Lecture 17

CIS 341: COMPILERS

Announcements

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Friday, April 23rd
  - Available soon afternoon
  - **Start Early!**

Simply-typed Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Recall the free variable calculation
  - Given: G, a map of variable identifiers to types, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e : T$ means "the expression e computes a value of type T, assuming its free variables have the types given in G"

$$\frac{x{:}T \in G}{G \vdash x : T}$$

"the variable x has type T an is in scope"

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

"e_1 is a function from T2 to T and e_2 is an expression of type T2"

$$\frac{G, x : T \vdash e : S}{G \vdash \texttt{fun}\ (x{:}T) \rightarrow e : T \rightarrow S}$$

"Given an input of type T, this function computes a result of type S"

Adding Integers

- For the language in "tc.ml" we have five inference rules:

INT

$$\frac{}{G \vdash i : \text{int}}$$

VAR

$$\frac{x : T \in G}{G \vdash x : T}$$

ADD

$$\frac{G \vdash e_1 : \text{int} \qquad G \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

FUN

$$\frac{G, x : T \vdash e : S}{G \vdash \texttt{fun (x:T) -> e} : T \to S}$$

APP

$$\frac{G \vdash e_1 : T \to S \qquad G \vdash e_2 : T}{G \vdash e_1\ e_2 : S}$$

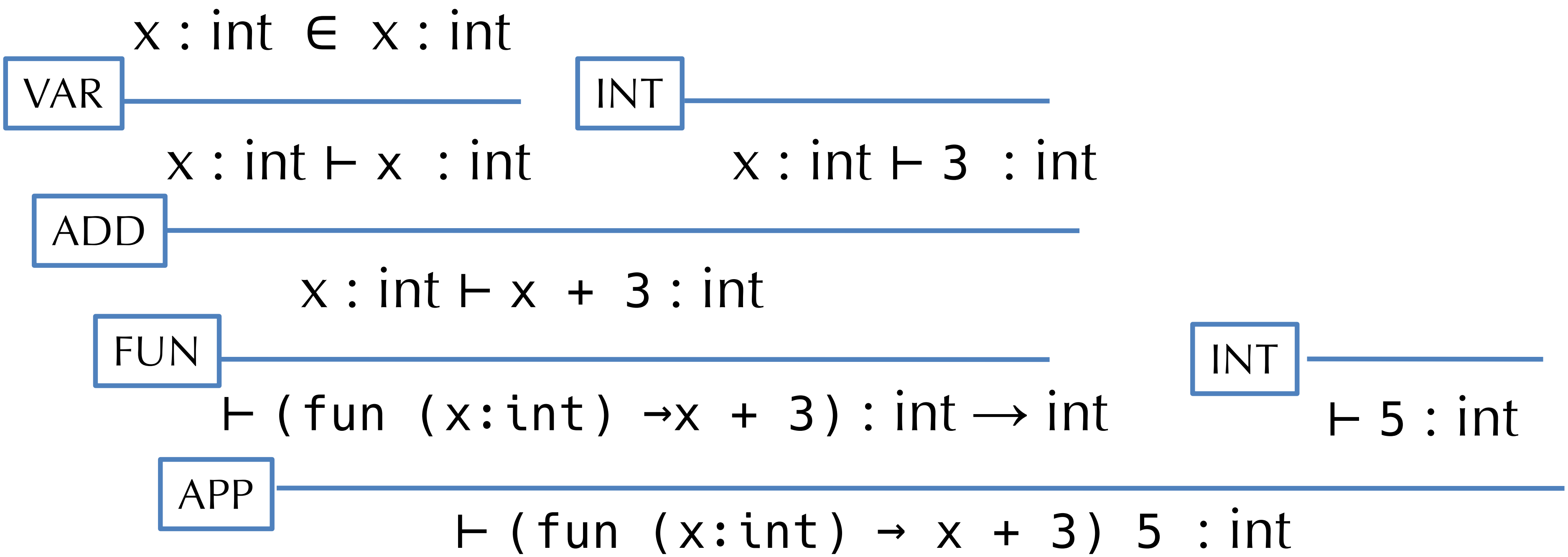
- Note how these rules correspond to the code.

