EECS 483 Midterm March 18, 2025

SOLUTIONS

1. **Frontend**

(a) The following program uses variables `x`,`y` and `z`. The occurrences of these in the program have all been labeled with superscripts. For each variable, identify if it is a *binding*, *bound* or *free* occurrence of the variable.

```
def main(x⁰):
  def g(y¹):
    def f(z²): z³ * x⁴ in
    y⁵ + z⁶
  in
  let x⁷ = (if x⁸ == 0: y⁹ else: z¹⁰)
  in x¹¹ + y¹²
```

variable occurrence	binding, bound or free
x^0	binding
y^1	binding
z^2	binding
z^3	bound
x^4	bound
y^5	bound
z^6	free
x^7	binding
x^8	bound
y^9	free
z^{10}	free
x^{11}	bound
y^{12}	free

(b) In Boa, the following program produced a compile-time error that the function `f` is called with the wrong number of arguments:

```
def main(x):
  def f(): x in
  def f(y): y + x in
  f()
```

However, it could instead be allowed, as the function call to `f` could be resolved to call the outer definition. This is a form of function *overloading* based on the *arity* of the function (the number of arguments that it takes). To resolve the program to one that does not use shadowing and supports this overloading we could rename the functions as follows:

```
def main(x):
  def f0(): x in
  def f1(y): y + x in
  f0()
```

Describe how you would adapt the frontend of the Boa compiler to allow for this form of overloading. In particular, what Rust datatype would you use to track the *environment* of functions that are in scope and how they should be resolved to use unique names?

Answer: The environment in the frontend used for functions can be adapted to use a `HashMap<(String,usize), FunName>`. That is, a mapping from pairs of a name and an arity to a resolved name. When a function `f` of arity `n` is defined, a fresh mangled name `g` is created and the environment is locally extended with a mapping `(f,n)` $\mapsto$ `g`. Then when resolving a function call, resolve based on both the name and the arity. If no function with that name and arity is found, raise a compile-time error, otherwise rename the function in the call to the one stored in the environment.

2. **Tail Calls**

(a) In the following snake program, the function calls have been labeled with superscripts. Identify which of these calls are tail calls and which are non-tail calls.

```
def main(x):
  def f(a):
    if a >= x:
      f^0(a - 1)
    else:
      g^1(a) - 1
  and def g(b):
    let x = h^2(7, b) in
    h^3(x,b)
  and def h(c, d):
    c + (d * d)
  in
  h^4(f^5(3), g^6(5))
```

function call	tail or non-tail
f^0	tail
g^1	non-tail
h^2	non-tail
h^3	tail
h^4	tail
f^5	non-tail
g^6	non-tail

(b) The following program is written in an imperative variant of the Snake language with support for mutable variables.

```
extern g()
def main(x):
  let mut i = 0;
  while i < 10 {
    i := i + g();
    if i < 0:
      i := 0
    else:
      break
  };
  return i
```

Complete the following Cobra program so that it has the same behavior as the imperative program.

```
extern g()
def main(x):
  let i = 0 in
  def loop(i):
    if i < 10:
      let i = i + g() in
      if i < 0:
        loop(0)
      else:
        i
    else:
      i
  in
  loop(i)
```

3. **Translation to SSA**

When writing SSA code by hand, you do not need to mangle the names of variables if they are already unique in the source program.

(a) Consider the following Cobra program which uses a new feature, a *match* expression, analogous to the feature of the same name in Rust

```
extern f(x)
extern g(x)
extern h(x)

def main(x):
  let y = (match x:
    | 0 => f(1)
    | 1 => g(0)
    | _ => h(x)
  ) in
  y + 1
```

The first case `0 => f(1)` should be taken if `x` is `0`, the second case if `x` is `1` and otherwise the last case should be executed.

Complete the SSA implementation of this program:

```
extern f(x)
extern g(x)
extern h(x)

main_fun(xf):
  br main(xf)
main(x):
  jn(y):
    r = y + 1
    ret r
  zero():
    a = f(1)
    br jn(a)
  one():
    b = g(0)
    br jn(b)
  catchall():
    c = h(x)
    br jn(c)
  notZero():
    e = x == 1
    cbr e one() catchall()
  d = x == 0
  cbr d zero() notZero()
```

(b) Translate the following Cobra program into SSA form. For full credit, be sure to add only the minimal number of arguments to any lifted functions.

```
def main(x):
  let y = x * 2 in
  let z = y + x in
  def f(a):
    a * y
  in
  f(f(z))
```

```
f_tail(y,a):
  r = a * y
  ret r
main(x):
  y = x * 2
  z = y + x
  tmp = f_fun(y, z)
  br f_tail(y, tmp)
fun f_fun(yf, af):
  br f_tail(yf, af)
fun main_fun(xf):
  br main(xf)
```

4. **SSA Well-formedness and Minimality**

(a) Identify 3 different ways in which the following SSA program is not well-formed:

```
main(i):
   f(x):
     ret (x + 1)
   y = f(i)
   z = y + 2
   br f()
```

Answer:

i. `x + 1` cannot be returned directly because it is not an immediate. It needs to be stored in a variable first.

ii. `y = f(i)` is invalid because f is a local sub-block, not a top-level function block

iii. `br f()` is not well formed because f takes one parameter but the branch has 0.

(b) The following SSA basic block is not in *minimal* SSA form in that it uses more basic block *arguments* than necessary. Write an equivalent SSA program that is minimal. Remember that rewriting to minimal SSA form may may require both *block sinking* (nesting a block inside of another) and *parameter dropping* (replacing basic block arguments with references to a variable). Note that here, unlike in the homework but as discussed in lecture, we allow conditional branch instructions to pass arguments.

```
main(x):
  f(m,n,o):
    p = m + o
    q = p * n
    r = p - 1
    s = q == 0
    cbr s g(q,m) f(m,n,r)
  g(t,u):
    v = t + u
    ret v
  f(x,5,6)
```

Answer:

```
main(x):
  m = x
  n = 5
  f(o):
    p = m + o
    q = p * n
    r = p - 1
    s = q == 0
    t = q
    u = m
    g():
      v = t + u
      ret v
    cbr g() f(r)
```

5. **Calling Conventions**

Consider the following unusual calling convention the *RisCy* calling convention. The RisCy calling convention is only for functions that take one argument as input and return one value as output. Additionally, the return address is passed in a register, rather than on the stack.

On entry into a function implementing the RisCy calling convention

(a) The return address is stored in `rdx`

(b) The argument is stored in `rsi`

(c) `rsp` is a stack pointer, memory at addresses at `rsp` and higher belong to the caller, whereas memory at addresses directly lower than `rsp` are free to be used by the callee.

(d) All registers besides `rdx`, `rsi` and `rax` are callee-save/non-volatile.

To return, the instruction pointer `rip` should be set to the return address that was stored in `rdx` upon function entry. The return value should be stored in `rax`.

Implement the following Cobra function in x86 assembly code using the RisCy calling convention. Here assume `g` is also implemented using the RisCy calling convention and has x86 label `g`. For simplicity, to move the address associated to a label `l` into a register `r`, you can simply write `mov r, l`, even though we mentioned in class that this does not work on all platforms.

```
def f(x): g(x) * x

f:
  push rdx ; save the return address
  push rsi ; save the argument
  mov rdx, kontinuation ; set the return address
  jmp g
kontinuation:
  ; rax stores g(x)
  pop rsi ; restore the argument
  pop rdx ; restore the return address
  imul rax, rsi
  jmp rdx
```