

**CS 502. Compilers  
Midterm Examination**

**October 14, 2008**

**Name:** \_\_\_\_\_

Answer each of the questions in the space provided. If you require additional space, use the blank pages attached at the end of the exam.

In the program fragments you write, you are free to use any syntactic abstractions you wish including `let` and `letrec`.

Score: \_\_\_\_\_

Maximum Score: 175

**Question 1** (35 points):

Consider the following expression:

```
fun f(p as (x,y,z)) =
  (case p of
    (_,false,true) => 1
  | (false,true,_) => 2
  | (_,_,false)   => 3
  | (_,_,true)    => 4)
```

(The notation "p as (x,y,z)" declares argument p to be a tuple whose first component is named x, second y, and third z.)

We would like to transform this program into an equivalent one that only uses if expressions.

1. (10 points) Consider a pattern-matching strategy that tests x first, then y, then z. Show the smallest program that enforces this order using conditionals instead of case.

```
fun f(x,y,z) =
  if x
  then if y
       then if z
            then 4
            else 3
       else if z
            then 1
            else 3
  else if y
       then 2
       else if z
            then 1
            else 3
```

2. (10 points) Suppose you employed a strategy that tested y first, instead of x. Show the smallest program that enforces this order using conditionals instead of case.

```
fun f(x,y,z) =
  if y
  then if x
       then if z
            then 4
            else 3
       else 2
  else if z
       then 1
       else 3
```

3. (15 points) Precisely describe a heuristic that would lead you to choose one strategy over the other.

A heuristic that considers the *necessity* of a term would work well: a subterm is *needed* when it is examined by all possible decision trees. In this example, x need not be examined if y is false. Similarly, z need not be examined if y is true. However, to eliminate the first two rows of the pattern, y's value is required. Since patterns are examined top-down, we can argue that y is more needed than either x or z.

**Question 2.** (10 points)

When a closure is closed over another a closure (i.e., a procedure reference appears free in another procedure), we can construct a closure representation strategy that flattens the inner closure entirely within the outer one (and transitively any closures it is closed over, etc.)

For example, in the following program fragment:

```
let val f = fn _ => ... a ... b ...
    val g = fn _ => ... b ...
    val h = fn _ => ... f ... a ... g ... c
    val i = fn _ => ... h ... d
in ...
end
```

- The closure record for **f** would contain slots for **a** and **b**;
- The closure record for **g** would contain a slot for **b**;
- The closure record for **h** would contain slots for **a,b,c**;
- The closure record for **i** would contain slots for **a,b,c,d**.

1. (5 points) What would the closure representation for these functions be using a traditional flat closure representation?

The closure record for **f** would contain **{a,b}**; the closure record for **g** would contain **{b}**; the closure record for **h** would contain **{a,c,f,g}**, and the closure record for **i** would contain **{h,d}**.

2. (5 points) What advantages and/or disadvantages does this scheme have compared to a flat closure strategy? Why?

This approach builds "flatter" closures than a typical flat closure strategy, and avoids some redundancy: since the inner closure record is flattened within the outer one, it may be possible to share closure records if we can prove the lifetime of the inner closure is no greater than the lifetime of the outer. The approach enables certain optimizations such as tuple flattening and inlining that would be otherwise difficult to express. If an inner closure's lifetime can be determined, the closure record built for the outer closure (that includes the inner closure's free variables) can be used as the closure record for the inner procedure.

The disadvantage with this scheme is that closure slots may be allocated that are not strictly necessary. For example, in the program:

```
let val f = fn () => a + b
    val g = fn () => f
in ...
end
```

a flat closure strategy would entail no allocation on calls to **g** but would necessitate a record allocation in the approach suggested here.

**Question 3.** (50 points)

Suppose a procedure  $p$  is always called with the same continuation  $k$  within some function  $q$ . Then, we can transform  $p$  into a continuation within  $q$  and eliminate the continuation argument from calls to  $p$ .

1. (10 points) Consider the CPS'ed program (where  $k$  and  $k'$  represent continuations, and the notation “`let cont k (x) = ...`” denotes a local continuation, and `gNonTail` denotes a non-tail call to function  $g$ ).

```
fun g (y, k) = k (y - 1)
fun f (b, k) = let cont k' (x) = k (x + 1)
               in if b then gNonTail (13, k') else gNonTail (15, k')
               end
```

Write down the original (non CPS-ed) version of this program.

```
fun g y = y - 1
fun f b = (if b then g 13 else g 15) + 1
```

2. (10 points) Observe that  $g$ 's continuation is always  $k'$ . Using this observation, write the CPS'ed program that transforms  $g$  into a continuation within  $f$ .

```
fun f (b,k) =
  let cont k'(x) = k (x + 1)
      fun g(y) = k'(y - 1)
  in if b then gTail(13) else gTail(15)
```

3. (20 points) Consider the following program that sums the elements of a vector  $v$ :

```
fun loop(v,i,s) =
  if i = length(v)
  then s
  else loop(v,i+1, s+ sub(v,i))

fun sum (v) = loop(v,0,0)
```

What is the effect of performing the transformation described above on this program? In particular, what properties of the program does this transformation reveal for subsequent optimizations?

