially it is very easy in most cases).
Type derivations, in the other hand, will be provided.

Product

Let:

$$A \times B := \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X$$

The type of the -unique- constructor *pair*, and destructors π_1 and π_2 are:

$$\begin{aligned} pair &: A \Rightarrow B \Rightarrow A \times B \\ \pi_1 &: A \times B \Rightarrow A \\ \pi_2 &: A \times B \Rightarrow B \end{aligned}$$

where they are defined as:

$$\begin{aligned} pair &:= \lambda abp.pab \\ \pi_1 &:= \lambda p.p(\lambda tf.t) \\ \pi_2 &:= \lambda p.p(\lambda tf.f) \end{aligned}$$

We do not use actual booleans since they would not be well typed here.
Denoting *pair* a b as (a, b) it is easy to show that:

$$\begin{aligned} \pi_1(a, b) &\succ^* a \\ \pi_2(a, b) &\succ^* b \end{aligned}$$

The typing proofs are the following:
For π_1 (for π_2 is analogous):

$$\dfrac{\dfrac{\dfrac{}{p : \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X \vdash p : \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X}\,\text{Ax}}{p : \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X \vdash p : (A \Rightarrow B \Rightarrow A) \Rightarrow A}\,\forall_e\,[X/A] \quad \dfrac{\dfrac{}{p : \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X, t : A, f : B \vdash t : A}\,\text{Ax}}{p : \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X \vdash (\lambda tf.t) : (A \Rightarrow B \Rightarrow A)}\,\Rightarrow_i, \Rightarrow_i}{\dfrac{p : \forall X.(A \Rightarrow B \Rightarrow X) \Rightarrow X \vdash p(\lambda tf.t) : A}{\vdash \lambda p.p(\lambda tf.t) : A \times B \Rightarrow A}\,\Rightarrow_i}$$

For *pair*:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash p : A \Rightarrow B \Rightarrow X}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash a : A}\,\text{Ax}}{\Gamma \vdash p\,a : B \Rightarrow X}\Rightarrow_e \quad \dfrac{}{\Gamma \vdash b : B}\,\text{Ax}}{(\Gamma :=) a : A, b : B, p : A \Rightarrow B \Rightarrow X \vdash pab : X}\Rightarrow_e}{\vdash \lambda abp.pab : A \Rightarrow B \Rightarrow A \times B}\Rightarrow_i, \Rightarrow_i, \forall_i, \Rightarrow_i$$

Co-Product

Let:

$$A + B := \forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X$$

Now we have two constructors, *inl*, *inr* and the case analysis destructor C:

$$\begin{aligned} inl &: A \Rightarrow A + B \\ inr &: B \Rightarrow A + B \\ C &: A + B \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C \end{aligned}$$

where they are defined as:

$$\begin{aligned} inl &:= \lambda alr.la \\ inr &:= \lambda blr.rb \\ C &:= \lambda x.x \end{aligned}$$

Note that we chosed the simple implementation of C now, in contrast to Exo 5, (but of course $C = \lambda cfg.cfg$ typechecks also, they are η-equivalent).
We provide typing proofs for *inl* and for C (*inr* is analogous to *inl*)

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash l : A \Rightarrow X}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash a : A}\,\text{Ax}}{(\Gamma :=) a : A, l : A \Rightarrow X, r : B \Rightarrow X \vdash la : X}\Rightarrow_e}{\vdash \lambda alr.la : A \Rightarrow \forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X}\Rightarrow_i, \forall_i, \Rightarrow_i, \Rightarrow_i$$

$$\dfrac{\dfrac{\dfrac{}{x : (\forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X \vdash x : (\forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X}\,\text{Ax}}{x : (\forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X) \vdash x : (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C}\forall_e\,[X/C]}{\vdash \lambda x.x : (\forall X.(A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X) \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C}\Rightarrow_i$$

When we make the substitution $X := C$ in types we are actually assuming a more general context where C is a type, no problem.

