CSE 110A: Winter 2020

Fundamentals of Compiler Design I

Branches and Binary Operators

Owen Arden
UC Santa Cruz

Based on course materials developed by Ranjit Jhala

BOA: Branches and Binary Operators

Next, lets add

- Branches (`if`-expressions)
- Binary Operators (`+`, `-`, etc.)

In the process of doing so, we will learn about

- **Intermediate Forms**
- **Normalization**

Branches

Let's start first with branches (conditionals).

We will stick to our recipe of:

1. Build intuition with **examples**,
2. Model problem with **types**,
3. Implement with **type-transforming-functions**,
4. Validate with **tests**.

```
data Expr = ENum                  -- 12
          | EPrim1 Op1  Expr      -- add1(e)
          | EVar   Id             -- x
          | ELet   Id   Expr Expr -- let x = e1 in e2
          | EIf    Expr Expr Expr
```

Examples

First, let's look at some examples of what we mean by branches.

- For now, lets treat `0` as "false" and non-zero as "true"

Example: `If1`

```
if 10:
  22
else:
  sub1(0)
```

- Since `10` is *not* `0` we evaluate the "then" case to get `22`

Examples

First, let's look at some examples of what we mean by branches.

- For now, lets treat `0` as "false" and non-zero as "true"

Example: `If2`

```
if sub(1):
  22
else:
  sub1(0)
```

- Since `sub(1)` *is* `0` we evaluate the "else" case to get `-1`

Control Flow in Assembly

To compile branches, we will use:

- **labels** of the form

```
our_code_label:
  …
```

"landmarks" from which execution (control-flow) can be started, or to which it can be diverted,

Control Flow in Assembly

To compile branches, we will use:

- **comparisons** of the form

```
cmp a1, a2
```

- Perform a (numeric) **comparison** between the values `a1` and `a2`, and
- Store the result in a special **processor flag**,

Control Flow in Assembly

To compile branches, we will use:

- **Jump** operations of the form

```
jmp LABEL     # jump unconditionally (i.e. always)
je  LABEL     # jump if previous comparison result was EQUAL
jne LABEL     # jump if previous comparison result was NOT-EQUAL
```

- Use the result of the **flag** set by the most recent `cmp`
- To *continue execution* from the given `LABEL`

Strategy

To compile an expression of the form

```
if eCond:
  eThen
else:
  eElse
```

We will:

1. Compile `eCond`
2. Compare the result (in `eax`) against `0`
3. Jump if the result *is zero* to a **special** `"IfFalse"` label
   - At which we will evaluate `eElse`,
   - Ending with a special `"IfExit"` label.
4. (Otherwise) continue to evaluate `eTrue`
   - And then jump (unconditionally) to the `"IfExit"` label.

Example: if1

```
if 10:
  22
else:
  sub1(0)
```

→

```
  mov eax, 10
  cmp eax, 0
  je  if_false
if_true:
  mov eax, 22
  jmp if_exit
if_false:
  mov eax, 0
  sub eax, 1
if_exit:
```


Example: if2

```
if sub(1):
  22
else:
  sub1(0)
```

→

```
  mov eax, 1
  sub eax, 1
  cmp eax, 0
  je  if_false
if_true:
  mov eax, 22
  jmp if_exit
if_false:
  mov eax, 0
  sub eax, 1
if_exit:
```


Example: if3

```
let x = if 10:
          22
        else:
          0
in
  if x:
    55
  else:
    999
```

→

```
  mov eax, 10
  cmp eax, 0
  je  if_false
  mov eax, 22
  jmp if_exit
if_false:
  mov eax, 0
if_exit:
  mov [esp - 4*1], eax
  mov eax, [esp - 4*1]
  cmp eax, 0
  je  if_false
  mov eax, 55
  jmp if_exit
if_false:
  mov eax, 999
if_exit:
```

Example: if3

Oops, **cannot reuse labels** across if-expressions!

