Lecture 18

CIS 4521/5521: COMPILERS

Announcements

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Wednesday, April 9th
  - Available soon (by Saturday morning)
  - **Start Early!**

TYPECHECKING

Recap

- A typechecking (static analysis) specification can be defined by collections of inference rules.
  - Each "judgment" form corresponds to a particular kind of analysis
  - The rule's premises and conclusion specify the intended checks

$$\frac{G \vdash e_1 : T \rightarrow S \qquad G \vdash e_2 : T}{G \vdash e_1\ e_2 : S}\ \boxed{\text{APPLICATION}}$$

- Subtyping introduces a notion of subsumption (inclusion):

subtyping relation

$$\frac{G \vdash e : T \qquad T <: S}{G \vdash e : S}\ \boxed{\text{SUBSUMPTION}}$$

SUBTYPING OTHER TYPES

Extending Subtyping to Other Types

- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$.
  - Example: (Pos * Neg) <: (Int * Int)

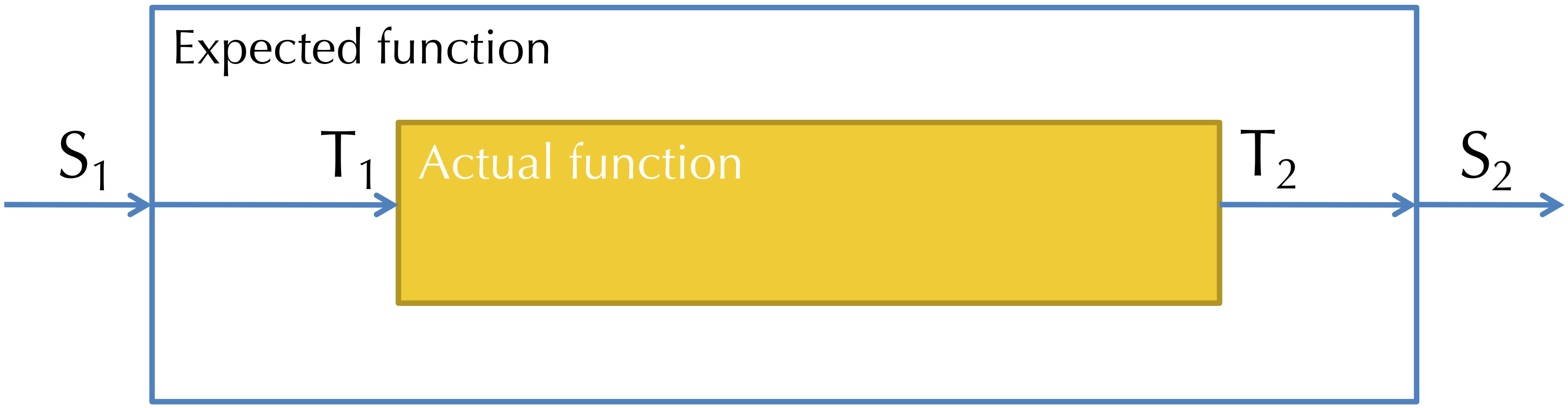
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- What about functions?

- When is $T_1 \rightarrow T_2 \ <: \ S_1 \rightarrow S_2$?

Subtyping for Function Types

- One way to see it:



- Need to convert an S1 to a T1 and T2 to S2, so the argument type is *contravariant* and the output type is *covariant*.

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)}$$

Immutable Records

- Record type: $\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}$
  - Each lab_i is a label drawn from a set of identifiers.

RECORD

$$\frac{G \vdash e_1 : T_1 \qquad G \vdash e_2 : T_2 \qquad \ldots \qquad G \vdash e_n : T_n}{G \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \ldots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}$$

PROJECTION

$$\frac{G \vdash e : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}{G \vdash e.lab_i : T_i}$$

Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

DEPTH

$$\frac{T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}U_1;\ lab_2{:}U_2;\ \ldots\ ;\ lab_n{:}U_n\}}$$

- Width subtyping:
  - Subtype record may have *more* fields:

WIDTH

$$\frac{m \leq n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_m{:}T_m\}}$$

Depth & Width Subtyping vs. Layout

- Width subtyping (without depth) is compatible with "inlined" record representation as with C structs:

```
{x:int; y:int; z:int}   <:   {x:int; y:int}
```

