Branches and Binary Operators

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

Lets 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.
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*

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

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

---

Example: *If2*

```python
if sub(1):
    22
else:
    sub1(0)
```
• Since \( \text{sub}(1) \) is 0 we evaluate the "else" case to get -1

**QUIZ: If3**

if-else is also an *expression* so we can nest them:

What should the following evaluate to?
let x = if sub(1):
    22
else:
    sub1(0)
in
    if x:
        add1(x)
else:
    999

- A. 999
- B. 0
- C. 1
- D. 1000
- E. -1
Control Flow in Assembly

To compile branches, we will use labels, comparisons and jumps

Labels

our_code_label:

...

Labels are “landmarks” – from which execution (control-flow) can be started, or – to which it can be diverted
Comparisons

\texttt{cmp a1, a2}

- Perform a (numeric) \textbf{comparison} between the values \texttt{a1} and \texttt{a2}, and
- Store the result in a special \textbf{processor flag}

Jumps
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

**QUIZ**

Which of the following is a valid x86 encoding of

```python
def if 10:
    22
else
    33
```
**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: If-Expressions to Asm

Let's see how our strategy works by example:

Example: if 1

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

Example: if 10:

22
else:
Example: if1

```
sub1(0)
```

```
if_false:
mov eax, 0
sub eax, 1
if_exit:
```

Example: if2

```
mov eax, 1
sub eax, 1
cmp eax, 0
ie if false
```
Example: if2

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

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

Example: if3

```assembly
mov eax, 10
```
Oops, **cannot reuse labels** across if-expressions!

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

```markdown
Example: if3

```
Example: if3 wrong

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

Example: if3 tagged
Types: Source

Lets modify the Source Expression to add if-else expressions

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

Polymorphic tags of type \(a\) for each sub-expression

- We can have different types of tags
- e.g. Source-Position information for error messages

Lets 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
```

\[ S \rightarrow \text{BareE} \rightarrow \text{TagE} \rightarrow \text{Asm} \]

**Labels, CMP, JUMP**

*Types: Assembly*

Now, let's 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.)
• Let's squeeze in a tagging transform into our pipeline

```
Adding Tagging to the Compiler Pipeline
```

**Transforms: Parse**

Just as before, but now puts a dummy ( ) into each position
\[ \lambda \text{let parseStr s} = \text{fmap (const ()) (parse "" s)} \]

\[ \lambda \text{let e} = \text{parseStr "if 1: 22 else: 33"} \]

\[ \lambda \text{e} \]
\[ \text{If (Number 1 ()) (Number 22 ()) (Number 33 ()) ()} \]

\[ \lambda \text{label e} \]
\[ \text{If (Number 1 ((),0)) (Number 22 ((),1)) (Number 33 ((),2)) ((),3)} \]

**Transforms: Tag**

The key work is done by \( \text{doTag \ i \ e} \)

1. Recursively walk over the \( \text{BareE} \) named \( \text{e} \) starting tagging at counter \( \text{i} \)
2. Return a pair \((i', e')\) of updated counter \(i'\) and tagged expression \(e'\)

```
QUIZ
doTag :: Int -> BareE -> (Int, TagE)
doTag i (Number n _) = (i + 1, Number n i)
doTag i (Var x _) = (i + 1, Var x i)
doTag i (Let x e1 e2 _) = (_2 , Let x e1' e2' i2)

where
    (i1, e1') = doTag i e1
    (i2, e2') = doTag _1 e2
```

What expressions shall we fill in for \(_1\) and \(_2\)?
\{- A \}\_1 = i \\
\_2 = i + 1 \\

\{- B \}\_1 = i \\
\_2 = i1 + 1 \\

\{- C \}\_1 = i \\
\_2 = i2 + 1 \\

\{- D \}\_1 = i1 \\
\_2 = i2 + 1 \\

\{- E \}\_1 = i2 \\
\_2 = i1 + 1
(ProTip: Use mapAccumL)

We can now tag the whole program by

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

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

Transforms: Code Generation
Now that we have the tags we let's 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 (skip `False`-block!)
    ++
        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

You know the drill.

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

Compiling Binary Operations

Lets look at some expressions and figure out how they would get compiled.
Recall: We want the result to be in \texttt{eax} after the instructions finish.

\begin{align*}
\text{\texttt{eax}} & \quad \text{\texttt{mov}} \quad \text{\texttt{eax}}, \quad \text{\texttt{mov}} \quad 33 \\
\text{\texttt{eax}} & \quad \text{\texttt{sub}} \quad \text{\texttt{eax}}, \quad \text{\texttt{mov}} \quad 10
\end{align*}

\textbf{QUIZ}

What is the assembly corresponding to \(33 - 10\)?

- A. \(?1 = \text{sub}, \ ?2 = 33, \ ?3 = \text{mov}, \ ?4 = 10\)
- B. \(?1 = \text{mov}, \ ?2 = 33, \ ?3 = \text{sub}, \ ?4 = 10\) \hfill \checkmark
- C. \(?1 = \text{sub}, \ ?2 = 10, \ ?3 = \text{mov}, \ ?4 = 33\)
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 $n_2$ to $eax$.

**Example: Bin2**

What if the first operand is a variable?

```
let \( x' = 12 \)
in
\[
\begin{align*}
\text{x} + 10
\end{align*}
\]
```

```
\text{mov eax, 12}
\text{mov [esp - 4], eax}
\text{mov eax, [esp - 4]}
\text{add eax, 10}
```

Example: Bin 2
Simple, just copy the variable off the stack into eax

**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
```

```
mov eax, 12
mov [esp - 4], eax
mov eax, 18
```
Example: Bin 3

Strategy: Given $x + n$

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

QUIZ

What is the assembly corresponding to $(10 + 20) \times 30$?
mov eax, 10
?1  eax, ?2
?3  eax, ?4

• A. ?1 = add, ?2 = 30, ?3 = mul, ?4 = 20
• B. ?1 = mul, ?2 = 30, ?3 = add, ?4 = 20
• C. ?1 = add, ?2 = 20, ?3 = mul, ?4 = 30
• D. ?1 = mul, ?2 = 20, ?3 = add, ?4 = 30

Second Operand is Constant

In general, to compile e + n we can do