Principles of Programming Languages (Sheet #3)

Marius Gavrilescu

1.
```
(result v) $> f
=
(Ok v) $> f
=
f x

(Ok v) $> result
=
result v
=
Ok v

Fail $> result
=
Fail

(Ok v $> f) $> g
=
f v $> g
=
(\x -> (f x $> g)) v
=
(Ok v) $> (\x -> (f x $> g))

(Fail $> f) $> g
=
Fail $> g
=
Fail
=
Fail $> (\x -> (f x $> g))

orelse (orelse (Ok v) g) h
=
orelse (Ok v) h
=
Ok v
=
orelse (Ok v) (orelse g h)

orelse (orelse Fail g) h
=
```

```
orelse g h
=
orelse Fail (orelse g h)

orelse Fail f = f
orelse (Ok v) Fail = Ok v
orelse Fail Fail = Fail
```

2. `result v = Ok v`

```
(Ok v)   $> f = f v
(Fail v) $> f = Fail v

failure v = Fail v

orelse (Ok v)   g = Ok v
orelse (Fail v) g = g v

eval (OrElse e1 e2) env =
  orelse (eval e1 env) (\x ->
     eval e2 env $> (\fv ->
       apply fv [x]))

primitive fail (\ [x] -> failure x)
```

   The interpreter returns 2 in the `2 orelse fail(3)` case, because the second operand is never evaluated. We could change the interpreter to never shortcircuit orelse (that is, to always evaluate the second operand even if it is not called in the end).

3. `result x mem = Just (x, mem)`

```
xm $> f mem = case xm mem of
  | Just (v, mem') -> f v mem'
  | Nothing -> Nothing

new mem = let (a, mem') = fresh mem in Just (a, mem')
get a mem = Just (contents mem a, mem)
put a v mem = Just ((), update mem a v)

failure mem = Nothing
orelse xm ym mem = case xm mem of
  | Just (v, mem') -> Just (v, mem')
  | Nothing -> ym mem
```

4. `result x mem = (Just x, mem)`

```
xm $> f mem =
  let (v, mem') = xm mem in
  case v of
  | Just w -> f w mem'
  | Nothing -> (Nothing, mem')
```

```
new mem = let (a, mem') = fresh mem in (Just a, mem')
get a mem d= (Just (contents mem a), mem)
put a v mem = (Just (), update mem a v)

failure mem = (Nothing, mem)
orelse xm ym mem =
  let (v, mem') = xm mem in
  case v of
  | Just w  -> (Just w, mem')
  | Nothing -> ym mem'
```

5. M1 and M2 model different kinds of failure: A M1 failure is a lack of result and lack of side effect, whereas a M2 failure is a lack of result that might however have side effects.

   The practical difference is that when an expression performs side effects before ultimately failing the side effects will be undone in M1 but will remain in M2. M2 is therefore easier to implement more efficiently, because changes to the memory are always final and thus we can just do them without remembering their history.

   Example program that gives different results is:

```
x := new();
x := 1;
let y = x := 2 in fail();
!x
```

   which returns 1 in the M1 case and 2 in the M2 case.

6.
```
ref e =
  Let (Val "x" (Apply (Variable "new") []))
    (Sequence (Assign (Variable "x") e) (Variable "x"))

etranslate (Number n) = Number n
etranslate (Variable x) = Variable x
etranslate (Apply f es) =
  case maybe_find init_env f of
  | Just _  -> Apply f (map etranslate es)
  | Nothing -> Apply f (map (\e -> ref (etranslate e)) es)
etranslate (If a b c) =
  If (etranslate a) (etranslate b) (etranslate c)
etranslate (Lambda ids e) = Lambda ids (etranslate e)
etranslate (Let defs e) = Let (dtranslate defs) (etranslate e)
etranslate (Assign a b) = Assign a (etranslate b)
etranslate (Sequence a b) = Sequence (etranslate a) (etranslate b)
etranslate (While a b) = While (etranslate a) (etranslate b)

dtranslate (Val id e) = Val id (ref (etranslate e))
dtranslate (Rec id e) = Rec id (etranslate e)
```

7.
```
rec swap(a, b) = let t = new() in t := !a; a := !b; b := !t;;

etranslate (Address e) = ref (etranslate e)
```

```
etranslate (Contents e) = Apply (Variable "!") [(etranslate e)]

rec swap(a, b) = let t = a in a := b; b := t;;
```

`*&v translates to !(let x = new() in x := v; x) which is v`

`&*v` does not return the same value as `v` because a new memory location is created by the `&` operator.

8.
```
fibcps n k =
  if n <= 1 then
    k n
  else
    fib (n-1) (\a -> fib (n-2) (\b -> k (a + b)))
```

9.
```
result v $> f
=
(\k -> k v) $> f
=
(\k -> (\k -> k v) (\x -> f x k))
=
(\k -> (\x -> f x k) v)
=
(\k -> f v k)
=
f v

m $> result
=
(\k -> m (\x -> result x k))
=
(\k -> m (\x -> k x))
=
(\k -> m k)
=
m

(m $> f) $> g
=
(\k -> m (\x -> f x k)) $> g
=
(\k -> (\k -> m (\x -> f x k)) (\x -> g x k))
=
(\k -> m (\x -> (\k -> f x (\x -> g x k))))
=
(\k -> m (\x -> (f x $> g)) k)
=
(\k -> m (\x -> (\x -> (f x $> g)) x k)
=
m $> (\x -> (f x $> g))
```