[Width Subtyping]

  – The layout and underlying field indices for 'x' and 'y' are identical.
  – The 'z' field is just ignored

- Depth subtyping (without width) is similarly compatible, assuming that the space used by A is the same as the space used by B whenever A <: B
- But… they don't mix without more work

Immutable Record Subtyping (cont'd)

- Width subtyping assumes an implementation in which order of fields in a record matters:

  {x:int; y:int} ≠ {y:int; x:int}

- But: {x:int; y:int; z:int} <: {x:int; y:int}
  - Implementation: a record is a struct, subtypes just add fields at the *end* of the struct.

- Alternative: allow permutation of record fields:

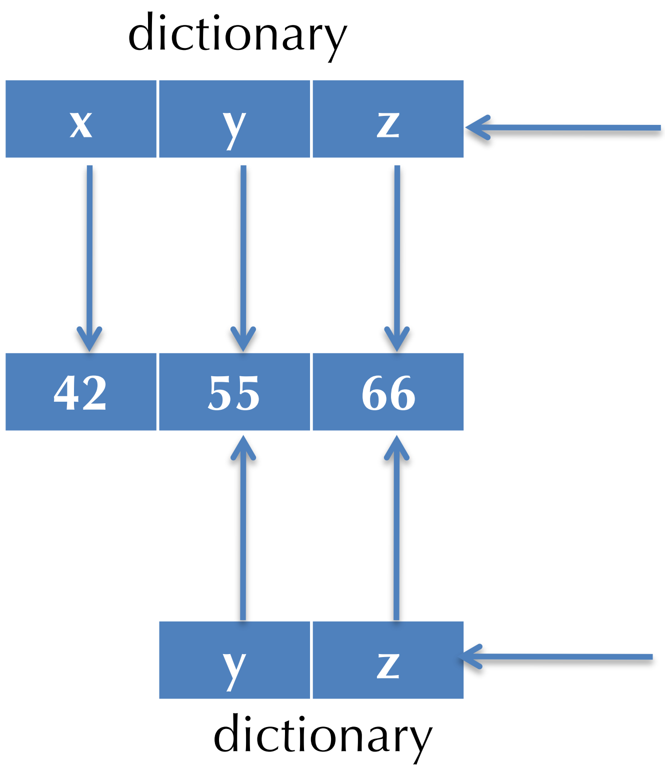
  {x:int; y:int} = {y:int; x:int}

  - Implementation: compiler sorts the fields before code generation.
  - Need to know *all* of the fields to generate the code
- Permutation is not directly compatible with width subtyping:

  {x:int; z:int; y:int} = {x:int; y:int; z:int} </: {y:int; z:int}

If you want both:

- If you want permutability & dropping, you need to either copy (to rearrange the fields) or use a dictionary like this:



p = {x=42; y=55; z=66}:{x:int; y:int; z:int}

q : {y:int; z:int} = p

MUTABILITY & SUBTYPING

NULL

- What is the type of `null`?
- Consider:

```
int[] a = null;    // OK?
int x   = null;    // not OK?
string s = null;   // OK?
```

$\boxed{\text{NULL}}$ $$\frac{}{G \vdash \text{null} : r}$$

- Null has any *reference type*
  – Null is generic

- What about type safety?
  – Requires defined behavior when dereferencing null
    e.g. Java's NullPointerException
  – Requires a safety check for every dereference operation
    (typically implemented using low-level hardware "trap" mechanisms.)

Subtyping and References

- What is the proper subtyping relationship for references and arrays?

- Suppose we have NonZero as a type and the division operation has type: Int → NonZero → Int
  - Recall that NonZero <: Int
- Should (NonZero ref) <: (Int ref) ?
- Consider this program:

```
Int bad(NonZero ref r) {
  Int ref a = r;   (* OK because (NonZero ref <: Int ref*)
  a := 0;          (* OK because 0 : Zero <: Int *)
  return (42 / !r) (* OK because !r has type NonZero *)
}
```

Mutable Structures are Invariant

- Covariant reference types are unsound
  - As demonstrated in the previous example
- Contravariant reference types are also unsound
  - i.e. If $T_1 <: T_2$ then ref $T_2 <:$ ref T_1 is also unsound
  - Exercise: construct a program that breaks contravariant references.