Type 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 typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:

$\vdash$ `(fun (x:int) → x + 3) 5` : int

Example Derivation Tree



- Note: the OCaml function `typecheck` verifies the existence of this tree. The structure of the recursive calls when running `typecheck` is the same shape as this tree!
- Note that x : int ∈ E is implemented by the function `lookup`

Ill-typed Programs

- Programs without derivations are ill-typed

  Example: There is no type T such that

  $\vdash$ `(fun (x:int) → x 3) 5` : T

$x : \text{int} \to T \;\notin\; x : \text{int}$

VAR

$x : \text{int} \vdash x \; : \text{int} \to T$ $\qquad x : \text{int} \vdash 3 \; : \text{int}$

APP

$x : \text{int} \vdash x \;\; 3 : T$

FUN

$\vdash$ `(fun (x:int) →x 3)` : int → T $\qquad \vdash 5 : \text{int}$

APP

$\vdash$ `(fun (x:int) → x 3) 5` : T

Type Safety

"Well typed programs do not go wrong."

– Robin Milner, 1978

Theorem: (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$.

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

Notes about this Typechecker

- The interpreter evaluates the body of a function only when it's applied.
- The typechecker always checks the body of the function
  - even if it's never applied
  - We *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 `->` t_2).

- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get.
  - But we can calculate e_1's type, check that e_2 is an argument of the right type, and determine what type e_1 will return.

- Question: Why is this an approximation?
- Question: What if `well_typed` always returns `false`?

oat.pdf

TYPECHECKING OAT

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);
```

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

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

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

Example Derivation

$$\mathcal{D}_3 \quad = \quad \dfrac{\dfrac{}{\vdash \texttt{-} : (\texttt{int}, \texttt{int}) \to \texttt{int}}[\text{ADD}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}[\text{VAR}] \quad \dfrac{x_2 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_2 : \texttt{int}}[\text{VAR}]}{\dfrac{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0 ; \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} ; \texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2 \texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}[\text{ASSN}]}[\text{BOP}]$$

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

Type Safety For General Languages

Theorem: (Type Safety)

If $\vdash P : t$ is a well-typed program, then either:
(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

- Well-defined termination could include:
  - halting with a return value
  - raising an exception
- Type safety rules out undefined behaviors:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language
- What is "defined" depends on the language semantics…

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 ($E \vdash e : t$) 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 rules are very similar in structure to the typechecking rules
- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CIS 500 if you're interested in type systems!

COMPILING

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

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

$[\![\vdash 341 : \text{int}]\!] =$ `(i64, Const 341, [])`

$[\![C \vdash 341 : \text{int}]\!] =$ `(i64, Const 341, [])`

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

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

$[\![C \vdash 341 + 5 : \text{int}]\!] =$ `(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:

$$\left[\!\left[\frac{x{:}t \in L}{G;L \vdash x : t}\ \text{TYP\_VAR} \right]\!\right] = $$

as expressions
(which denote values)

```
(%tmp,  [%tmp = load i64* %id_x])
```

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

- What about statements?

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

as addresses
(which can be assigned)

= stream @
`[store` ⟦t⟧ `opn,` ⟦t⟧`* %id_x]`

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

Other Judgments?

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

- Declaration:
  ⟦G;L ⊢ t x = exp ⇒ 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 ⊢ 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 ⊢ if (e_1) s_1 else s_2 ⇒ 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:

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

Conditional branch translation of booleans, without materializing the value:

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

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