- Can't use same label in two places (invalid assembly)

```
let x = if 10:
          22
        else:
          0
in
  if x:
    55
  else:
    999
```

```
mov eax, 10
cmp eax, 0
je  if_false
mov eax, 22
jmp if_exit
if_false:
mov eax, 0
if_exit:
mov [esp - 4*1], eax
mov eax, [esp - 4*1]
cmp eax, 0
je  if_false
mov eax, 55
jmp if_exit
if_false:
mov eax, 999
if_exit:
```


x`

Oops, need **distinct labels** for each branch!

- Require **distinct tags** for each if-else expression

```
let x = if 10:
        1 22
        else:
          0
in
  if x:
  2 55
  else:
    999
```

```
mov eax, 10
cmp eax, 0
je  if_1_false
mov eax, 22
jmp if_1_exit
if_1_false:
mov eax, 0
if_1_exit:
mov [esp - 4*1], eax
mov eax, [esp - 4*1]
cmp eax, 0
je  if_2_false
mov eax, 55
jmp if_2_exit
if_2_false:
mov eax, 999
if_2_exit:
```


Types: Source

Lets modify the *Source Expression*

```
data Expr a
  = Number Int                       a
  | Add1   (Expr a)                  a
  | Sub1   (Expr a)                  a
  | Let    Id (Expr a) (Expr a)      a
  | Var    Id                        a
  | If     (Expr a) (Expr a) (Expr a) a
```

- Add if-else expressions and
- Add **tags** of type a for each sub-expression
  - Tags are polymorphic a so we can have *different types* of tags
  - e.g. Source-Position information for error messages

Types: Source

Lets modify the *Source Expression*

```
data Expr a
  = Number Int                        a
  | Add1   (Expr a)                   a
  | Sub1   (Expr a)                   a
  | Let    Id (Expr a) (Expr a)       a
  | Var    Id                         a
  | If     (Expr a) (Expr a) (Expr a) a
```

- Add `if-else` expressions and
- Add **tags** of type `a` for each sub-expression
  - Tags are polymorphic `a` so we can have *different types* of tags
  - e.g. Source-Position information for error messages

Types: Source

Let's define a name for `Tag` (just integers).

```
type Tag   = Int
```

We will now use:

```
type BareE = Expr ()      -- AST after parsing
type TagE  = Expr Tag     -- AST with distinct tags
```


Types: Assembly

Now, lets extend the *Assembly* with labels, comparisons and jumps:

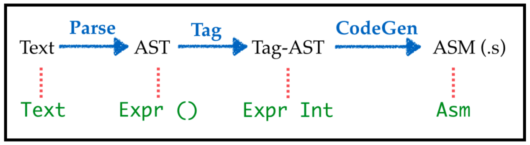
```
data Label
  = BranchFalse Tag
  | BranchExit  Tag

data Instruction
  = ...
  | ICmp   Arg   Arg   -- Compare two arguments
  | ILabel Label       -- Create a label
  | IJmp   Label       -- Jump always
  | IJe    Label       -- Jump if equal
  | IJne   Label       -- Jump if not-equal
```

Transforms

We can't expect *programmer* to put in tags (yuck.)

- Lets squeeze in a `tagging` transform into our pipeline



Transforms: Parse

Just as before, but now puts a dummy `()` into each position

```
λ> let parseStr s = fmap (const ()) (parse "" s)

λ> let e = parseStr "if 1: 22 else: 33"

λ> e
If (Number 1 ()) (Number 22 ()) (Number 33 ()) ()

λ> label e
If (Number 1 ((),0)) (Number 22 ((),1)) (Number 33 ((),2)) ((),3)
```


Transforms: Tag

The key work is done by `doTag i e`

1. Recursively walk over the `BareE` named `e` starting tagging at counter `i`
2. Return a pair `(i', e')` of *updated counter* `i'` and tagged expr `e'`

Transforms: Tag

We can now tag the whole program by

- Calling `doTag` with the initial counter (e.g. `0`),
- Throwing away the final counter.

```
tag :: BareE -> TagE
tag e = e'  where  (_, e') = doTag 0 e
```


