assignment09

July 10, 2019

1 Functional Programming SS19

2 Assignment 09 Solutions

2.0.1 Exercise 1 (Pure Lambda Calculus)

a) *We use the representation of Boolean values in the pure λ-calculus presented in the lecture, i.e., True is represented as $\lambda xy.x$ and False as $\lambda xy.y$. Using this representation give pure λ-terms for the following Boolean functions:*

(i) *xor, such that e.g.* $\overline{xor}\,\overline{True}\,\overline{True} \rightarrow_\beta^* False$ *and* $\overline{xor}\,\overline{False}\,\overline{True} \rightarrow_\beta^* True$

Given two arguments p and q, we can represent *xor* as $(p \wedge \neg q) \vee (\neg p \wedge q)$. We thus first need the pure lambda calculus representations of *or* and *and*.

We can represent *or* in pure lambda calculus as follows:

$$or = \lambda xy.x\ x\ y$$

Intuitively, we have that, if the first argument is $\overline{True}$, the result of the expression after β-reduction is also $\overline{True}$ since $x\ x\ y$ takes the first argument of *or* as a result when $x = \overline{True}$; on the other hand, if the first argument is $\overline{False}$, the result after β-reduction is the second argument of *or* since $x\ x\ y$ takes the second argument when $x = \overline{False}$.

Similarly, we can represent *and* in pure lambda calculus as follows:

$$and = \lambda xy.x\ y\ x$$

In this case, the result of the expression after β-reduction is the second argument of *and* if the first argument is $\overline{True}$ since $x\ y\ x$ takes the second argument when $x = \overline{True}$. If the first argument is $\overline{False}$, the result after β-reduction is also $\overline{False}$ since $x\ y\ x$ takes the first argument of *and* as a result when $x = \overline{False}$.

Using the representation of *not* given in the hint, namely $\overline{not} = \lambda a.a\ (\lambda xy.y)\ (\lambda xy.x)$, we can now write *xor* in pure lambda calculus as

$$xor = \lambda pq.(exp_1\ exp_1\ exp_2)$$

where

$$\begin{aligned} exp_1 &= p\ ((\lambda a.a\ (\lambda xy.y)\ (\lambda xy.x))\ q)\ p = p\ (\overline{not}\ q)\ p \\ exp_2 &= ((\lambda a.a\ (\lambda xy.y)\ (\lambda xy.x))\ p)\ q\ ((\lambda a.a\ (\lambda xy.y)\ (\lambda xy.x))\ p) = (\overline{not}\ p)\ q\ (\overline{not}\ p) \end{aligned}$$

(ii) *implies, such that e.g.* $\overline{implies}\overline{False}\overline{True} \to_\beta^* \overline{True}$ *and* $\overline{implies}\overline{True}\overline{False} \to_\beta^* \overline{False}$

Given two arguments p and q, the operator *implies* is equivalent to $p\ implies\ q \iff \neg p\ or\ q$, which is the form we will use here. Using the definition of *not* given in the hint, namely $\overline{not} = \lambda a.a\ (\lambda xy.y)\ (\lambda xy.x)$, and the definition of *or* from above, *implies* can be written as follows:

$$\begin{aligned} \overline{implies} &= \lambda pq.(((\lambda a.a\ (\lambda xy.y)\ (\lambda xy.x))\ p)\ ((\lambda a.a\ (\lambda xy.y)\ (\lambda xy.x))\ p)\ q) \\ &= \lambda pq.(\overline{not}\ p)\ (\overline{not}\ p)\ q \end{aligned}$$

b) *We use the representation of Boolean values from part a). Additionally we represent natural numbers by* $\overline{n} = \lambda f\ x.f^n x$ *as in the lecture, i.e. 0 is represented by* $\lambda f\ x.x$*, 1 is represented by* $\lambda f\ x.f\ x$ *and so on. Using this representation, give* λ*-terms for the following functions and also give the indicated* β*-reduction sequence showing the correctness:*

(i) *mult and the reduction sequence* $\overline{mult}\ \overline{n}\ \overline{m} \to_\beta^* \overline{n \cdot m}$

We can represent multiplication using the following pure lambda calculus formulation:

$$\lambda nmfx.n\ ((\lambda pq.p)\ (m\ f\ x))\ x = \lambda nmfx.n\ (\overline{True}\ (m\ f\ x))\ x$$

