CSE 131 Final (Fa16)

Ranjit Jhala

December 9th, 2016

NAME ______________________________

SID ______________________________

- Where limits are given, write no more than the amount specified.
- Write your *full name* on the line at the top of this page.
- Do not separate pages.
- You may refer to a *double-sided cheat sheet*.
- Avoid seeing anyone else's work or allowing yours to be seen.
- Do not communicate with anyone but an exam proctor.
- If you have a question, raise your hand.

Right after the last lecture, your professor realized, to his horror, that we finished 131 without actually implementing a *real* language, which must, of course, have pointers, mutation and loops! In the final, we will implement these features so that, for example, we can compile **imperative** programs like:

```
def fib(n): let i     = (0, false)
              , vals  = (0, 1)
            in while (i[0] < n):
                ( let next = vals[0] + vals[1] in
                  i[0]    := i[0] + 1;
                  vals[0] := vals[1];
                  vals[1] := next
                );
                vals[1]
in fib(10)
```

Part I. Updating Tuples [35pts]

First, lets add support for **updating tuples**.

Concretely, that means that the program

```
let t = (10, 20)
  , a = (t[0] := t[0] + 2)
in
   (t[0], t[1])
```

should evaluate to `(12, 20)`, as the first line creates a tuple, and the second line executes an **tuple-update expression** to increments the `0`-th field by `2`.

Similarly,

```
let t = (10, 20)
  , a = t[1] := t[1] + 2
in
   (t[0], t[1])
```

updates the `1`-th field and hence, should evaluate to `(10, 22)`

Q1: Represent [3 pts]

Lets represent updates by extending the `Expr` type with a `SetItem` constructor:

```
data Expr a
  = ...
  | SetItem (Expr a) Field  (Expr a) a
```

The first `Expr` is the tuple being updated, the `Field` describes which part of the tuple is changed, and the second `Expr` is the value that it is changed to. As before, a `Field` is defined as:

```
data Field = Zero | One
```

Intuitively, the *update expression* `e1[fld] := e2` will be represented by `SetItem e1 fld e2 l` (where `l` is the tag meta-data as in your assignment.)

Fill in the blank below to get a Haskell represention of `t[0] := t[0] + 2`

```
ans1 :: Expr ()

ans1 = __________________________________________
```

Q2: ANF Example [5pts]

What is the ANF form of

```
t[0] := t[0] + 2
```

fill in the blanks below to get the ANF version of the above

```
let _________ =  ________________________________________

  , _________ =  ________________________________________
in
  _______________________________________________________
```

Q3: ANF [5pts]

Assume that in `e1[f] := e2` we want to *first* evaluate the tuple `e1` *and then* evaluate `e2`.

Next, fill in the blanks below to extend `anf` to handle the case for `SetItem`

```
anf :: Int -> Expr a -> (Int, AnfExpr a)

anf i (SetItem e1 fld e2 l) = ___________________________________
  where
        ___________________ = ___________________________________
```

HINT: Use `imms` and `stitch` described in the Appendix.

Q4: Type Inference [7pts]

Next, lets extend the type inference function to handle tuple updates.

Just like `Prim1`, `Prim2`, `If`, `Tuple` and `GetItem` we can simply treat *SetItem* as special kind of function call that takes two parameters, the source tuple, and the updated value.

```
ti :: (Located a) => TypeEnv -> Subst -> Expr a -> (Subst, Type)
ti env su (GetItem e f    l) = instApp (sourceSpan l) env su (fieldPoly  f) [e]
ti env su (SetItem e f e' l) = instApp (sourceSpan l) env su (updatePoly f) [e1, e2]
```

Complete the code for `ti` by completing the definition for `updatePoly`.

```
updatePoly      :: Field -> Poly

updatePoly Zero = Forall [____] ([_______, ________] :=> _________ )

updatePoly One  = Forall [____] ([_______, ________] :=> _________ )
```

The output value will be the same as that **being assigned**. That is,

- `(t[0] := false)` should evaluate to `false`, and
- `(t[0] := 12) + 1` should evaluate to `13`.

HINT: See how `ti` was implemented for `Prim1` in the Appendix.