Transforms: CodeGen

Now that we have the tags we lets implement our compilation strategy

```
compile env (If eCond eTrue eFalse i)
  =   compile env eCond ++        -- compile `eCond`
    [ ICmp (Reg EAX) (Const 0)    -- compare result to 0
    , IJe (BranchFalse i)         -- if-zero then jump to 'False'-block
    ]
   ++ compile env eTrue  ++       -- code for `True`-block
    [ IJmp   lExit      ]         -- jump to exit (don't execute `False`)
   ++
      ILabel (BranchFalse i)      -- start of `False`-block
   : compile env eFalse ++        -- code for `False`-block
    [ ILabel (BranchExit i) ]     -- exit
```


Recap: Branches

- `Tag` each sub-expression,
- Use tag to generate control-flow labels implementing branch.

Lesson: Tagged program representation simplifies compilation…

- Next: another example of how intermediate representations help.

Binary Operations

Compiling Binary Operations

You know the drill.

1. Build intuition with **examples**,
2. Model problem with **types**,
3. Implement with **type-transforming-functions**,
4. Validate with **tests**.

Let's look at some expressions and figure out how they would get compiled.

- Recall: We want the result to be in `eax` after the instructions finish.

Compiling Binary Operations

How to compile `n1 * n2`

```
mov eax, n1
mul eax, n2
```

Example: Bin1

Let's start with some easy ones. The source:

```
2 + 3  ──────▶  mov eax, 2
                add eax, 3
```

Strategy: Given `n1 + n2`

- Move `n1` into `eax`,
- Add `n2` to `eax`.

Example: Bin2

```
let x = 10          -- position 1 on stack
  , y = 20          -- position 2 on stack
  , z = 30          -- position 3 on stack
in
   x + (y * z)
```

```
let x = 10          -- position 1 on stack
  , y = 20          -- position 2 on stack
  , z = 30          -- position 3 on stack
  , tmp = y * z
in
   x + tmp
```


Example: Bin2

```
mov eax, 10
mov [ebp - 4*1], eax     ; put x on stack
mov eax, 20
mov [ebp - 4*2], eax     ; put y on stack
mov eax, 30
mov [ebp - 4*3], eax     ; put z on stack

mov eax, [ebp - 4*2]     ; grab y
mul eax, [ebp - 4*3]     ; mul by z
mov [ebp - 4*4], eax     ; put tmp on stack

mov eax, [ebp - 4*1]     ; grab x
add eax, [ebp - 4*4]
```

Example: Bin2

What if the first operand is a variable?

Simple, just copy the variable off the stack into `eax`

```
let x = 12              mov eax, 12
in            ---->     mov [esp - 4], eax
    x + 10              mov eax, [esp - 4]
                        add eax, 10
```

Strategy: Given `x + n`

- Move `x` (from stack) into `eax`,
- Add `n` to `eax`.

Example: Bin3

Same thing works if the second operand is a variable.

```
                        mov eax, 12
let x = 12              mov [esp - 4], eax
  , y = 18    ---->     mov eax, 18
in                      mov [esp - 8], eax
    x + y               mov eax, [esp - 4]
                        add eax, [esp - 8]
```

Strategy: Given `x + n`

- Move `x` (from stack) into `eax`,
- Add `n` to `eax`.

Second Operand is Constant

In general, to compile `e + n` we can do

```
   compile e
++                -- result of e is in eax
   [add eax, n]
```

Example: Bin4

But what if we have *nested* expressions

`(1 + 2) * (3 + 4)`

- Can compile `1 + 2` with result in `eax` ...
- .. but then need to *reuse* `eax` for `3 + 4`

Need to **save** `1 + 2` somewhere!

Idea How about use *another* register for `3 + 4`?

- But then what about `(1 + 2) * (3 + 4) * (5 + 6)` ?
- In general, may need to *save* more sub-expressions than we have registers.

Idea: Immediate Expressions

Why were `1 + 2` and `x + y` so easy to compile but `(1 + 2) * (3 + 4)` not?