The reduction sequence $\overline{mult}\ \overline{n}\ \overline{m}$ is given below:

$$\begin{aligned} &\overline{mult}\ \overline{n}\ \overline{m} = (\lambda nmfx.n\ ((\lambda pq.p)\ (m\ f\ x))\ x)\ (\lambda gy.g^n y)\ (\lambda hz.h^m z) \\ &\to_\beta (\lambda mfx.(\lambda gy.g^n y)\ ((\lambda pq.p)\ (m\ f\ x))\ x)\ (\lambda hz.h^m z) \\ &\to_\beta \lambda fx.(\lambda gy.g^n y)\ ((\lambda pq.p)\ ((\lambda hz.h^m z)\ f\ x))\ x \\ &\to_\beta \lambda fx.(\lambda gy.g^n y)\ ((\lambda pq.p)\ ((\lambda z.f^m z)\ x))\ x \\ &\to_\beta \lambda fx.(\lambda gy.g^n y)\ ((\lambda pq.p)\ (f^m x))\ x \\ &\to_\beta \lambda fx.(\lambda gy.g^n y)\ ((\lambda q.f^m)\ x)\ x \\ &\to_\beta \lambda fx.(\lambda gy.g^n y)\ f^m\ x \\ &\to_\beta \lambda fx.(\lambda y.f^{nm} y)\ x \\ &\to_\beta \lambda fx.f^{nm} x = \overline{n \cdot m} \end{aligned}$$

Just to illustrate this with a concrete example, let us use *mult* to find $\overline{2} \cdot \overline{3}$:

$$
\begin{aligned}
&\overline{mult}\ \overline{2}\ \overline{3} = (\lambda nmfx.n\ ((\lambda pq.p)\ (m\ f\ x))\ x)\ (\lambda gy.g^2y)\ (\lambda hz.h^3z) \\
&\rightarrow_\beta (\lambda mfx.(\lambda gy.g^2y)\ ((\lambda pq.p)\ (m\ f\ x))\ x)\ (\lambda hz.h^3z) \\
&\rightarrow_\beta \lambda fx.(\lambda gy.g^2y)\ ((\lambda pq.p)\ ((\lambda hz.h^3z)\ f\ x))\ x \\
&\rightarrow_\beta \lambda fx.(\lambda gy.g^2y)\ ((\lambda pq.p)\ ((\lambda z.f^3z)\ x))\ x \\
&\rightarrow_\beta \lambda fx.(\lambda gy.g^2y)\ ((\lambda pq.p)\ (f^3x))\ x \\
&\rightarrow_\beta \lambda fx.(\lambda gy.g^2y)\ ((\lambda q.f^3)\ x)\ x \\
&\rightarrow_\beta \lambda fx.(\lambda gy.g^2y)\ f^3\ x \\
&\rightarrow_\beta \lambda fx.(\lambda y.f^{2\cdot 3}y)\ x \\
&\rightarrow_\beta \lambda fx.f^6x
\end{aligned}
$$

(ii) *even and the reduction sequence* $\overline{even}\ \overline{n} \rightarrow_\beta^* \overline{not}^n\ \overline{True}$

We can represent the function *even* in lambda calculus as follows:

$$\lambda n.n\ \overline{not}\ \overline{True}$$

The reduction sequence $\overline{even}\ \overline{n}$ is given below:

$$
\begin{aligned}
&\overline{even}\ \overline{n} = (\lambda n.n\ \overline{not}\ \overline{True})\ (\lambda fx.f^nx) \\
&\rightarrow_\beta (\lambda fx.f^nx)\ (\overline{not}\ \overline{True}) \\
&\rightarrow_\beta (\lambda x.\overline{not}^n x)\ \overline{True} \\
&\rightarrow_\beta \overline{not}^n\ \overline{True}
\end{aligned}
$$