- Moral: Mutable structures are invariant:

  T_1 ref $<:$ T_2 ref   implies $T_1 = T_2$

- Same holds for arrays, OCaml-style mutable records, object fields, etc.
  - Note: Java and C# get this wrong. They allows covariant array subtyping, but then compensate by adding a dynamic check on *every* array update!

Another Way to See It

- We can think of a reference cell as an immutable record (object) with two functions (methods) and some hidden state:

  $$T \text{ ref} \quad \simeq \quad \{\text{get: unit} \rightarrow T; \quad \text{set: } T \rightarrow \text{unit}\}$$

  – get returns the value hidden in the state.
  – set updates the value hidden in the state.

- When is T ref <: S ref?
- Records are like tuples: subtyping extends pointwise over each component.
- $\{\text{get: unit} \rightarrow T; \text{ set: } T \rightarrow \text{unit}\} <: \{\text{get: unit} \rightarrow S; \text{ set: } S \rightarrow \text{unit}\}$
  – get components are subtypes: $\text{unit} \rightarrow T \;<:\; \text{unit} \rightarrow S$
    set components are subtypes: $T \rightarrow \text{unit} \;<:\; S \rightarrow \text{unit}$
- From get, we must have T <: S (covariant return)
- From set, we must have S <: T (contravariant arg.)
- From T <: S and S <: T we conclude T = S.

STRUCTURAL VS. NOMINAL TYPES

Structural vs. Nominal Typing

- Is type equality / subsumption defined by the *structure* of the data or the *name* of the data?
- Example 1: type abbreviations (OCaml) vs. "newtypes" (a la Haskell)

```
(* OCaml: *)
type cents = int    (* cents = int in this scope *)
type age = int

let foo (x:cents) (y:age) = x + y
```

```
(* Haskell: *)
newtype Cents = Cents Integer  (* Integer and Cents arr
                                isomorphic, not identical. *)
newtype Age = Age Integer

foo :: Cents -> Age -> Int
foo x y = x + y                (* Ill typed! *)
```

- Type abbreviations are treated "structurally"
  Newtypes are treated "by name"

Nominal Subtyping in Java

- In Java, Classes and Interfaces must be named and their relationships *explicitly* declared:

```
(* Java: *)
interface Foo {
  int foo();
}

class C {       /* Does not implement the Foo interface */
  int foo() {return 2;}
}

class D implements Foo {
  int foo() {return 4521/5521;}
}
```

- Similarly for inheritance: programmers must declare the subclass relation via the "`extends`" keyword.
  – Typechecker still checks that the classes are structurally compatible

See oat.pdf in HW5

OAT'S TYPE SYSTEM

OAT's Treatment of Types

- Primitive (non-reference) types:
  - `int`, `bool`
- Definitely non-null reference types: `R`
  - (named) mutable structs with (right-oriented) *width* subtyping
  - `string`
  - arrays (including length information, per HW4)
- Possibly-null reference types: `R?`
  - Subtyping: `R <: R?`
  - *Checked downcast* syntax `if?`:

```
int sum(int[]? arr) {
    var z = 0;
    if?(int[] a = arr) {
      for(var i = 0; i<length(a); i = i + 1;) {
        z = z + a[i];
      }
    }
    return z;
}
```

OAT Features

- Named structure types with mutable fields
  - but using structural, width subtyping
- Typed function pointers
- Polymorphic operations: `length` and `==` / `!=`
  - need special case handling in the typechecker
- Type-annotated null values: `t null` always has type `t?`
- Definitely-not-null values means we need an "atomic" array initialization syntax
  - for example, `null` is not allowed as a value of type `int[]`, so to construct a record containing a field of type `int[]`, we need to initialize it
  - subtlety: `int[][]` cannot be initialized by default, but `int[]` can be

