# CSE 505 Assignment 3 Solution and Grading Guide

Andy Collins
acollins@cs.washington.edu

November 17, 2003

## Point distribution

Once again, homework 3 is graded out of 30 points. Overall, questions 1 and 3 were worth 4 points apiece, and questions 2 and 4 were worth 11. By parts, parts (1a), (1b), (2a), (2b), (2d), (3a), (3b), and (4b) were each worth 2 points, part (4a) was worth 1 point, part (2c) (the type safety proof) was worth 5 points, and parts (4c) and (4d) were worth 4 points each.

## 1 Evaluation order

### 1.1 Reverse engineering

Most people wanted to define functions and then call them separately, which was fine, although it was not strictly necessary and lead to some confusion about when exceptions might be raised depending on whether something was a function definition or a variable definition. It is also not strictly necessary, as the following will suffice:

```
exception Left
exception Right

let left_first_app() =
  try
    (raise Left) (raise Right)
  with Left  -> true
     | Right -> false
```

The "answer" is false, in the sense that in an application like `e1 e2` the argument `e2` is evaluated before the function `e1` (and of course both are evaluated before the application itself happens). In principle, these two issues (right-to-left evaluation of subexpressions prior to application and call-by-value semantics) are distinct, and we can construct tests that would point this out. Consider the following:

```
exception Left
exception Right

let left_first_app() =
  let my_fun () = raise Left; fun x -> () in
  try
    (my_fun ()) (raise Right) ; true
  with Left  -> true
     | Right -> false

let not_call_by_value() =
  let my_fun () = raise Left; fun x -> () in
```

```
try
  (my_fun) (raise Right) ; true
with Left  -> true
   | Right -> false
```

This is the same as the first solution, except that I've made a helper function. The only difference between `left_first_app` and `not_call_by_value` is that the former applies `my_fun` to `()` before applying that to `(raise Right)`. Both functions return false (try it), but for different reasons. Because `my_fun` will raise an exception when it is applied, `left_first_app` tells us that O'Caml really does evaluate the argument to a value before evaluating the function to a value, while `not_call_by_value` tells us only that O'Caml evaluates the argument to a value before applying the function, and does not tell us whether the sub-expression `(my_fun)` was evaluated to a value before the sub-expression `(raise Right)`. Because O'Caml is both right-to-left and call-by-value, this distinction is almost ambiguous but not quite, and I gave one point out of two for solutions that exhibited call-by-valueness rather than right-to-leftness. To put this into english rather than O'Caml, convince yourself that the following two questions are asking something different:

- In an expression of the form $e_1\ e_2$, is $e_2$ evaluated before $e_1$?
- In an expression of the form $e_1\ e_2$, is $e_2$ evaluated before being applied to $e_1$?

Some solutions ended up setting up expressions of the form $e_1\ e_2\ e_3$ (i.e. $e_1$ is a curried function of two arguments, and $e_2$ and $e_3$ are the arguments) and then showing that $e_3$ is evaluated before $e_2$. This is correct, and in the sense of the argument above is showing right-to-leftness.

### 1.2 Thunking to control evaluation order

Most solutions strongly resembled the following, which is a straightforward application of thunking:

```
let which_first f1 f2 b =
  if b
  then
    let v1 = f1() in
    let v2 = f2() in
    v1 v2
  else
    let v2 = f2() in
    let v1 = f1() in
    v1 v2
```

## 2 Closedness as type safety

### 2.1 Type system

Nearly everybody's solution looked like the following:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x{:}\mathsf{ok} \vdash e : \mathsf{ok}}{\Gamma \vdash \lambda x.\ e : \mathsf{ok}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ok} \qquad \Gamma \vdash e_2 : \mathsf{ok}}{\Gamma \vdash e_1\ e_2 : \mathsf{ok}}$$

using our normal definition of $\Gamma$ as a function mapping variables to their types. Of course this is a degenerate case because there is only one type, so we might equally well define $\Gamma$ as just a set of variables which are $\mathsf{ok}$, and rewrite the rules as

$$\frac{x \in \Gamma}{\Gamma \vdash x : \mathsf{ok}} \qquad \frac{\Gamma \cup \{x\} \vdash e : \mathsf{ok}}{\Gamma \vdash \lambda x.\ e : \mathsf{ok}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ok} \qquad \Gamma \vdash e_2 : \mathsf{ok}}{\Gamma \vdash e_1\ e_2 : \mathsf{ok}}$$

using set membership and union and avoiding the entire issue of defining $\Gamma$ as a function.