For instance, we have:

$$
\begin{aligned}
&\overline{even}\ \overline{o} = (\lambda n.n\ \overline{not}\ \overline{True})\ (\lambda fx.x) \\
&\rightarrow_\beta (\lambda fx.x)\ (\overline{not}\ \overline{True}) \\
&\rightarrow_\beta (\lambda x.x)\ \overline{True} \\
&\rightarrow_\beta \overline{True}
\end{aligned}
$$

$$
\begin{aligned}
&\overline{even}\ \overline{1} = (\lambda n.n\ \overline{not}\ \overline{True})\ (\lambda fx.fx) \\
&\rightarrow_\beta (\lambda fx.fx)\ (\overline{not}\ \overline{True}) \\
&\rightarrow_\beta (\lambda x.\overline{not}\ x)\ \overline{True} \\
&\rightarrow_\beta \overline{not}\ \overline{True} = \overline{False}
\end{aligned}
$$

$$\begin{aligned}
&\overline{even}\ \overline{2} = (\lambda n.n\ \overline{not}\ \overline{True})\ (\lambda f x.f^2 x)\\
&\rightarrow_\beta (\lambda f x.f^2 x)\ (\overline{not}\ \overline{True})\\
&\rightarrow_\beta (\lambda x.\overline{not}^2\ x)\ \overline{True}\\
&\rightarrow_\beta \overline{not}^2\ \overline{True} = \overline{True}
\end{aligned}$$

and so forth.
Hints:

- *You may use $\overline{not} = \lambda a.a\ (\lambda xy.y)(\lambda xy.x)$ in your solution.*
- *You may use the abbreviations $\overline{not}$, $\overline{True}$, $\overline{False}$ and $\overline{n}$ for $n \in \mathbb{N}$ in your solutions..*

2.0.2 Exercise 2 (Inferring Types Using the Algorithm $\mathcal{W}$)

In this exercise, please use the initial type assumption A_0 as presented in the lecture. This type assumption contains at least the following:

$$\begin{aligned}
A_0(1) &= \text{Int}\\
A_0(\text{True}) &= \text{Bool}\\
A_0(\text{plus}) &= \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\\
A_0(\text{Nil}) &= \forall a.\text{List } a\\
A_0(\text{fix}) &= \forall a.(a \rightarrow a) \rightarrow a\\
A_0(\text{if}) &= \forall a.\text{Bool} \rightarrow a \rightarrow a \rightarrow a\\
A_0(\text{Cons}) &= \forall a.a \rightarrow \text{List } a \rightarrow \text{List } a
\end{aligned}$$

Use the type inference algorithm $\mathcal{W}$ to determine the most general type of the following λ-terms. Show the results of all sub computations and unifications, too. If the term is not well typed, show at what step and why the $\mathcal{W}$-algorithm detects this and furthermore, give the most general non-shallow type scheme of the expression, if possible.

a) $\lambda x\ y.\text{if}\ y\ (\text{Cons}\ x\ \text{Nil})\ \text{Nil}$

The run of the $\mathcal{W}$ algorithm on the expression $\lambda x\ y.\text{if}\ y\ (\text{Cons}\ x\ \text{Nil})\ \text{Nil}$ is given below.