Because `1` and `x` are **immediate expressions**

Their values don't require any computation!

- Either a **constant**, or,
- **variable** whose value is on the stack.

Idea: Administrative Normal Form (ANF)

An expression is in **Administrative Normal Form (ANF)** if **all primitive operations** have **immediate** arguments

Primitive Operations: Those whose values we *need* for computation to proceed.

- `v1 + v2`
- `v1 - v2`
- `v1 * v2`

Conversion to ANF

However, note the following variant *is* in ANF

```
let t1 = 1 + 2
  , t2 = 3 + 4
in
    t1 * t2
```

How can we compile the above code?

Binary Operations: Strategy

We can convert *any* expression to ANF by adding “temporary” variables for sub-expressions

Compiler Pipeline with ANF

- **Step 1:** Compiling ANF into Assembly
- **Step 2:** Converting Expressions into ANF

Types: Source

Lets add binary primitive operators

```
data Prim2
  = Plus | Minus | Times
```

and use them to extend the source language:

```
data Expr a
  = ...
  | Prim2 Prim2  (Expr a) (Expr a) a
```

So, for example, `2 + 3` would be parsed as:

```
Prim2 Plus (Number 2 ()) (Number 3 ()) ()
```

Types: Assembly

Need to add X86 instructions for primitive arithmetic:

```
data Instruction
  = ...
  | IAdd Arg Arg
  | ISub Arg Arg
  | IMul Arg Arg
```


Types: ANF

We *can* define a separate type for ANF (try it!)

… but …

super tedious as it requires duplicating a bunch of code.

Instead, lets write a *function* that describes **immediate expressions**

```
isImm :: Expr a -> Bool
isImm (Number _ _) = True
isImm (Var    _ _) = True
isImm _            = False
```

We can now think of **immediate** expressions as:

The subset of `Expr` such that `isImm` returns `True`

QUIZ

Similarly, lets write a function that describes **ANF** expressions

```
isAnf :: Expr a -> Bool
isAnf (Number  _      _) = True
isAnf (Var     _      _) = True
isAnf (Prim2 _ e1 e2 _) = _1
isAnf (If e1 e2 e3    _) = _2
isAnf (Let x e1 e2    _) = _3
```

What should we fill in for `_1`?

```
{- A -} isAnf e1
{- B -} isAnf e2
{- C -} isAnf e1 && isAnf e2
{- D -} isImm e1 && isImm e2
{- E -} isImm e2
```

QUIZ

Similarly, lets write a function that describes **ANF** expressions

```haskell
isAnf :: Expr a -> Bool
isAnf (Number  _       _) = True
isAnf (Var     _       _) = True
isAnf (Prim2 _ e1 e2 _) = _1
isAnf (If e1 e2 e3    _) = _2
isAnf (Let x e1 e2    _) = _3
```

What should we fill in for `_2`?

```haskell
{- A -} isAnf e1
{- B -} isImm e1
{- C -} True
{- D -} False
```


ANF

We can now think of **ANF** expressions as:

The subset of `Expr` such that `isAnf` returns `True`

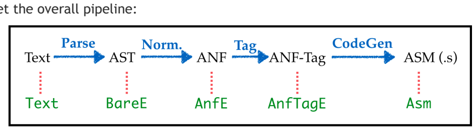
Use the above function to **test** our ANF conversion.

Types & Strategy

Writing the type aliases:

```haskell
type BareE   = Expr ()
type AnfE    = Expr ()  -- such that isAnf is True
type AnfTagE = Expr Tag -- such that isAnf is True
type ImmTagE = Expr Tag -- such that isImm is True
```

we get the overall pipeline:



Transforms: Compiling AnfTagE to Asm

The compilation from ANF is easy, lets recall our examples and strategy:

Strategy: Given `v1 + v2` (where `v1` and `v2` are **immediate expressions**)

- Move `v1` into `eax`,
- Add `v2` to `eax`.