## 2.2 Type safety proof

Type Safety Theorem: If $\cdot \vdash e : \mathsf{ok}$ then $e$ diverges or $e \rightarrow^n v$ for an $n$ and $v$ such that $\cdot \vdash v : \mathsf{ok}$.

The proof relies on these lemmas, which we prove below:
Preservation Lemma: If $\cdot \vdash e : \mathsf{ok}$ and $e \rightarrow e'$, then $\cdot \vdash e' : \mathsf{ok}$.
Progress Lemma: If $\cdot \vdash e : \mathsf{ok}$, then $e$ is a value or there exists an $e'$ such that $e \rightarrow e'$.

Given these lemmas, we prove safety via this strengthened induction hypothesis: If $\cdot \vdash e : \mathsf{ok}$, then for all $n$ either $e \rightarrow^m v$ for some $m \leq n$ and $v$ such that $\cdot \vdash v : \mathsf{ok}$ or $e \rightarrow^n e'$, $\cdot \vdash e' : \mathsf{ok}$, and there exists an $e''$ such that $e' \rightarrow e''$. The proof is by induction on $n$. For $n = 0$, the Progress Lemma satisfies our hypothesis. For $n > 0$ induction ensures the hypothesis for $n - 1$. If $e \rightarrow^m v$ for some $m \leq (n - 1)$ and $v$ such that $\cdot \vdash v : \mathsf{ok}$, then we're done because $n - 1 \leq n$. Else $e \rightarrow^{n-1} e_1$ for some $e_1$ such that $\cdot \vdash e_1 : \mathsf{ok}$. By the Progress Lemma, there's some $e'$ such that $e_1 \rightarrow e'$, so $e \rightarrow^n e'$. By the Preservation Lemma $\cdot \vdash e' : \mathsf{ok}$.

We prove the Progress Lemma using this Canonical Forms Lemma: If $\cdot \vdash v : \mathsf{ok}$, then $v$ has the form $\lambda x.\ e$. Proof: Inspection of our rules shows there is no way a constant can type-check. The proof of Progress continues by induction on the structure of $e$:

- If $e$ is a value, the lemma is satisfied.
- If $e$ is a variable $x$, then $\cdot \vdash x : \mathsf{ok}$, which is impossible. So this case holds vacuously.
- If $e$ has the form $e_1\ e_2$, then inverting $\cdot \vdash e_1\ e_2 : \mathsf{ok}$ ensures $\cdot \vdash e_1 : \mathsf{ok}$ and $\cdot \vdash e_2 : \mathsf{ok}$. So if $e_1$ is not a value, induction ensures $e_1 \rightarrow e_1'$ for some $e_1'$. So we can derive $e_1\ e_2 \rightarrow e_1'\ e_2$. Else $e_1$ is some $v_1$. Then if $e_2$ is not a value, induction ensures $e_2 \rightarrow e_2'$ for some $e_2'$. So we can derive $v_1\ e_2 \rightarrow v_1\ e_2'$. Else $e_2$ is some $v_2$. Then the Canonical Forms Lemma ensures $v_1$ has the form $\lambda x.\ e_3$. So we can derive $(\lambda x.\ e_3)\ v_2 \rightarrow e_3[v_2/x]$.

The Preservation Lemma uses this Substitution Lemma: If $\Gamma, x{:}\mathsf{ok} \vdash e_1 : \mathsf{ok}$ and $\Gamma \vdash e_2 : \mathsf{ok}$, then $\Gamma \vdash e_1[e_2/x] : \mathsf{ok}$. The proof is later. The proof of Preservation continues by induction on the derivation of $\cdot \vdash e : \mathsf{ok}$, proceeding by cases on the bottom-most rule:

- The last rule cannot be for variables because $\cdot(x)$ is never defined.
- If the last rule concludes $\cdot \vdash \lambda x.\ e_1 : \mathsf{ok}$, then the lemma holds trivially because there is no $e'$ such that $\lambda x.\ e_1 \rightarrow e'$.
- If the last rule concludes $\cdot \vdash e_1\ e_2 : \mathsf{ok}$, then inversion ensures $\cdot \vdash e_1 : \mathsf{ok}$ and $\cdot \vdash e_2 : \mathsf{ok}$. The derivation of $e_1\ e_2 \rightarrow e'$ can have 1 of 3 forms:
  - $e'$ is $e_1'\ e_2$ because $e_1 \rightarrow e_1'$. Then $e_1 \rightarrow e_1'$, $\cdot \vdash e_1 : \mathsf{ok}$, and induction ensure $\cdot \vdash e_1' : \mathsf{ok}$. So with this and $\cdot \vdash e_2 : \mathsf{ok}$ we can derive $\cdot \vdash e_1'\ e_2 : \mathsf{ok}$.
  - $e'$ is $e_1\ e_2'$ because $e_2 \rightarrow e_2'$. Then $e_2 \rightarrow e_2'$, $\cdot \vdash e_2 : \mathsf{ok}$, and induction ensure $\cdot \vdash e_2' : \mathsf{ok}$. So with this and $\cdot \vdash e_1 : \mathsf{ok}$ we can derive $\cdot \vdash e_1\ e_2' : \mathsf{ok}$.
  - $e'$ is $e_3[v/x]$ because $e_1$ is $\lambda x.\ e_3$ and $e_2$ is $v$. Then inverting $\cdot \vdash \lambda x.\ e_3 : \mathsf{ok}$ ensures $\cdot, x{:}\mathsf{ok} \vdash e_3 : \mathsf{ok}$. This fact, $\cdot \vdash v : \mathsf{ok}$, and the Substitution Lemma ensure $\cdot \vdash e_3[v/x] : \mathsf{ok}$.

We prove the Substitution Lemma by induction on the derivation of $\Gamma, x{:}\mathsf{ok} \vdash e_1 : \mathsf{ok}$ using these technical lemmas: Exchange: If $\Gamma, x{:}\mathsf{ok}, y{:}\mathsf{ok} \vdash e : \mathsf{ok}$ then $\Gamma, y{:}\mathsf{ok}, x{:}\mathsf{ok} \vdash e : \mathsf{ok}$. Weakening: If $\Gamma \vdash e : \mathsf{ok}$ and $x$ does not occur in $e$, then $\Gamma, x{:}\mathsf{ok} \vdash e : \mathsf{ok}$. The We proceed on the bottom-most rule used to derive $\Gamma, x{:}\mathsf{ok} \vdash e_1 : \mathsf{ok}$:

- If $e_1$ is some $y$, then either $y = x$ or $y \neq x$. If $y = x$, then $e_1[e_2/x]$ is $e_2$ and we assumed $\Gamma \vdash e_2 : \mathsf{ok}$. If $y \neq x$, then $(\Gamma, x{:}\mathsf{ok})(y) = \Gamma(y)$, so $\Gamma \vdash y : \mathsf{ok}$. Since $y[e_2/x]$ is $y$, this is what we need.
- If $e_1$ is some $e_a\ e_b$, then inverting the typing derivation ensures $\Gamma, x{:}\mathsf{ok} \vdash e_a : \mathsf{ok}$ and $\Gamma, x{:}\mathsf{ok} \vdash e_b : \mathsf{ok}$. So by induction $\Gamma \vdash e_a[e_2/x] : \mathsf{ok}$ and $\Gamma \vdash e_b[e_2/x] : \mathsf{ok}$. So we can derive $\Gamma \vdash e_a[e_2/x]\ e_b[e_2/x] : \mathsf{ok}$. Since $e_a[e_2/x]\ e_b[e_2/x]$ is $(e_a\ e_b)[e_2/x]$, this is what we need.

- If $e_1$ is some $\lambda y.\ e_a$, then $\alpha$-conversion lets us assume $y$ is not $x$ does not occur in $e_2$. Inverting the typing derivation ensures $\Gamma, x{:}\mathsf{ok}, y{:}\mathsf{ok} \vdash e_a : \mathsf{ok}$. So the Exchange Lemma ensures $\Gamma, y{:}\mathsf{ok}, x{:}\mathsf{ok} \vdash e_a : \mathsf{ok}$. And the Weakening Lemma with $\Gamma \vdash e_2 : \mathsf{ok}$ ensures $\Gamma, y{:}\mathsf{ok} \vdash e_2 : \mathsf{ok}$. So induction ensures $\Gamma, y{:}\mathsf{ok} \vdash e_a[e_2/x] : \mathsf{ok}$. So we can derive $\Gamma \vdash \lambda y.\ (e_a[e_2/x]) : \mathsf{ok}$. Since $\lambda y.\ (e_a[e_2/x])$ is $(\lambda y.\ e_a)[e_2/x]$, this is what we need.