$$
\begin{aligned}
&\mathcal{W}(A_0, \lambda x\ y.\text{if } y\ (\text{Cons } x \text{ Nil}) \text{ Nil})\\
&\quad \mathcal{W}(A_0 + \{x :: b_1\}, \lambda y.\text{if } y\ (\text{Cons } x \text{ Nil}) \text{ Nil})\\
&\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: b_2\}, \text{if } y\ (\text{Cons } x \text{ Nil}) \text{ Nil})\\
&\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: b_2\}, \text{if } y\ (\text{Cons } x \text{ Nil}))\\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: b_2\}, \text{if } y)\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: b_2\}, \text{if})\\
&\quad\quad\quad\quad\quad = (id, \text{Bool} \to c \to c \to c)\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: b_2\}, \text{y})\\
&\quad\quad\quad\quad\quad = (id, b_2)\\
&\quad\quad\quad\quad\quad \text{mgu}(\text{Bool} \to c \to c \to c, b_2 \to b_3)\\
&\quad\quad\quad\quad\quad = [b_2/\text{Bool}, b_3/c \to c \to c]\\
&\quad\quad\quad\quad = ([b_2/Bool], c \to c \to c)\\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: Bool\}, \text{Cons } x \text{ Nil})\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: Bool\}, \text{Cons } x)\\
&\quad\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: Bool\}, \text{Cons})\\
&\quad\quad\quad\quad\quad\quad = (id, d \to \text{List } d \to \text{List } d)\\
&\quad\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1, y :: Bool\}, x)\\
&\quad\quad\quad\quad\quad\quad = (id, b_1)\\
&\quad\quad\quad\quad\quad\quad \text{mgu}(d \to \text{List } d \to \text{List } d, b_1 \to b_4)\\
&\quad\quad\quad\quad\quad\quad = [b_1/d, b_4/\text{List } d \to \text{List } d]\\
&\quad\quad\quad\quad\quad = ([b_1/d], \text{List } d \to \text{List } d)\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: d, y :: Bool\}, \text{Nil})\\
&\quad\quad\quad\quad\quad = (id, \text{List } e)\\
&\quad\quad\quad\quad\quad \text{mgu}(\text{List } d \to \text{List } d, \text{List } e \to b_5)\\
&\quad\quad\quad\quad\quad = [e/d, \text{List } e/\text{List } d, b_5/\text{List } d]\\
&\quad\quad\quad\quad = ([b_1/d], \text{List } d)\\
&\quad\quad\quad\quad \text{mgu}(c \to c \to c, \text{List } d \to b_6)\\
&\quad\quad\quad\quad = [c/\text{List } d, b_6/\text{List } d \to \text{List } d]\\
&\quad\quad\quad = ([b_1/d, b_2/\text{Bool}], \text{List } d \to \text{List } d)\\
&\quad\quad\quad \mathcal{W}(A_0 + \{x :: d, y :: Bool\}, \text{Nil})\\
&\quad\quad\quad = (id, \text{List } f)\\
&\quad\quad\quad \text{mgu}(\text{List } d \to \text{List } d, \text{List } f \to b_7)\\
&\quad\quad\quad = [f/d, b_7/\text{List } d]\\
&\quad\quad = ([b_1/d, b_2/\text{Bool}], \text{List } d)\\
&\quad = ([b_1/d, b_2/\text{Bool}], \text{Bool} \to \text{List } d)\\
&= ([b_1/d, b_2/\text{Bool}], d \to \text{Bool} \to \text{List } d)
\end{aligned}
$$

The type of $\lambda x\ y.\text{if } y\ (\text{Cons } x \text{ Nil}) \text{ Nil}$ is thus $d \to Bool \to \text{List } d$ for a generic type d.
For verification, the output of Haskell's type checker on the given expression is also included

below.

```
In [1]: data List a = Nil | Cons a (List a)

        instance Show a => Show (List a) where
            show Nil = "Nil"
            show (Cons x xs) = "Cons " ++ show x ++ " " ++ show xs

        f = \x y -> if y then (Cons x Nil) else Nil
        :t f
```

```
f :: forall a. a -> Bool -> List a
```

b) $\lambda x.\text{if } (x \text{ True}) \; (x \; 1) \; (x \; 1)$

$$
\begin{aligned}
&\mathcal{W}(A_0, \lambda x.\text{if } (x \text{ True}) \; (x\;1) \; (x\;1)) \\
&\quad \mathcal{W}(A_0 + \{x :: b_1\}, \text{if } (x \text{ True}) \; (x\;1) \; (x\;1)) \\
&\quad\quad \mathcal{W}(A_0 + \{x :: b_1\}, \text{if } (x \text{ True}) \; (x\;1)) \\
&\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1\}, \text{if } (x \text{ True})) \\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1\}, \text{if}) \\
&\quad\quad\quad\quad = (id, \text{Bool} \to b \to b \to b) \\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1\}, x \text{ True}) \\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1\}, x) \\
&\quad\quad\quad\quad\quad = (id, b_1) \\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_1\}, \text{True}) \\
&\quad\quad\quad\quad\quad = (id, \text{Bool}) \\
&\quad\quad\quad\quad\quad \text{mgu}(b_1, \text{Bool} \to b_2) \\
&\quad\quad\quad\quad\quad = [b_1/\text{Bool} \to b_2] \\
&\quad\quad\quad\quad = ([b_1/\text{Bool} \to b_2], b_2) \\
&\quad\quad\quad\quad \text{mgu}(\text{Bool} \to b \to b \to b, b_2) \\
&\quad\quad\quad\quad = [b_2/\text{Bool}] \\
&\quad\quad\quad = ([b_1/\text{Bool} \to \text{Bool}], b \to b \to b) \\
&\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool} \to \text{Bool}\}, x\;1) \\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool} \to \text{Bool}\}, x) \\
&\quad\quad\quad\quad = (id, \text{Bool} \to \text{Bool}) \\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool} \to \text{Bool}\}, 1) \\
&\quad\quad\quad\quad = (id, \text{Int}) \\
&\quad\quad\quad\quad \text{mgu}(\text{Bool} \to \text{Bool}, \text{Int} \to b_3) \\
&\quad\quad\quad\quad - \text{ERROR}
\end{aligned}
$$