Q5: Assemble [7pts]

Lets assume that as in the assignments:

1. Variables at stack position `i` have their value at address `[ebp - 4 * i]`,
2. The variable `t` lives at position `10` on the stack,
3. Tuple pointers are 8-byte aligned, and end with `001` (in binary.)
4. Tuples are layed out in the heap as in `egg-eater` so, a tuple of values `v0,...,vn` is layed out in the heap as below and user-constructed tuples (i.e. not closures) tuples have `n = 1` (i.e. just `v0, v1`).

```
  (4 bytes)   (4 bytes)  (4 bytes)              (4 bytes)
---------------------------------------------------------
|    n + 1  |      v0   |     v1     |    ...   |   vn  |
---------------------------------------------------------
```

Consider the expression

```
t[0] := 12
```

Assume that `t` lives at position `10` on the stack. Fill in the blanks below to get the assembly generated by the above expression. Recall that `(t[0] := 12) + 1` should evaluate to `13`, so at the end, `eax` should hold (the representation of) `12`.

```
_____________________________________     # __________________________

_____________________________________     # __________________________

_____________________________________     # __________________________

_____________________________________     # __________________________
```

Q6: Compile [8pts]

Inspired by the above, fill in the implementation that compiles an assignment expression.

```
compileEnv :: Env -> AExp -> [Instruction]
compileEnv env (SetItem v1 fld v2)

  = [ ________________________________________________

    , ________________________________________________

    , ________________________________________________

    , ________________________________________________
    ]

  where
    fOff :: Field -> Int
    fOff Zero = 4
    fOff One  = 8
```

HINT: You may want to use the helper `immArg` in the Appendix

Part II. Sequencing [30pts]

Next, lets implement sequencing, i.e. evaluating one expression **after** another, so, for example:

```
let t = (0, 0)
in
  t[0] := 2;
  t[1] := 6;
  t[0] + t[1]
```

should evaluate to `8`. That is `e1; e2` should evaluate `e1` and then `e2` and then evaluate the value that `e2` produced.

Q7: Represent [5pts]

Lets represent sequenced expressions as:

```
data Expr a
  = ...
  | Seq   (Expr a) (Expr a)   a
```

The first `Expr` is executed first, and *then* the second `Expr` should be executed.

Fill in the blanks below to show how the expression

```
  t[0] := 2;
  t[1] := 6;
  t[0] + t[1]
```

can be represented as an `Expr`

```
ans7 :: Expr ()

ans7 = Seq ( ____________________________________________ )

           (Seq ( ____________________________________________ )

                 ( ____________________________________________ ) ()) ()
```

Q8: ANF Example [5pts]

Fill in the blanks below to get an A-Normal Form of:

```
t[0] := t[0] + 2;
t[0] := t[0] + 6
```

HINT: Be careful about that ; !

```
--------------------------------------------------

--------------------------------------------------

--------------------------------------------------

--------------------------------------------------

--------------------------------------------------

--------------------------------------------------
```

Q9: ANF [5pts]

Fill in the blanks below to implement `anf` for sequences.

```
anf :: Int -> Expr a -> (Int, AnfExpr a)

anf i (Seq e1 e2 l)      = ____________________________________________

  where

    __________________   = ____________________________________________

    __________________   = ____________________________________________

    __________________   = ____________________________________________

    __________________   = ____________________________________________
```

HINT: You definitely don't need all the space given above. . .

Q10: Type Inference [5pts]

Next, lets extend the type inference function to handle sequences.

Again, we can do so by treating `Seq` as a special kind of function call that "takes" two parameters, the first and second expressions, and *"returns" the second expression's value as the result.*

```
ti :: (Located a) => TypeEnv -> Subst -> Expr a -> (Subst, Type)
ti env su (Seq e1 e2 l) = instApp (sourceSpan l) env su seqPoly [e1, e2]
```

Complete the above implementation by filling in the definition of `SeqPoly`

```
seqPoly :: Poly

seqPoly = Forall [_______] ([ _______ , ________ ] :=> _________)
```

Q11: Assemble [5pts]

Fill in the blanks below to show the assembly that should be generated for

```
t[0] := 12; t[1] := add1(t[0])
```

Again, assume that `t` lives at position `10` on the stack.

```
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
_____________________________________     # __________________________
```

Q12: Compile [5pts]

Next, fill in the implementation of `compileEnv` for sequences.

```
compileEnv :: Env -> AExp -> [Instruction]
compileEnv env (Seq e1 e2 _)

  = ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________
```

HINT: You definitely don't need all the space given above…

Part III. While [40pts]

Finally, lets add support for `while` loops.

Concretely that means that the expression:

```
let t = (0, 1) in
while (t[0] < 10):
  ( t[0] := t[0] + 1
  ; t[1] := t[1] * 2
  );
t[1]
```

should evaluate to `1024`.

Q13: Represent [4pts]

Lets represent `while` loops by extending `Expr` as:

```
data Expr a
  = ...
  | While (Expr a) (Expr a)  a
```

The first `Expr` is the loop "condition" and the latter is the loop "body".

