BOA: Branches and Binary Operators

Next, let's 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 | EPrim1 Op1 Expr | EVar Id | ELet Id Expr Expr | EIf Expr Expr Expr
```
Examples

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

- For now, let’s treat 0 as “false” and non-zero as “true”

Example: If1

```c
if 10:
    22
else:
    sub(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, let’s treat 0 as “false” and non-zero as “true”

Example: If2

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

- Since sub(1) is 0 we evaluate the “else” case to get 22

Control Flow in Assembly

To compile branches, we will use:

- labels of the form

```c
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.

- **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

```assembly
Example: if1

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

```assembly
Example: if2

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

```assembly
Example: if3

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

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

X`

Oops, need distinct labels for each branch:
- Require distinct tags for each if-else expression

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

Types: Source

Lets modify the Source Expression

```plaintext
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 modify the Source Expression.

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

```haskell
type Tag = Int
```

We will now use:

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

Types: Assembly

Now, let's extend the Assembly with labels, comparisons and jumps:

```haskell
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

let parseStr s = fmap (const ()) (parse "")
let e = parseStr "if 1: 22 else: 33"

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

```
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.

\[
tag :: \text{BareE} \rightarrow \text{TagE}
\]
\[
tag e = e' \quad \text{where} \quad (_, e') = \text{doTag} \; 0 \; e
\]

Transforms: CodeGen

Now that we have the tags we let’s implement our compilation strategy

\[
\text{compile env (if eCond eTrue eFalse i)}:
\]
\[
= \text{compile env eCond ++} \quad \text{--- 'Cond'}
\]
\[
\quad \quad | \text{Icmp Reg EAX (Const 0)} \quad \text{--- compare result to 0}
\]
\[
\quad \quad \quad , \text{IJe BranchFalse i} \quad \text{--- if-zero then jump to 'False'-block}
\]
\[
\quad ++ \text{compile env eTrue ++} \quad \text{--- code for 'True'-block}
\]
\[
\quad \quad | \text{Ijmp Exit} \quad \text{--- jump to exit (don’t execute 'False')}
\]
\[
\quad ++ \text{ILabel (BranchFalse i)} \quad \text{--- start of 'False'-block}
\]
\[
\quad \quad \quad \text{; compile env eFalse ++} \quad \text{--- code for 'False'-block}
\]
\[
\quad \quad \quad \quad \text{; ILabel (BranchExit i)} \quad \text{--- exit}
\]

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 \( n_1 \times n_2 \)

```assembly
mov eax, n1
mul eax, n2
```
Example: Bin1

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

\[
\begin{align*}
2 + 3 & \quad \text{mov eax, 2} \\
& \quad \text{add eax, 3}
\end{align*}
\]

Strategy: Given \( n1 + n2 \)
- Move \( n1 \) into \( eax \),
- Add \( n2 \) to \( eax \).

Example: Bin2

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

```plaintext
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

```plaintext
mov eax, 10          ; put x on stack
mov [ebp - 4*1], eax ; put x on stack
mov eax, 20          ; put y on stack
mov [ebp - 4*2], eax ; put y on stack
mov eax, 30          ; put z on stack
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] ; grab x
```
Example: Bin2

What if the first operand is a variable?

Simple, just copy the variable off the stack into eax

```
let x = 12
in
  x + 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.

```
let x = 12
, y = 18
in
  x + y
```

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

Example: Bin4

But what if we have nested expressions

\((1 + 2) \times (3 + 4)\)

- 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 \((1 + 2) \times (3 + 4) \times (5 + 6)\) ?
- 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 \((1 + 2) \times (3 + 4)\) not? Because \(1\) and \(x\) are immediate expressions

Their values don’t require any computation!

- Either a \texttt{constant}, or,
- \texttt{variable} whose value is on the stack.

Idea: Administrative Normal Form (ANF)

An expression is in \texttt{Administrative Normal Form (ANF)} if all primitive operations have immediate arguments

\textbf{Primitive Operations:} Those whose values we \texttt{need} for computation to proceed.

- \(v1 + v2\)
- \(v1 - v2\)
- \(v1 \times v2\)
**Conversion to ANF**

However, note the following variant is in ANF

```latex
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

```
Parser
```

**Compiler Pipeline with ANF**

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

**Types: Source**

Let's add binary primitive operators

```latex
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:

```latex
Prim2 Plus (Number 2 ()) (Number 3 ())()
```
Types: Assembly

Need to add X86 instructions for primitive arithmetic:

```haskell
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

```haskell
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

```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 \_1?  