Transforms: Compiling AnfTagE to Asm

```
compile :: Env -> TagE -> Asm
compile env (Prim2 o v1 v2)
  = [ IMov      (Reg EAX) (immArg env v1)
    , (prim2 o) (Reg EAX) (immArg env v2)
    ]
```

where we have a helper to find the `Asm` variant of a `Prim2` operation

```
prim2 :: Prim2 -> Arg -> Arg -> Instruction
prim2 Plus  = IAdd
prim2 Minus = ISub
prim2 Times = IMul
```

and another to convert an *immediate expression* to an x86 argument:

```
immArg :: Env -> ImmTag -> Arg
immArg _   (Number n _) = Const n
immArg env (Var    x _) = RegOffset ESP i
  where
    i    = fromMaybe err (lookup x env)
    err  = error (printf "Error: Variable '%s' is unbound" x)
```


Transforms: Compiling Bare to Anf

Next lets focus on **A-Normalization** i.e. transforming expressions into ANF

We can fill in the base cases easily

```
anf (Number n)      = Number n
anf (Var x)         = Var x
```

Interesting cases are the binary operations

A-Normalization

Example Anf-1: Left operand is not immediate

```
(1 + 2) + 3   ──────▶   let t1 = 1 + 2
                        in
                          t1 + 3
```

Key Idea: Helper Function

```
imm :: BareE -> ([(Id, AnfE)], ImmE)
```

`imm e` returns `([(t1, a1),...,(tn, an)], v)` where

- `ti, ai` are new temporary variables bound to ANF exprs,
- `v` is an **immediate value** (either a constant or variable)

Such that `e` is *equivalent to*

```
let t1 = a1
  , ...
  , tn = an
in
   v
```


A-Normalization

Example Anf-2: Left operand is not *internally* immediate

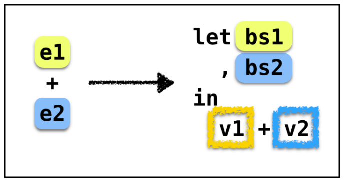
```
((1 + 2) + 3) + 4   ──────▶   let t1 = 1 + 2
                                , t2 = t1 + 3
                              in
                                t2 + 4
```


A-Normalization

Example Anf-3: Both operands are not immediate

```
((1 + 2) + 3)              let t1 = 1  + 2
      +                      , t2 = t1 + 3
((4 + 5) + 6)   ──────▶      , t3 = 4  + 5
                             , t4 = t3 + 6
                           in
                             t2 + t4
```

ANF: General Strategy



ANF Strategy

1. **Invoke** `imm` on both the operands
2. **Concat** the `let` bindings
3. **Apply** the binop to the immediate values

ANF: Implementation

Lets implement the above strategy

```
anf (Prim2 o e1 e2) = lets (b1s ++ b2s)
                        (Prim2 o (Var v1) (Var v2))
  where
    (b1s, v1)       = imm e1
    (b2s, v2)       = imm e2

lets :: [(Id, AnfE)] -> AnfE -> AnfE
lets []         e' = e
lets ((x,e):bs) e' = Let x e (lets bs e’)
```

Intuitively, `lets` *stitches* together a bunch of definitions as follows:

```
lets [(x1, e1), (x2, e2), (x3, e3)] e
  ===> Let x1 e1 (Let x2 e2 (Let x3 e3 e))
```


ANF: Implementation

For `Let` just make sure we recursively `anf` the sub-expressions.

```
anf (Let x e1 e2)    = Let x e1' e2'
  where
    e1'              = anf e1
    e2'              = anf e2
```

Same principle applies to `If`

- use `anf` to recursively transform the branches.

```
anf (If e1 e2 e3) = If e1' e2' e3'
  where
    e1'            = anf e1
    e2'            = anf e2
    e3'            = anf e3
```

ANF: Making Arguments Immediate

The workhorse is the function

```
imm :: BareE -> ([(Id, AnfE)], ImmE)
```

which creates temporary variables to crunch an arbitrary `Bare` into an *immediate* value.