The transformation moves `loop` inside `sum`, and treats it as a local continuation. The tail call to `loop` by `sum`, and the inner call to `loop` can now be expressed as `goto`'s (jumps) within `sum`, avoiding the overhead of a procedure call.

4. (10 points) Should this transformation be performed on CPS'ed programs before or after closure conversion? Make sure to explain your answer.

Since the transformation moves code (it takes a procedure and moves it into another procedure as a local continuation), free variable capture must be avoided. If the transformation is performed on source code, tracking where higher-order procedures are called is substantially more complicated. Expressing the transformation as a source-level rewrite further complicated by the difficulty in distinguishing continuations from procedures. Thus, it is probably best to perform the transformation on closure-converted (first-order) programs.

**Question 4.** (40 points)

A *useless variable* is one which contributes nothing to the final outcome of a computation. For example, in the following program, the second argument to `loop` does not contribute to the value of the computation, and can be eliminated.

```
let fun loop(a,bogus,j) =
  if (j > 100)
  then a * j
  else loop(f(a,j), bogus*2, j+1)
in loop(a,3,1)
end
```

One transformation to eliminate useless variables would turn the above program into:

```
let fun loop(a,j) =
  if (j > 100)
  then a * j
  else loop(f(a,j), j+1)
in loop(a,1)
end
```

1. (15 points) What is the result of applying useless variable elimination to the following ML program:

```
let fun f1(x,y) = x
    fun f2(x,y) = x+x
    fun f3(x,y) = y
    val g = if p(a,b) then f1 else f2
    val h = if q(a,b) then f1 else f3
in g(x,h)
end
```

```
let fun f1(x,y) = x
    fun f2(x,y) = x + x
    val g = if p(a,b) then f1 else f2
in g(x)
end
```

2. (25 points) Describe how to detect useless variables using polymorphic type inference. Hint: we can separate the optimization into two parts: in the first, we replace all sub-expressions that are useless with a special value (say "empty"); in the second part, the program is rewritten to eliminate such values. Use type-inference to identify those variables that could be replaced with `empty`. You will need to exploit polymorphism.

Perform type inference in a top-down manner. Consider the following expression:

$$((\text{fn } (x,y) \Rightarrow x) (1,2)) + 1$$

Let  $M$  be the application, and let  $N$  be the function  $\text{fn}(x,y) \Rightarrow x$ .  $M$  has type `int`. Thus,  $N$ 's type must be `int × 'a → int`, and the type of `(1,2)` is `int × 'a`. Ordinary type inference would unify `'a` and `int`, but useless variable elimination can simply instantiate `'a` to `empty` since `'a` is not subsequently used (i.e., unified) in the outer context.

**Question 5.** (40 points)

A coroutine is a computation that can be suspended and resumed. An executing coroutine can suspend itself, and resume another coroutine. Coroutines are declared like named procedures. Coroutines transfer control using the `throw` operation. The expression `throw(c,v)` causes control to revert to the suspended state of coroutine `c` supplying value `v`, and causes the state of the coroutine performing the throw to become suspended.

For example, in the program:

```
let val coroutine prod() = let val x = ref 0
                           fun loop () = let val _ = x := (!x) + 1
                                         val _ = throw(cons,x)
                                         in loop()
                                         end
                           in loop()
                           end
and val coroutine cons() = let fun loop () =
                           let val z = throw(prod,())
                           in (compute using z;
                               loop())
                           end
                           in loop()
                           end
in cons()
end
```

two coroutines are defined named `prod` and `cons` that serve as a producer and consumer of data. When the `cons` coroutine is invoked, it transfers control to `prod` which increments a counter, and returns control back to `cons`.

1. (10 points) Can you express the behavior of coroutines in terms of the source-level primitives available to you in Mini-ML? Why or why not?

Since we can transform Mini-ML programs to a CPS'ed version, we can express the behavior of `throw` in terms of capturing and storing a continuation; in Mini-ML, continuations would be represented as functions. To transfer control to another continuation, we would store the current continuation in a distinguished reference cell associated with the coroutine. In the absence of a CPS'ed representation, coroutines can be expressed within Mini-ML; in other words, we cannot write `throw` as a procedure within a non-CPS'ed Mini-ML program.

2. (15 points) Can you express the behavior of coroutines within a CPS intermediate representation? Why or why not?

Yes. See explanation above.

3. (15 points) Consider the addition of coroutines to your Mini-ML compiler. Is the current CPS signature (in terms of `Tail`, `Non-Tail` and `Goto` calls) sufficient to implement coroutines? Explain your answer.

Neither `Non-Tail` or `Tail` calls capture the behavior of `throw` which does not behave like a normal procedure call. The semantics of `Goto` come close, but note a `Goto` simply performs a jump, and does not restore any context information associated with the newly resumed coroutine. The continuations created and captured in a source-level transform are closed over a full context (environment), unlike the continuation invoked by `Goto`. To properly represent coroutines in the CPS core language, we would need a way of capturing the dynamic context of a continuation so that it can be properly restored on subsequent throws.