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

Similarly, let's 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

{- A ->} isAnf el
{- B ->} isImm el
{- 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 \text{AnfTagE} to Asm

The compilation from ANF is easy, let's recall our examples and strategy:

\textbf{Strategy}: Given $v_1 + v_2$ (where $v_1$ and $v_2$ are immediate expressions)

\begin{itemize}
  \item Move $v_1$ into eax,
  \item Add $v_2$ to eax.
\end{itemize}

\texttt{Transforms: Compiling AnfTagE to Asm}

\begin{verbatim}
compile :: Env -> TagE -> Asm
compile env (Prim2 o v1 v2)
  = \{ Dmov \ Reg EAX \ (immArg env v1)
       \ (prim2 o) \ Reg EAX \ (immArg env v2)
\}
where we have a helper to find the Asm variant of a \text{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:

\begin{verbatim}
immArg :: Env -> ImmTag -> Arg
immArg env (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)
\end{verbatim}
\end{verbatim}

Transforms: Compiling \text{Bare} to \text{Anf}

Next let's focus on A-Normalization i.e. transforming expressions into ANF

\begin{verbatim}
anf (Number n) = Number n
anf (Var x) = Var x

Interesting cases are the binary operations
\end{verbatim}
A-Normalization

Example Anf-1: Left operand is not immediate

\[
(1 + 2) + 3 \quad \text{in} \quad t1 = 1 + 2 \quad \text{in} \quad t1 + 3
\]

Key Idea: Helper Function

\[
\text{imm} :: \text{BareE} \rightarrow ((\text{Id}, \text{AnfE}), \text{ImmE})
\]

\[
\text{imm e} \quad \text{returns} \quad ((t1, a1),..., (tn, an), v)
\]

where

- \(t1, a1\) are new temporary variables bound to ANF exprs,
- \(v\) is an immediate value (either a constant or variable)

Such that \(e\) is equivalent to

\[
\text{let} \quad t1 = a1; \quad \ldots; \quad t_n = a_n \quad \text{in} \quad v
\]

A-Normalization

Example Anf-2: Left operand is not internally immediate

\[
(1 + 2) + 3 + 4 \quad \text{in} \quad \text{let} t1 = 1 + 2, \quad t2 = t1 + 3 \quad \text{in} \quad t2 + 4
\]

A-Normalization

Example Anf-3: Both operands are not immediate

\[
((1 + 2) + 3) + (4 + 5) + 6 \quad \text{in} \quad \text{let} t1 = 1 + 2, \quad t2 = t1 + 3, \quad t3 = t2 + 4, \quad t4 = t3 + 5, \quad t5 = t4 + 6, \quad t6 = t5 + 7 \quad \text{in} \quad t6 + 8
\]
ANF: General Strategy