No need to create an variables if the expression is *already* immediate:

```
imm (Number n l) = ( [], Number n l )
imm (Id     x l) = ( [], Id     x l )
```


ANF: Making Arguments Immediate

The tricky case is when the expression has a primop:

```
imm (Prim2 o e1 e2) = ( b1s ++ b2s ++ [(t,  Prim2 o v1 v2)]
                      , Id t  )
  t                 = makeFreshVar ()
  (b1s, v1)         = imm e1
  (b2s, v2)         = imm e2
```

Oh, what shall we do when:

```
imm (If e1 e2 e3)   = ???
imm (Let x e1 e2)   = ???
```


ANF: Making Arguments Immediate

Lets look at an example for inspiration.

That is, simply

- `anf` the relevant expressions,
- bind them to a fresh variable.

```
imm e@(If _ _ _) = immExp e
imm e@(If _ _ _) = immExp e

immExp :: AnfE -> ([(Id, AnfE)], ImmE)
immExp e = ([(t, e')], t)
  where
    e'   = anf e
    t    = makeFreshVar ()
```

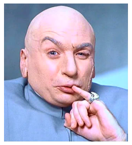
One last thing

Whats up with **makeFreshVar** ?

Wait a minute, what is this magic **FRESH** ?

How can we create **distinct** names out of thin air?

What's that? Global variables? Increment a counter?



"I demand... ONE-*MILLION* fresh variables"

Fresh variables

We will use a counter, but will have to **pass its value around** (just like doTag)

```
anf :: Int -> BareE -> (Int, AnfE)

anf i (Number n l)      = (i, Number n l)

anf i (Id     x l)      = (i, Id     x l)

anf i (Let x e b l)     = (i'', Let x e' b' l)
  where
    (i',  e')           = anf i e
    (i'', b')           = anf i' b

anf i (Prim2 o e1 e2 l) = (i'', lets (b1s ++ b2s) (Prim2 o e1' e2' l))
  where
    (i' , b1s, e1')     = imm i  e1
    (i'', b2s, e2')     = imm i' e2

anf i (If c e1 e2 l)    = (i''', lets bs  (If c' e1' e2' l))
  where
    (i'  , bs, c')      = imm i   c
    (i'' ,     e1')     = anf i'  e1
    (i''',     e2')     = anf i'' e2
```


Fresh variables

```
imm :: Int -> AnfE -> (Int, [(Id, AnfE)], ImmE)

imm i (Number n l)        = (i  , [], Number n l)

imm i (Var x l)           = (i  , [], Var x l)

imm i (Prim2 o e1 e2 l) = (i''', bs, Var v l)
  where
    (i'  , b1s, v1)     = imm i  e1
    (i'' , b2s, v2)     = imm i' e2
    (i''', v)           = fresh i''
    bs                  = b1s ++ b2s ++ [(v, Prim2 o v1 v2 l)]

imm i e@(If _ _ _  l)   = immExp i e

imm i e@(Let _ _ _ l)   = immExp i e

immExp :: Int -> BareE -> (Int, [(Id, AnfE)], ImmE)
immExp i e l  = (i'', bs, Var v ())
  where
    (i' , e') = anf i e
    (i'', v)  = fresh i'
```

Fresh variables

where now, the `fresh` function returns a *new counter* and a variable

```
fresh :: Int -> (Int, Id)
fresh n = (n+1, "t" ++ show n)
```

Note this is super clunky. There *is* a really slick way to write the above code without the clutter of the `i` but thats too much of a digression, but feel free to look it up yourself

Recap and Summary

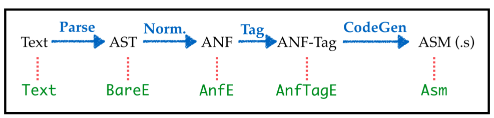
Just created `Boa` with

- Branches (`if`-expressions)
- Binary Operators (`+`, `-`, etc.)

In the process of doing so, we will learned about

- **Intermediate Forms**
- **Normalization**

Specifically,



Questions?