Fill in the blanks below to show how the expression

```
while (t[0] < 10):
  t[0] := add1(t[0])
```

can be represented as an `Expr`

```
ans13 :: Expr ()

ans13 = While ( ____________________________________________ )

              ( ____________________________________________ ) ()
```

Q14: ANF Example [6pts]

Consider the expression:

```
while (t[0] < 10):
  t[0] := add1(t[0])
```

Fill in the blanks below to get an A-Normal Form representation of the above.

```
---------------------------------------------------------------------

---------------------------------------------------------------------

---------------------------------------------------------------------

---------------------------------------------------------------------
```

Q15: ANF [5pts]

Drawing inspiration from the above, fill in the blanks below to implement `anf` for `while`

```
anf :: Int -> Expr a -> (Int, AnfExpr a)

anf i (While e1 e2 l)   = ____________________________________________

  where

    __________________  = ____________________________________________

    __________________  = ____________________________________________

    __________________  = ____________________________________________

    __________________  = ____________________________________________
```

HINT Just because I gave you four lines doesn't mean you need to use them.

Q16: Type Inference [7pts]

Stop me if you've heard this before: we can do type inference for while using a special function call that takes two *input* parameters: the "condition" and "body" expressions, and returns an *output* that is . . . ? I don't know! Can you help me complete `ti` for `While` by filling in the definition of `whilePoly` ?

```
ti :: (Located a) => TypeEnv -> Subst -> Expr a -> (Subst, Type)
ti env su (While e1 e2 l) = instApp (sourceSpan l) env su whilePoly [e1, e2]

whilePoly :: Poly
whilePoly = Forall [_______] ([ __________ , ____________ ] :=> ______________)
```

Q17: Assemble [8pts]

Suppose that

- `instrs1` is a list of instructions evaluating `t[0] < 10`, at the end of which `eax` holds the representation for `true` if `t[0]` was indeed less than `10` (and `false` otherwise)
- `instrs2` is a list of instructions evaluating `t[0] := add1(t[0])`,

Use `instrs1` and `instrs2`, to obtain the assembly corresponding to:

```
while (t[0] < 10):
  t[0] := add1(t[0])
```

HINT You may write `repr True` and `repr False` if you need to use the (representations) of `true` and `false` in the assembly below.

Q18: Compile [10pts]

Drawing inspiration from the above, complete the implementation of `compileEnv` for `while` loops. You can assume that `tagLabel` generates assembly control flow labels:

```
tagLabel :: Int -> Tag -> Label
```

and that `l` has type `Tag`, a unique value for each sub-expression.

```
compileEnv :: Env -> AExp -> [Instruction]
compileEnv env (While e1 e2 l)

  = ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________
  where
    labelBegin :: Label
    labelBegin = tagLabel 0 l

    labelEnd   :: Label
    labelEnd   = tagLabel 1 l
```

HINT: See the appendix for a list of assembly instructions.

Part IV. Recursion via Mutation [75pts]

Lets rewind and assume we **do not** have `while`-loops in the language. It turns out that with tuple assignment (i.e. *mutation*), we can get rid of recursive functions, i.e. we can **implement** recursive `def` functions by combining mutable tuples and plain old `lambda` functions.

Q19: Factorial without Recursion [35pts]

Consider the following lambda-expression that is a wrapper around a recursive definition of `factorial`

```
lambda(m):
  def factorial(n):
    if (n < 1): 1 else: n * factorial(n - 1)
  in
    factorial(m)
```

Write an *equivalent* `lambda`-expression that

- *does not* use `def` i.e. does not use recursive functions, but
- *does* use tuple assignment (and `lambda`)

to have the the same behavior as the original, i.e. computes the `factorial` function.

```
lambda(m):

  let __________  = ____________________________________________  in

    ( __________ := ____________________________________________ ) ;

    ______ (m)
```

HINT: Assume that the above (non-recursive) program need **not** be type-checked.