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

ANF: Implementation

Let's implement the above strategy

```
anf (Prim2 op e1 e2) = lets (bs1 ++ bs2)
    where
        (bs1, v1) = imm e1
        (bs2, v2) = imm e2
lets :: [(Id, AnfE)] -> AnfE
lets [] e' = e'
lets [(x, e1)] e' = Let x e1 (lets e')
lets [(x1, e1), (x2, e2), (x3, e3)] e' =
    Let x1 e1 (Let x2 e2 (Let x3 e3 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

\[ \text{imm} :: \text{BareE} \rightarrow \{(\text{Id, AnfE}), \text{ImmE}\} \]

which creates temporary variables to crunch an arbitrary \text{Bare} into an immediate value.

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

\[
\begin{align*}
\text{imm} (\text{Number } n l) &= (\{\}, \text{Number } n l) \\
\text{imm} (\text{Id } x l) &= (\{\}, \text{Id } x l)
\end{align*}
\]

ANF: Making Arguments Immediate

The tricky case is when the expression has a primop:

\[
\begin{align*}
\text{imm} (\text{Prim2 } o e1 e2) &= (\text{b1s } \uplus \text{b2s } \uplus [(\text{t, Prim2 } o v1 v2)], \text{Id } t) \\
\text{t} &= \text{makeFreshVar} () \\
\text{b1s, v1} &= \text{imm e1} \\
\text{b2s, v2} &= \text{imm e2}
\end{align*}
\]

Oh, what shall we do when:

\[
\begin{align*}
\text{imm} (\text{If } e1 e2 e3) &= ??? \\
\text{imm} (\text{Let } x e1 e2) &= ???
\end{align*}
\]

ANF: Making Arguments Immediate

Let's look at an example for inspiration.

That is, simply

\[
\begin{align*}
\text{immExp} :: \text{AnfE} \rightarrow \{(\text{Id, AnfE}), \text{ImmE}\} \\
\text{immExp} e@ (\text{If } _ _ _) &= \text{immExp e} \\
\text{immExp} e@ (\text{Let } x e1 e2) &= ???
\end{align*}
\]

where

\[
\begin{align*}
\text{e'} &= \text{anf e} \\
\text{t} &= \text{makeFreshVar} ()
\end{align*}
\]
One last thing

What's 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?

“Demand... ONE-MILLION fresh variables”

Fresh variables

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

\[
\begin{align*}
\text{anf} & : \text{Int} \to \text{BareE} \to \text{Int, AnfE} \\
\text{anf} n & = (i, \text{Number } n i) \\
\text{anf} x & = (i, x i) \\
\text{anf} \text{Let } x e b & = (i'', \text{Let } x e'' b' i) \\
\text{where} & \\
(i'', e') & = \text{anf } e \\
(i', b') & = \text{anf } b \\
\text{anf} (\text{Prim2 } o e1 e2) & = (i'', \text{lets } bs \cdot \text{Prim2 } o e1' e2' i) \\
\text{where} & \\
(i', b1s, e1') & = \text{imm } i e1 \\
(i'', b2s, e2') & = \text{imm } i' e2 \\
\text{anf} (\text{If } c e1 e2) & = (i''', \text{lets } bs \cdot \text{If } c' e1' e2' i) \\
\text{where} & \\
(i', bs, c') & = \text{imm } i c \\
(i'', e1') & = \text{anf } l' e1 \\
(i''', e2') & = \text{anf } l' e2
\end{align*}
\]

\[
\begin{align*}
\text{imm} & :: \text{Int} \to \text{BareE} \to \text{Int, AnfE} \to (\text{Int}, ((\text{Id}, \text{BareE})), \text{ImmE}) \\
\text{imm} n & = (i, [], \text{Number } n i) \\
\text{imm} x & = (i, [], \text{Var } x i) \\
\text{imm} \text{Prim2 } o e1 e2 & = (i'', \text{bs}, \text{Var } v i) \\
\text{where} & \\
(i', bxs, v1) & = \text{imm } (i, e1) \\
(i'', bxs, v2) & = \text{imm } (i, e2) \\
(i''', v) & = \text{fresh } i' \\
bs & = bxs ++ bs \cdot \text{Prim2 } o v1 v l \\
\text{imm} \text{If } _ _ _ & = \text{immExp } i e \\
\text{imm} \text{Let } _ _ _ & = \text{immExp } i e \\
\text{immExp} & :: \text{Int} \to \text{BareE} \to \text{Int, AnfE} \to (\text{Int}, ((\text{Id}, \text{BareE})), \text{ImmE}) \\
\text{immExp } i e l & = (i'', bs, \text{Var } v i) \\
\text{where} & \\
(i', e) & = \text{anf } i e \\
(i'', v) & = \text{fresh } i'
\end{align*}
\]
Fresh variables

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

```haskell
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 | 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 learned about

- Intermediate Forms
- Normalization

Specifically:

Questions?