The $\mathcal{W}$ algorithm fails at this point because there is no unifier for the types Bool and Int. Once again, the output of Haskell's type checker is also given below for verification purposes.

```
In [2]: \x -> if (x True) then (x 1) else (x 1)
```

```
Line 1: Redundant if
Found:
if (x True) then (x 1) else (x 1)
Why not:
(x 1)Line 1: Redundant bracket
Found:
if (x True) then (x 1) else (x 1)
Why not:
if x True then (x 1) else (x 1)Line 1: Redundant bracket
Found:
if (x True) then (x 1) else (x 1)
Why not:
if (x True) then x 1 else (x 1)Line 1: Redundant bracket
Found:
if (x True) then (x 1) else (x 1)
Why not:
if (x True) then (x 1) else x 1
```

```
<interactive>:1:27: error:
 No instance for (Num Bool) arising from the literal 1
 In the first argument of x, namely 1
  In the expression: (x 1)
  In the expression: if (x True) then (x 1) else (x 1)
```

c) fix $(\lambda x.\text{if } x \text{ (plus } x \text{ 1) } 1)$

$$
\begin{aligned}
&\mathcal{W}(A_0, \text{fix } (\lambda x.\text{if } x \text{ (plus } x \text{ 1) 1}))\\
&\quad \mathcal{W}(A_0, \text{fix})\\
&\quad = (id, (b_1 \to b_1) \to b_1)\\
&\quad \mathcal{W}(A_0, \lambda x.\text{if } x \text{ (plus } x \text{ 1) 1})\\
&\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, \text{if } x \text{ (plus } x \text{ 1) 1})\\
&\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, \text{if } x \text{ (plus } x \text{ 1)})\\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, \text{if } x)\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, \text{if})\\
&\quad\quad\quad\quad\quad = (id, \text{Bool} \to b \to b \to b)\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: b_2\}, x)\\
&\quad\quad\quad\quad\quad = (id, b_2)\\
&\quad\quad\quad\quad\quad \text{mgu}(\text{Bool} \to b \to b \to b, b_2 \to b_3)\\
&\quad\quad\quad\quad\quad = [b_2/\text{Bool}, b_3/b \to b \to b]\\
&\quad\quad\quad\quad = ([b_2/\text{Bool}], b \to b \to b)\\
&\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool}\}, \text{plus } x \text{ 1})\\
&\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool}\}, \text{plus } x)\\
&\quad\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool}\}, \text{plus})\\
&\quad\quad\quad\quad\quad\quad = (id, \text{Int} \to \text{Int} \to \text{Int})\\
&\quad\quad\quad\quad\quad\quad \mathcal{W}(A_0 + \{x :: \text{Bool}\}, x)\\
&\quad\quad\quad\quad\quad\quad = (id, \text{Bool})\\
&\quad\quad\quad\quad\quad\quad \text{mgu}(\text{Int} \to \text{Int} \to \text{Int}, \text{Bool} \to b_4)\\
&\quad\quad\quad\quad\quad\quad - \text{ERROR}
\end{aligned}
$$

As in b), the algorithm fails because there is no unifier for Bool and Int.
As before, we verify this outcome using the output of Haskell's type checker.

```
In [3]: fix x = x
        plus x y = x + y

        fix (\x -> if x then (plus x 1) else 1)
```

```
Line 1: Redundant bracket
Found:
if x then (plus x 1) else 1
Why not:
if x then plus x 1 else 1
```

```
<interactive>:1:23: error:
 No instance for (Num Bool) arising from a use of plus
 In the expression: (plus x 1)
```

```
In the expression: if x then (plus x 1) else 1
In the first argument of fix, namely (\ x -> if x then (plus x 1) else 1)
```