Lecture 15

CIS 341: COMPILERS

Announcements

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: March 28th**

- No lecture on Thursday, March 22
  - Dr. Z will be away

Adding Integers to Lambda Calculus

exp ::=	
$\mid$ …	
$\mid$ n	*constant integers*
$\mid$ $exp_1 + exp_2$	*binary arithmetic operation*
val ::=	
$\mid$ `fun` x `->` exp	*functions are values*
$\mid$ n	*integers are values*
$n\{v/x\} = n$	*constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$	*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \;\Downarrow (n1 \;[\![+]\!]\; n2)}$$

Object-level '+'

Meta-level '+'

Compiling lambda calculus to straight-line code.
Representing evaluation environments at runtime.

CLOSURE CONVERSION

Compiling First-class Functions

- To implement first-class functions on a processor, there are two problems:
  - First: we must implement substitution of free variables
  - Second: we must separate 'code' from 'data'

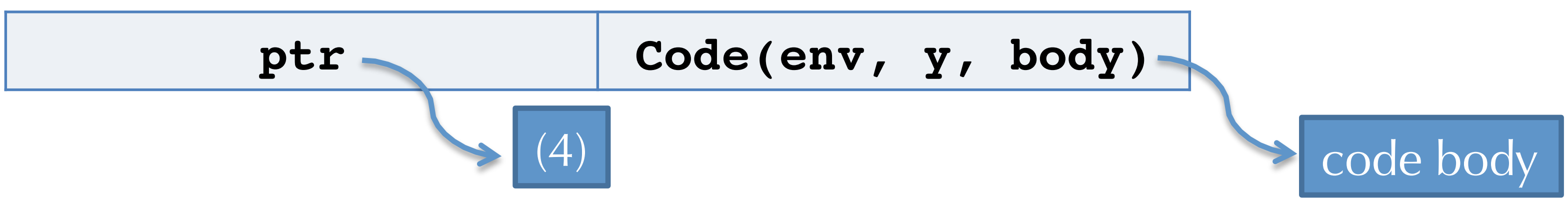
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the “add” function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

- When run the function application: `add 4`
  the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):

```
fun (env, y) -> let x = nth env 0 in x + y
```