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 oat.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;
```

OAT "Returns" Analysis

- Typesafe, statement-oriented imperative languages like OAT (or Java) must ensure that a function (always) returns a value of the appropriate type.
  - Does the returned expression's type match the one declared by the function?
  - Do all paths through the code return appropriately?
- OAT's statement checking judgment
  - takes the expected return type as input: what type should the statement return (or `void` if none)
  - produces a boolean flag as output: does the statement definitely return?

COMPILING WITH TYPES

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?

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

- $[\![C]\!]$ translates contexts
- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) stream 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

- $C \vdash 4521 + 5 : \text{int}$ what is $[\![C \vdash 4521 + 5 : \text{int}]\!]$?

$[\![\vdash 4521 : \text{int}]\!] =$ `(i64, Const 4521, [])` $[\![\vdash 5 : \text{int}]\!] =$ `(i64, Const 5, [])`

-- --

$[\![C \vdash 4521 : \text{int}]\!] =$ `(i64, Const 4521, [])` $[\![C \vdash 5 : \text{int}]\!] =$ `(i64, Const  5, [])`

$[\![C \vdash 4521 + 5 : \text{int}]\!] =$ `(i64, %tmp, [%tmp = add i64 (Const 4521) (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 target types and ⟦x⟧

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

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

$$\left[\!\!\left[\frac{x{:}t \in L}{H;G;L \vdash_{lhs}\ x\ :\ t;\ \top} \right]\!\!\right]$$

as addresses
(which can be assigned)

```
= (T, true, %id_x, [])
```

where (`T`, `%id_x`) = lookup ⟦L⟧ x
and by invariant: `T` = ⟦t⟧*

$$\left[\!\!\left[\frac{H;G;L \vdash_{lhs}\ lhs\ :\ t;\ r}{H;G;L \vdash\ lhs\ :\ t} \right]\!\!\right]$$

lhs as expressions
(which are values)

```
= (%tmp, T,
   stream @
   [%tmp = load T* %ptr])
```

where
⟦H;G;L $\vdash_{lhs}$ lhs : t ; T⟧ = (⟦t⟧, true, `ptr`, stream)

Interpretation of Assignment Stmts

$$\left[\!\left[\frac{\begin{array}{l} H;G;L \vdash_{lhs}\ lhs\ :\ t;\ \top \\ H;G;L \vdash\ exp\ :\ t' \\ H \vdash t'\ \leq\ t \end{array}}{H;G;L;rt \vdash\ lhs = exp;\ \Rightarrow\ L;\bot} \right]\!\right]$$

```
= (⟦H;G;L⟧,
    ptr_code @
    exp_code @
    [store T %e_op, %ptr])
```

assignment to a lhs

where

⟦H;G;L $\vdash_{\text{lhs}}$ lhs : t ; T⟧ = (⟦t⟧, `true, ptr, ptr_code`)

and

⟦H;G;L $\vdash_{\text{lhs}}$ exp : t'⟧ = (⟦t'⟧, `%e_op, exp_code`)

Other Judgments?

- Statement:
  ⟦H;G;L; rt ⊢ stmt ⇒ C′⟧ = ⟦C′⟧ , stream

- Declaration:
  ⟦H;G;L ⊢ var x = exp ⇒ G;L,x:t ⟧ = ⟦G;L,x:t⟧, stream

  INVARIANT: stream is of the form:
  ```
  stream' @
  [E %id_x = alloca ⟦t⟧;
   I store ⟦t⟧ opn, ⟦t⟧* %id_x ]
  ```

  and ⟦H;G;L ⊢ 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 ⊢ while(e) s ⇒ 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 ⊢ e : bool⟧` is pun
  - translating ⟦C ⊢ e : 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.

$$[\![C;rt \vdash \texttt{if } (e_1)\ s_1 \texttt{ else } s_2 \Rightarrow C']\!] = [\![C']\!]$$

```
    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 i64` ⟦e_1⟧ ⟦e_2⟧])

- But when the boolean 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

- Make up new "judgement" that is similar to ⟦C ⊢ e : bool⟧ but has a different semantics. Call it ⟦C ⊢ e : bool@⟧

Idea: Use a different translation for tests

Usual Expression translation:

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

Conditional branch translation of booleans, without materializing the value:

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

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

```
    insns₃
then:
     ⟦s1⟧
     br %merge
else:
    ⟦s₂⟧
     br %merge
merge:
```

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!]$, $insns_1$

$[\![C, rt \vdash s_2 \Rightarrow C'']\!] = [\![C'']\!]$, $insns_2$

$[\![C \vdash e : bool@]\!]$ `then` `else` = $insns_3$

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…

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
```