**Grading notes**

I was pretty happy with what I saw here overall. Not surprsingly the proofs largely followed those we did in class (or in the book), and most people were able to adapt the general simply-typed $\lambda$-calculus type safety proof to our system.

A number of people omitted the final step of proving type safety itself given preservation and progress. Given that this is almost painfully easy (and unchanged from the proof we did in class), I decided to let it slide.

For the most part, where people lost points it was for bottoming out into informal arguments too soon. I tried to judge these based on how painfully obvious the thing that ought to have been proven is in our type system. Thus I let almost any canonical forms argument go, as long as you pointed out at the appropriate place that you know if you have a value, and it typechecks, then is is a $\lambda$ an can be applied. On the other hand, several people argued the substitution lemma by saying only, "$e_1$ has no free variables, and $e_2$ has no free variables, so substituting one into the other can't introduce any free variables." This is clearly true at the "okay means no free variables" level of reasoning, but the real definition of the typesystem is the derivation rules, so that's the level where we need to argue the substitution lemma.

### 2.3 Stuck states

The following exactly defines the stuck states $s$: $s ::= c\ v \mid s\ e \mid v\ s$. The intuition here (which most people seemed to get) is that we can end up with something of the form $c\ v$ where we are trying to apply a constant, because constants are now indistinguishable from proper functions in the type system (we cannot get $x\ v$ because this will not typecheck). The recursive cases define the rest of the stuck states because our operational semantics are call-by-value, left-to-right. Thus we cannot reduce the sub-expression $e$ in an expression $s\ e$ because the stuck part is blocking us. Similarly we cannot reduce $v\ s$, even if $v$ is a function that we could apply.

Note that we were asking for the *stuck* states, not the *will-be-stuck* states. Thus $c\ e$ is not a stuck state because until we've reduced $e$ to a value we can still make progress, even if we are ulimately doomed.

## 3 Sum-type projection

### 3.1 (Non)Type-safety

Pretty much everyone used a counter-example essentially like the following. Consider $(\mathsf{inl}(0)).\mathsf{right}$. This expression is not a value but no rule in the operational semantics applies. This derivation shows it typechecks:

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 0 : \mathsf{int}}}{\cdot \vdash \mathsf{inl}(0) : \mathsf{int} + \mathsf{int}}}{\cdot \vdash (\mathsf{inl}(0)).\mathsf{right} : \mathsf{int}}$$

I did not require more than a casual explanation of why this is stuck given the operational semantics, but I did want to see at least an informal derivation to show that it does typecheck. I prefer this solution, with a concrete value 0 and concrete type $\mathsf{int} + \mathsf{int}$, but I accepted abstract solutions expressed using arbitrary values $v$ and types.

### 3.2 Case as a derived form

Most answers strongly resembed the following macro, where we replace $\mathsf{case}\ e_1\ x.e_2 \mid x.e_3$ with

$$\mathsf{if\ isleft}(e_1)\ \mathsf{then}\ (\lambda x.\ e_2)\ (e_1.\mathsf{left})\ \mathsf{else}\ (\lambda x.\ e_3)\ (e_1.\mathsf{right})$$

The only arguable complaint with this formulation is that it evaluates $e_1$ twice (but not three times, because of the way if expressions work), which some people did point out. This can be avoided by thunking:

$$(\lambda y.\ \mathsf{if\ isleft}(y)\ \mathsf{then}\ (\lambda x.\ e_2)\ (y.\mathsf{left})\ \mathsf{else}\ (\lambda x.\ e_3)\ (y.\mathsf{right}))\ e_1$$

provided we can generate a suitably "fresh" $y$.

Several people wanted to use our substitution syntax in place of the $(\lambda x.\ e)\ e'$ to substitute $e'$ for $x$ in $e$. I don't like this solution because it isn't really a derived form in the normal way of thinking about it, because we are doing most of the heavy lifting in the translation itself, rather than in the translated statement. In the end, however, I decided to let it go. For those of you who did this, the complaints that I may or may not have written on your paper are a function only of where it was in the pile relative to the evolution of my thinking on the issue. I went back and assigned the points after writing the comments.

## 4 Implementation

Overall, most people did very well here. I would say that the biggest gotcha was the way we defined the typing syntax for `rec` versus `fn`. Each has an argument, a type, and a body, but for `rec` the type is for the function as a whole (it has to be), while for `fn` the type is just for the argument, and the type of the function as a whole is derived from that and the body. This showed up in various places in a few solutions.