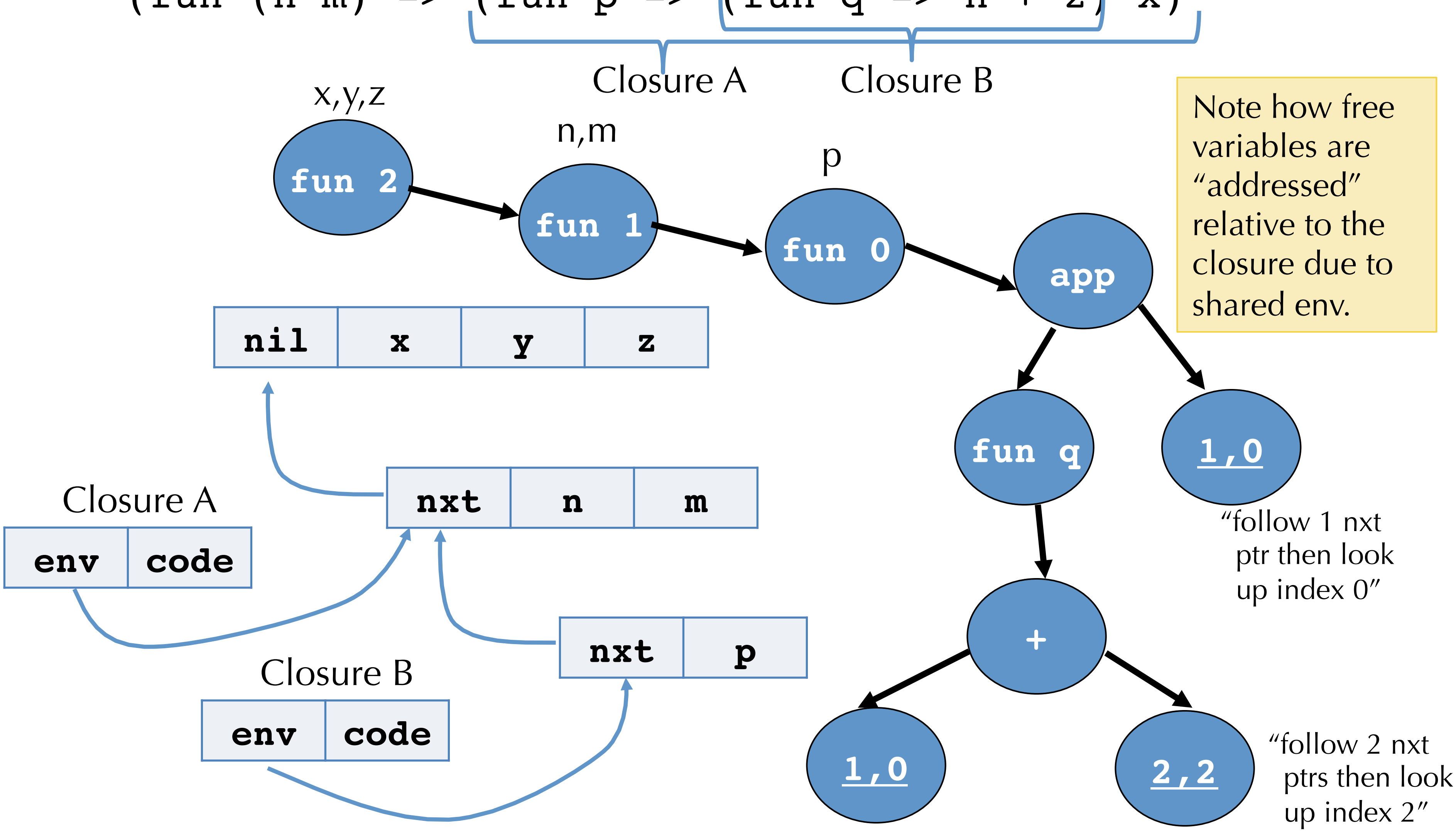
Representing Closures

- As we saw, the simple closure conversion algorithm doesn't generate very efficient code.
  - It stores all the values for variables in the environment, even if they aren't needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures

Array-based Closures with N-ary Functions

```
(fun (x y z) ->
  (fun (n m) -> (fun p -> (fun q -> n + z) x)
```



Scope, Types, and Context

STATIC ANALYSIS

Variable Scoping

- Consider the problem of determining whether a programmer-declared variable is in scope.
- Issues:
  - Which variables are available at a given point in the program?
  - Shadowing – is it permissible to re-use the same identifier, or is it an error?
- Example: The following program is syntactically correct but not well-formed. (y and q are used without being defined anywhere)

```
int fact(int x) {
  var acc = 1;
  while (x > 0) {
    acc = acc * y;
    x = q - 1;
  }
  return acc;
}
```

Q: Can we solve this problem by changing the parser to rule out such programs?

Contexts and Inference Rules

- Need to keep track of contextual information.
  - What variables are in scope?
  - What are their types?

- How do we describe this?
  - In the compiler there's a mapping from variables to information we know about them.

Why Inference Rules?

- They are a compact, precise way of specifying language properties.
  - E.g. ~20 pages for full Java vs. 100's of pages of prose Java Language Spec.
- Inference rules correspond closely to the recursive AST traversal that implements them
- Type checking (and type inference) is nothing more than attempting to prove a different judgment ($G;L \vdash e : t$) by searching backwards through the rules.
- Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment ($G \vdash src \Rightarrow target$)
  - Moreover, the compilation judgment is similar to the typechecking judgment
- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CIS 500 next Fall if you're interested in type systems!

Inference Rules

- We can read a judgment $\mathtt{G;L} \vdash e : t$ as "the expression e is well typed and has type t"
- For any environment G, expression e, and statements s_1, s_2.

$$\mathtt{G;L;rt} \vdash \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2$$

holds if $\mathtt{G;L} \vdash e : \text{bool}$ and $\mathtt{G;L;rt} \vdash s_1$ and $\mathtt{G;L;rt} \vdash s_2$ all hold.

- More succinctly: we summarize these constraints as an *inference rule*:

Premises

Conclusion

$$\frac{\mathtt{G;L} \vdash e : \text{bool} \qquad \mathtt{G;L;rt} \vdash s_1 \qquad \mathtt{G;L;rt} \vdash s_2}{\mathtt{G;L;rt} \vdash \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2}$$

- This rule can be used for *any* substitution of the syntactic metavariables $\mathtt{G}$, e, s_1 and s_2.

Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- Goal of the type checker: verify that such a tree exists.
- Example1: Find a tree for the following program using the inference rules in oat0-defn.pdf:

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

Example2: There is no tree for this ill-scoped program:

```
var x2 = x1 + x1;
return(x2);
```

Example Derivation

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0 ; \cdot ; \texttt{int} \vdash \texttt{var}\ x_1 = 0 ; \texttt{var}\ x_2 = x_1 + x_1 ; x_1 = x_1 - x_2 ; \texttt{return}\ x_1 ; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\,[\text{STMTS}]}{\vdash \texttt{var}\ x_1 = 0 ; \texttt{var}\ x_2 = x_1 + x_1 ; x_1 = x_1 - x_2 ; \texttt{return}\ x_1 ;}\,[\text{PROG}]$$

Example Derivation

$$\mathcal{D}_1 \quad = \quad \cfrac{\cfrac{\cfrac{\cfrac{}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\;[\text{INT}]}{G_0\,;\cdot \vdash 0 : \mathtt{int}}\;[\text{CONST}]}{G_0\,;\cdot \vdash \mathtt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \mathtt{int}}\;[\text{DECL}]}{G_0\,;\,\cdot\,;\mathtt{int} \vdash \mathtt{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \mathtt{int}}\;[\text{SDECL}]$$

$$\mathcal{D}_2 \quad = \quad \cfrac{\cfrac{\cfrac{\cfrac{}{\vdash + : (\mathtt{int}, \mathtt{int}) \to \mathtt{int}}\;[\text{ADD}] \quad \cfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\;[\text{VAR}] \quad \cfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0;\cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}\;[\text{VAR}]}{G_0;\cdot, x_1 : \mathtt{int} \vdash x_1 + x_1 : \mathtt{int}}\;[\text{BOP}]}{G_0;\cdot, x_1 : \mathtt{int};\mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\;[\text{DECL}]}{G_0;\cdot, x_1 : \mathtt{int};\mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}\;[\text{SDECL}]$$

Example Derivation

$$\mathcal{D}_3 \quad \dfrac{\dfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \rightarrow \texttt{int}}[\textsc{add}] \quad \dfrac{x_1\texttt{:int} \in \cdot, x_1\texttt{:int}, x_2\texttt{:int}}{G_0\texttt{;}\cdot, x_1\texttt{:int}, x_2\texttt{:int} \vdash x_1 : \texttt{int}}[\textsc{var}] \quad \dfrac{x_2\texttt{:int} \in \cdot, x_1\texttt{:int}, x_2\texttt{:int}}{G_0\texttt{;}\cdot, x_1\texttt{:int}, x_2\texttt{:int} \vdash x_2 : \texttt{int}}[\textsc{var}]}{\dfrac{G_0\texttt{;}\cdot, x_1\texttt{:int}, x_2\texttt{:int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0\texttt{;}\cdot, x_1\texttt{:int}, x_2\texttt{:int}\texttt{;int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2\texttt{;} \Rightarrow \cdot, x_1\texttt{:int}, x_2\texttt{:int}}[\textsc{assn}]}[\textsc{bop}]$$

$$\mathcal{D}_4 \quad = \quad \dfrac{\dfrac{x_1\texttt{:int} \in \cdot, x_1\texttt{:int}, x_2\texttt{:int}}{G_0\texttt{;}\cdot, x_1\texttt{:int}, x_2\texttt{:int} \vdash x_1 : \texttt{int}}[\textsc{var}]}{G_0\texttt{;}\cdot, x_1\texttt{:int}, x_2\texttt{:int}\texttt{;int} \vdash \texttt{return } x_1\texttt{;} \Rightarrow \cdot, x_1\texttt{:int}, x_2\texttt{:int}}[\textsc{Ret}]$$

Why Inference Rules?

- They are a compact, precise way of specifying language properties.
  - E.g. ~20 pages for full Java vs. 100's of pages of prose Java Language Spec.
- Inference rules correspond closely to the recursive AST traversal that implements them
- Compiling in a context is nothing more an "interpretation" of the inference rules that specify typechecking*: $[\![C \vdash e : t]\!]$
  - Compilation follows the typechecking judgment
- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CIS 500 next Fall if you're interested in type systems!

*Here (and later) we'll write context C for `G;L`, the combination of the global and local contexts.

Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:

$$C \vdash e : t$$

- How do we interpret this information in the target language?

$$[\![C \vdash e : t]\!] = \quad ?$$

- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) sequence of instructions, that, when run, computes the result into some operand

- INVARIANT: if $[\![C \vdash e : t]\!]$ = ty, operand , stream
  then the type (at the target level) of the operand is ty=$[\![t]\!]$

Example

- $C \vdash 341 + 5 : \mathrm{int}$ what is $[\![C \vdash 341 + 5 : \mathrm{int}]\!]$?

$$\dfrac{\dfrac{[\![\vdash 341 : \mathrm{int}]\!] = \texttt{(i64, Const 341, [])}}{[\![C \vdash 341 : \mathrm{int}]\!] = \texttt{(i64, Const 341, [])}} \qquad \dfrac{[\![\vdash 5 : \mathrm{int}]\!] = \texttt{(i64, Const 5, [])}}{[\![C \vdash 5 : \mathrm{int}]\!] = \texttt{(i64, Const 5, [])}}}{[\![C \vdash 341 + 5 : \mathrm{int}]\!] = \texttt{(i64, \%tmp, [\%tmp = add i64 (Const 341) (Const 5)])}}$$

What about the Context?

- What is ⟦C⟧?
- Source level C has bindings like: x:int, y:bool
  - We think of it as a finite map from identifiers to types

- What is the interpretation of C at the target level?

- ⟦C⟧ maps source identifiers, "x" to source types and ⟦x⟧

- What is the interpretation of a variable ⟦x⟧ at the target level?
  - How are the variables used in the type system?

$$\frac{x : t \in L}{G ; L \vdash x : t} \quad \textbf{TYP\_VAR}$$

as expressions
(which denote values)

$$\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L} \quad \textbf{TYP\_ASSN}$$

as addresses
(which can be assigned)

Interpretation of Contexts

- $[\![C]\!]$ = a map from source identifiers to types and target identifiers

- INVARIANT:
  x:t ∈ C   means that

  (1) lookup $[\![C]\!]$ x = (t, `%id_x`)
  (2) the (target) type of `%id_x` is $[\![t]\!]$*   (a pointer to $[\![t]\!]$)

Interpretation of Variables

- Establish invariant for expressions:

$$\left[\!\!\left[\frac{x{:}t \in L}{G;L \vdash x : t}\ \text{TYP\_VAR} \right]\!\!\right] = \texttt{(\%tmp, [\%tmp = load i64* \%id\_x])}$$

as expressions
(which denote values)

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

- What about statements?

$$\left[\!\!\left[\frac{x{:}t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp;\ \Rightarrow L}\ \text{TYP\_ASSN} \right]\!\!\right] = \text{stream @}$$

`[store ⟦t⟧ opn, ⟦t⟧* %id_x]`

as addresses
(which can be assigned)

where (t, `%id_x`) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, `opn`, stream)

Other Judgments?

- Statement:
  $[\![C;\ rt \vdash stmt \Rightarrow C']\!] \;=\; [\![C']\!]$, stream

- Declaration:
  $[\![G;L \vdash t\ x = exp \Rightarrow G;L,x{:}t\ ]\!] = [\![G;L,x{:}t]\!]$, stream

  INVARIANT: stream is of the form:

  ```
        stream' @
        [ %id_x = alloca ⟦t⟧;
          store ⟦t⟧ opn, ⟦t⟧* %id_x ]
  ```

  and $[\![G;L \vdash exp : t\ ]\!] = ([\![t]\!],$ **opn**, stream′)

- Rest follow similarly

COMPILING CONTROL

Translating while

- Consider translating "`while(e) s`":
  - Test the conditional, if true jump to the body, else jump to the label after the body.

$[\![$`C;rt` $\vdash$ `while(e) s` $\Rightarrow$ `C'`$]\!]$ = $[\![$`C'`$]\!]$,

```
lpre:
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
     ⟦C;rt ⊢ s ⇒ C'⟧
     br %lpre
lpost:
```

- Note: writing `opn` = $[\![$`C` $\vdash$ `e : bool`$]\!]$ is pun
  - translating $[\![C \vdash e : \text{bool}]\!]$ generates *code* that puts the result into `opn`
  - In this notation there is implicit collection of the code

Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge and the else branch is optional.