9 Exercise 39 - Lists in System F

For this excercise we could reason in the algebraic way, since we have Product and coproduct, or directly encode lists usind the type of its constructors. The result is the same.
Let:

$$A^* := \forall X.X \Rightarrow (A \Rightarrow X \Rightarrow X) \Rightarrow X$$

We want to program *nil* and *cons*, constructors with types:

$$\begin{aligned} nil &: A^* \\ cons &: A \Rightarrow A^* \Rightarrow A^* \end{aligned}$$

The reasoning when programming is the same as ever when we encode datatypes, each (sub)term with type A^* should have a lambda for each existing constructor, and the application of the suitable one in the body, to the actual parameters that the constructor has, which are furthermore closed by outermost abstractions.
So:

$$\begin{aligned} nil &= \lambda nc.n \\ cons &= \lambda alnc.ca(lnc) \end{aligned}$$

Typing proofs:

$$\dfrac{\dfrac{}{n : X, c : A \Rightarrow X \Rightarrow X \vdash n : X}\,\text{Ax}}{\vdash \lambda nc.n : \forall X.X \Rightarrow (A \Rightarrow X \Rightarrow X) \Rightarrow X}\,\forall_i, \Rightarrow_i, \Rightarrow_i$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash c : A \Rightarrow X \Rightarrow X}\,\text{Ax} \quad \dfrac{}{\Gamma \vdash a : A}\,\text{Ax}}{\Gamma \vdash ca : X \Rightarrow X}\,\Rightarrow_e \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash l : \forall X.X \Rightarrow (A \Rightarrow X \Rightarrow X) \Rightarrow X)}\,\text{Ax}}{\Gamma \vdash l : X \Rightarrow (A \Rightarrow X \Rightarrow X) \Rightarrow X}\,\forall_e \quad \dfrac{}{\Gamma \vdash n : X}\,\text{Ax}}{\Gamma \vdash ln : (A \Rightarrow X \Rightarrow X) \Rightarrow X)}\,\Rightarrow_e \quad \dfrac{}{\Gamma \vdash c : A \Rightarrow X \Rightarrow X}\,\text{Ax}}{\Gamma \vdash lnc : X}\,\Rightarrow_e}{(\Gamma :=) a : A, l : A^*, n : X, c : A \Rightarrow X \Rightarrow X \vdash ca(lnc) : X}\,\Rightarrow_i, \Rightarrow_i, \forall_i, \Rightarrow_i$$

$$\vdash \lambda alnc.ca(lnc) : A \Rightarrow A^* \Rightarrow A^*$$

Finally, the map function must have type:

$$map : (A \Rightarrow B) \Rightarrow A^* \Rightarrow B^*$$

(We could close type variables to have one polymorphic function instead of this version parametrized by A and B, but the proof is the same, adding $\forall$ introductions on the root).
Define it as:

$$map := map = \lambda flnc.ln(\lambda v.c(fv))$$

Note that actually $(\lambda v.c\ (f\ v))$ can be sugarized as $c \circ f$. We could add a typing rule or a lemma to type compositions, but since it is only used here we prefer to unfold to the pointwise term when used.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash l : \forall X.X \Rightarrow (A \Rightarrow X \Rightarrow X) \Rightarrow X)}\,\mathrm{Ax}}{\Gamma \vdash l : X \Rightarrow (A \Rightarrow X \Rightarrow X) \Rightarrow X}\,\forall_e \quad \dfrac{}{\Gamma \vdash n : X}\,\mathrm{Ax}}{\Gamma \vdash ln : (A \Rightarrow X \Rightarrow X) \Rightarrow X}\,\Rightarrow_e \quad \dfrac{\dfrac{}{\Gamma_2 \vdash c : B \Rightarrow X \Rightarrow X}\,\mathrm{Ax} \quad \dfrac{\dfrac{}{\Gamma_2 \vdash f : A \Rightarrow B}\,\mathrm{Ax} \quad \dfrac{}{\Gamma_2 \vdash v : A}\,\mathrm{Ax}}{\Gamma_2 \vdash fv : B}\,\Rightarrow_e}{(\Gamma_2 :=)\Gamma, v : A \vdash c(fv) : X \Rightarrow X}\,\Rightarrow_i}{\Gamma \vdash \lambda v.c(fv) : A \Rightarrow X \Rightarrow X}\,\Rightarrow_e}{(\Gamma :=) f : A \Rightarrow B, l : A^*, n : X, c : B \Rightarrow X \Rightarrow X \vdash ln(c \circ f) : X}\,\Rightarrow_i, \Rightarrow_i, \forall_i, \Rightarrow_i, \Rightarrow_i$$
$$\vdash \lambda flnc.ln(c \circ f) : (A \Rightarrow B) \Rightarrow A^* \Rightarrow B^*$$

Note that in the $\forall_e$ we substitute the type X (a type in the context) to put it instead of the variable X. Finally It is easy to show that *map* satisfies its specification:

$$\begin{aligned} map\ f\ nil &\succ^* nil \\ map\ f\ (cons\ a\ l) &\succ^* cons\ (f\ a)(map\ f\ l) \end{aligned}$$