Q20: Translating Recursion to Mutation [40pts]

Recall from `FDL` that internally, `def` was represented as a *named* function `Fun`, so the (recursive version) of the above code is internally represented as an `Expr` that looks something like:

```
(Lam ["m"]
  (Let "factorial"
      (Fun "factorial" ["n"] ( If (n < 1) 1 (n * App "factorial" [n - 1])))
      (App "factorial" ["m"])))
```

Complete the implementation of `noFun` that systematically performs the above translation; i.e. which would convert `Expr` that contain *named* (recursive) functions `Fun f xs e` into **equivalent programs** that contain *no occurrences* of `Fun` and hence, no explicitly recursive functions.

```
noFun :: Expr a -> Expr a
noFun e = go e
  where
    go (Number n l       ) = __________________________________________

    go (Id  x l          ) = __________________________________________

    go (Prim2 o e1 e2 l  ) = __________________________________________

    go (If b e1 e2 l     ) = __________________________________________

    go (Tuple e1 e2 l    ) = __________________________________________

    go (GetItem e1 f l   ) = __________________________________________

    go (App e es l       ) = __________________________________________

    go (Lam xs e l       ) = __________________________________________

    go (Seq e1 e2 l      ) = __________________________________________

    go (SetItem e1 f e2 l) = __________________________________________

    go (Fun f xs e l     ) = __________________________________________

                             __________________________________________

                             __________________________________________
```

HINT: The interesting stuff happens in the case for `Fun`. You may assume that you have at your disposal, a library function such that `substitute e (x, e')` replaces all "free" occurrences of `x` inside `e` with `e'`.

```
substitute :: Expr a -> (Id a, Expr a) -> Expr a
```

Appendix

Type Definitions

```haskell
-- | ANF Expressions labeled with a unique Tag
type AExp = Expr Tag

-- | Representing Expressions
data Expr a
  = ...
  | Number  Integer                     a
  | Id      Id                          a
  | Prim2   Prim2    (Expr a)  (Expr a) a
  | Tuple   (Expr a) (Expr a)           a
  | GetItem (Expr a) Field              a
  | SetItem (Expr a) Field     (Expr a) a -- NEW

-- | Fields
data Field = Zero | One

-- | Primitive Operations
data Prim2 = ... | Plus

-- | Polymorphic Types
data Poly =  Forall [TVar] Type -- forall a. a -> a -> Bool

data Type =  TVar TVar           -- a
          |  TInt                -- Int
          |  TBool               -- Bool
          |  [Type] :=> Type     -- (t1,...,tn) => t2
          |  TPair Type Type     -- (t0, t1)

-- | Machine (x86) Instructions
data Instruction
  = IMov    Arg   Arg
  | IAdd    Arg   Arg
  | ISub    Arg   Arg
  | IMul    Arg   Arg
  | IShr    Arg   Arg
  | ISar    Arg   Arg
  | IShl    Arg   Arg
  | IAnd    Arg   Arg
  | IOr     Arg   Arg
  | IXor    Arg   Arg
  | ILabel  Label
  | IPush   Arg
```

```haskell
  | IPop    Arg
  | ICmp    Arg   Arg
  | IJe     Label
  | IJne    Label
  | IJg     Label
  | IJge    Label
  | IJl     Label
  | IJo     Label
  | IJmp    Label
  | ICall   Arg
  | IRet

-- | Machine Arguments
data Arg
  = Const     Int
  | HexConst  Int
  | Reg            Reg
  | RegOffset Nat  Reg
  | RegIndex  Reg  Reg
  | Sized     Size Arg
  | CodePtr   Label
  | GlobVar   Text

-- | Registers
data Reg
  = EAX | EBX | ECX
  | ESP | EBP | ESI
```

Functions for ANF Conversion

```haskell
-- | `imms i es` takes as input a "start" counter `i` and expressions `es`, and
--   and returns an output `(i', bs, es')` where
--   * `i'` is the output counter (i.e. i'- i) anf-variables were generated
--   * `bs` are the temporary binders needed to convert `es` to immediate vals
--   * `es'` are the immediate values  equivalent to es

imms :: Int -> [Expr a] -> (Int, Binds a, [ImmExpr a])

-- | `stitch bs e` takes a "context" `bs` which is a list of temp-vars and their
--   definitions, and an expression `e` that uses the temp-vars in `bs` and glues
--   them together into a `Let` expression.

stitch :: Binds a -> AnfExpr a -> AnfExpr a
```

Functions for Type Inference

```haskell
ti :: (Located a) => TypeEnv -> Subst -> Expr a -> (Subst, Type)
ti env su (Prim1 p e l)    = instApp (sourceSpan l) env su (prim1Poly p) [e]

prim1Poly :: Prim1 -> Poly
prim1Poly Add1  = Forall [   ] ([TInt] :=> TInt)
prim1Poly Sub1  = Forall [   ] ([TInt] :=> TInt)
prim1Poly Print = Forall ["a"] ([ "a"] :=> "a")
```

Functions for Compiling

```haskell
-- | immArg converts an immediate value, i.e. a Number, Boolean or Id
--   (on the stack) into an Arg
immArg :: Env -> ImmExpr a -> Arg
```