$[\![\texttt{C;rt} \vdash \texttt{if (e}_1\texttt{) s}_1 \texttt{ else s}_2 \Rightarrow \texttt{C'}]\!] = [\![C']\!]$

```
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
     ⟦C;rt ⊢ s₁ ⇒ C'⟧
     br %merge
else:
    ⟦C; rt s₂ ⇒ C'⟧
     br %merge
merge:
```

Connecting this to Code

- Instruction streams:
  - Must include labels, terminators, and “hoisted” global constants
- Must post-process the stream into a control-flow-graph
- See frontend.ml from HW4

OPTIMIZING CONTROL

Standard Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
   z = 3;
else
   z = 4;
return z;
```

```
    %tmp1 = icmp Eq ⟦y⟧, 0        ; !y
    %tmp2 = and ⟦x⟧ ⟦tmp1⟧
    %tmp3 = icmp Eq ⟦w⟧, 0
    %tmp4 = or %tmp2, %tmp3
    %tmp5 = icmp Eq %tmp4, 0
    br %tmp4, label %else, label %then

then:
    store ⟦z⟧, 3
    br %merge

else:
    store ⟦z⟧, 4
    br %merge

merge:
    %tmp5 = load ⟦z⟧
    ret %tmp5
```

Observation

- Usually, we want the translation ⟦e⟧ to produce a value
  - ⟦C ⊢ e : t⟧ = (ty, operand, stream)
  - e.g. ⟦C ⊢ $e_1 + e_2$: int⟧ = (i64, `%tmp`, [`%tmp = add` ⟦e_1⟧ ⟦e_2⟧])

- But when the expression we're compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value.

- In many cases, we can avoid "materializing" the value (i.e. storing it in a temporary) and thus produce better code.
  - This idea also lets us implement different functionality too: e.g. short-circuiting boolean expressions

Idea: Use a different translation for tests

Usual Expression translation:

$[\![C \vdash e : t]\!] = (\text{ty, operand, stream})$

Conditional branch translation of booleans, without materializing the value:

$[\![C \vdash e : \text{bool@}]\!] \ \text{ltrue lfalse} = \text{stream}$

$[\![C, rt \vdash \text{if (e) then s1 else s2} \Rightarrow C']\!] = [\![C']\!],$

```
    insns3
then:
     [[s1]]
     br %merge
else:
    [[s2]]
     br %merge
merge:
```

Notes:

- takes two extra arguments: a "true" branch label and a "false" branch label.
- Doesn't "return a value"
- Aside: this is a form of continuation-passing translation…

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!], \text{insns}_1$

$[\![C, rt \vdash s_2 \Rightarrow C'']\!] = [\![C'']\!], \text{insns}_2$

$[\![C \vdash e : \text{bool@}]\!] \ \texttt{then} \ \texttt{else} = \text{insns}_3$

Short Circuit Compilation: Expressions

- $[\![C \vdash e : \text{bool@}]\!]$ ltrue lfalse = insns

$$\frac{}{[\![C \vdash \text{false} : \text{bool@}]\!]\ \text{ltrue lfalse} = [\texttt{br \%lfalse}]}\ \text{FALSE}$$

$$\frac{}{[\![C \vdash \text{true} : \text{bool@}]\!]\ \text{ltrue lfalse} = [\texttt{br \%ltrue}]}\ \text{TRUE}$$

$$\frac{[\![C \vdash e : \text{bool@}]\!]\ \text{lfalse ltrue} = \text{insns}}{[\![C \vdash !e : \text{bool@}]\!]\ \text{ltrue lfalse} = \text{insns}}\ \text{NOT}$$

Short Circuit Evaluation

Idea: build the logic into the translation

$$\frac{[\![C \vdash e1 : bool@]\!]\ \ ltrue\ right = insns_1 \qquad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1\,|\,e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns₁
right:
     insn₂
```

$$\frac{[\![C \vdash e1 : bool@]\!]\ \ right\ lfalse = insns_1 \qquad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1\,\&\,e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns₁
right:
     insn₂
```

where `right` is a fresh label

Short-Circuit Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
    %tmp1 = icmp Eq ⟦x⟧, 0
    br %tmp1, label %right2, label %right1

right1:
    %tmp2 = icmp Eq ⟦y⟧, 0
    br %tmp2, label %then, label %right2

right2:
    %tmp3 = icmp Eq ⟦w⟧, 0
    br %tmp3, label %then, label %else

then:
    store ⟦z⟧, 3
    br %merge

else:
    store ⟦z⟧, 4
    br %merge

merge:
    %tmp5 = load ⟦z⟧
    ret %tmp5
```