### 4.1 Formal typing rule

The following is the formal typing rule for `(rec f:t x. e)`. Nearly everyone got this right, or made what were probably just typos.

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{rec}\ f\ x.e : \tau_1 \to \tau_2}$$

### 4.2 Exponentiation

The following implements exponentiation (which as you discovered is mildly more complicated when you don't have multiplication as a primitive).

```
(rec exp : int->int->int base.
   (fn pow:int.
      (if pow > 0
       then
 (rec mult : int->int->int x.
    (fn y:int .
               (if x > 0
                then y + mult (x + -1) y
                else 0)))
 base (exp base (pow + -1))
       else 1)))
```

It didn't matter (beyond a little editing on my part) whether your file just defined the function (which was technically what we asked for and what most people did) or also applied it to a couple of test values. Everybody's solutions correctly computed my testcases of $2^3 = 8$ and $5^7 = 78125$.

### 4.3 Type-checker implementation

```
let mt_ctxt = []
let extend_ctxt ctxt x t = (x,t)::ctxt
let rec lookup_ctxt ctxt y =
  match ctxt with
    [] -> raise TypeError
  | (x,t)::tl -> if x=y then t else lookup_ctxt tl y

let rec typecheck ctxt e =
  match e with
    True  -> Bool
  | False -> Bool
  | If(e1,e2,e3) ->
      let t1 = typecheck ctxt e1 in
      let t2 = typecheck ctxt e2 in
      let t3 = typecheck ctxt e3 in
      if (t1=Bool) && (t2=t3)
      then t2
      else raise TypeError
  | Const i -> Int
  | Plus(e1,e2) ->
      let t1 = typecheck ctxt e1 in
      let t2 = typecheck ctxt e2 in
      if (t1=Int) && (t2=Int)
      then Int
      else raise TypeError
  | GreaterThan(e1,e2) ->
      let t1 = typecheck ctxt e1 in
      let t2 = typecheck ctxt e2 in
      if (t1=Int) && (t2=Int)
      then Bool
      else raise TypeError
  | Var(x) -> lookup_ctxt ctxt x
  | Fun(x,t,e) ->
      let t1 = typecheck (extend_ctxt ctxt x t) e in
      Arrow(t,t1)
  | Apply(e1,e2) ->
      let t1 = typecheck ctxt e1 in
      let t2 = typecheck ctxt e2 in
      (match t1 with
        Arrow(t3,t4) -> if t3=t2 then t4 else raise TypeError
      | _ -> raise TypeError)
  | Rec(f,t,x,e) ->
      (match t with
        Arrow(t1,t2) ->
          let c1 = extend_ctxt ctxt f t  in
          let c2 = extend_ctxt c1 x t1 in
          let t3 = typecheck c2 e in
          if t3=t2
          then t
          else raise TypeError
      | _ -> raise TypeError)
  | Closure(_,_) -> raise TypeError
```

## 4.4 Interpreter implementation

```
let rec interpret env e =
  match e with
    True -> e
  | False -> e
  | If(e1,e2,e3) ->
      let v1 = interpret env e1 in
      (match v1 with
        True -> interpret env e2
      | False -> interpret env e3
      | _ -> raise RunTimeError)
  | Const i -> e
  | Closure(_,_) -> e
  | Plus(e1,e2) ->
      let v1 = interpret env e1 in
      let v2 = interpret env e2 in
      (match (v1,v2) with
        (Const i1, Const i2) -> Const(i1+i2)
      | _ -> raise RunTimeError)
  | GreaterThan(e1,e2) ->
      let v1 = interpret env e1 in
      let v2 = interpret env e2 in
      (match (v1,v2) with
        (Const i1, Const i2) -> if i1 > i2 then True else False
      | _ -> raise RunTimeError)
  | Var(x) -> lookup_env env x
  | Fun(_,_,_) -> Closure(e,env)
  | Apply(e1,e2) ->
      let v1 = interpret env e1 in
      let v2 = interpret env e2 in
      (match v1 with
        Closure(Fun(x,_,e3),envc) -> interpret (extend_env envc x v2) e3
      | _ -> raise RunTimeError)
  | Rec(f,t,x,e) ->
      Closure(Fun(x,Ignore,e),extend_env_rec env f x e)
```

### Grading notes for (4c) and (4d)

Don't be surprised to see error messages in the run log for these parts; it was another mistake on my part, but I didn't bother correcting it on paper because everybody's code ran my two simple testcases just fine.