```
   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

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

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

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

Short Circuit Compilation: Expressions

- ⟦C ⊢ e : 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
```

Beyond describing "structure"… describing "properties"

Types as sets

Subsumption

TYPES, MORE GENERALLY

Tuples

- ML-style tuples with statically known number of products:
- First: add a new type constructor: $T_1 * \ldots * T_n$

TUPLE

$$\frac{G \vdash e_1 : T_1 \quad \ldots \quad G \vdash e_n : T_n}{G \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ

$$\frac{G \vdash e : T_1 * \ldots * T_n \quad 1 \leq i \leq n}{G \vdash \mathrm{prj}_i\ e : T_i}$$

Arrays

- Array constructs are not hard
- First: add a new type constructor: T[]

NEW

$$\frac{G \vdash e_1 : \text{int} \qquad G \vdash e_2 : T}{G \vdash \text{new } T[e_1](e_2) \; : T[]}$$

e_1 is the size of the newly allocated array. e_2 initializes the elements of the array.

INDEX

$$\frac{G \vdash e_1 : T[] \qquad G \vdash e_2 : \text{int}}{G \vdash e_1[e_2] \; : T}$$

UPDATE

$$\frac{G \vdash e_1 : T[] \qquad G \vdash e_2 : \text{int} \qquad G \vdash e_3 : T}{G \vdash e_1[e_2] = e_3 \text{ ok}}$$

Note: These rules don't ensure that the array index is in bounds – that should be checked *dynamically*.

References

- ML-style references (note that ML uses only expressions)
- First, add a new type constructor: T ref

REF

$$\frac{G \vdash e : T}{G \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{G \vdash !e \ : T}$$

ASSIGN

$$\frac{G \vdash e_1 : T \text{ ref} \quad E \vdash e_2 : T}{G \vdash e_1 := e_2 \ : \text{unit}}$$

Note the similarity with the rules for arrays…

What are types, anyway?

- A *type* is just a predicate on the set of values in a system.
  - For example, the type "int" can be thought of as a boolean function that returns "true" on integers and "false" otherwise.
  - Equivalently, we can think of a type as just a *subset* of all values.

- For efficiency and tractability, the predicates are usually taken to be very simple.
  - Types are an *abstraction* mechanism

- We can easily add new types that distinguish different subsets of values:

```
type tp =
    | IntT                    (* type of integers *)
    | PosT | NegT | ZeroT     (* refinements of ints *)
    | BoolT                   (* type of booleans *)
    | TrueT | FalseT          (* subsets of booleans *)
    | AnyT                    (* any value *)
```

Modifying the typing rules

- We need to refine the typing rules too…
- Some easy cases:
  - Just split up the integers into their more refined cases:

P-INT

$$\frac{i > 0}{G \vdash i : \mathrm{Pos}}$$

N-INT

$$\frac{i < 0}{G \vdash i : \mathrm{Neg}}$$

ZERO

$$\frac{}{G \vdash 0 : \mathrm{Zero}}$$

- Same for booleans:

TRUE

$$\frac{}{G \vdash \mathrm{true} : \mathrm{True}}$$

FALSE

$$\frac{}{G \vdash \mathrm{false} : \mathrm{False}}$$

What about "if"?

- Two cases are easy:

IF-T
$$\frac{G \vdash e_1 : True \quad G \vdash e_2 : T}{G \vdash if\ (e_1)\ e_2\ else\ e_3 : T}$$

IF-F
$$\frac{G \vdash e_1 : False \quad E \vdash e_3 : T}{G \vdash if\ (e_1)\ e_2\ else\ e_3 : T}$$

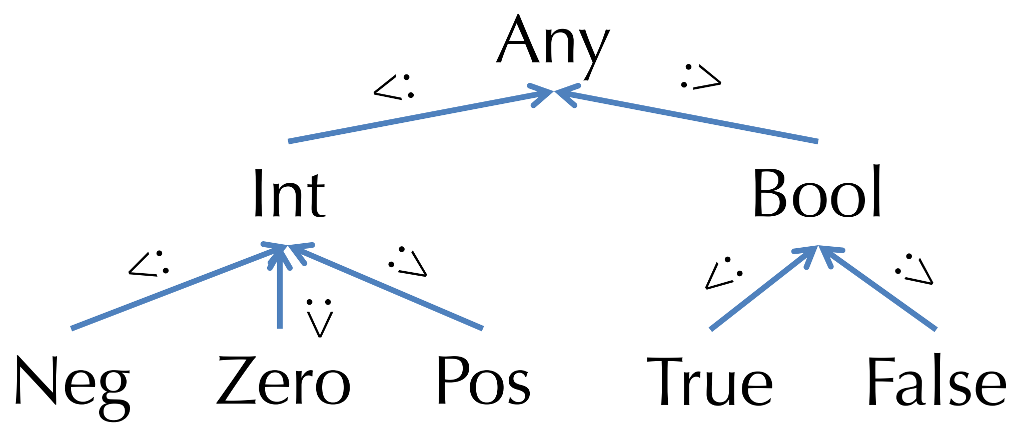
- What happens when we don't know statically which branch will be taken?
- Consider the typechecking problem:

$$x{:}bool \vdash if\ (x)\ 3\ else\ {-1} : ?$$

- The true branch has type Pos and the false branch has type Neg.
  - What should be the result type of the whole if?

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: Pos ⊆ Int
- This subset relation gives rise to a *subtype* relation: Pos <: Int
- Such inclusions give rise to a *subtyping hierarchy*:



- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB) according to the hierarchy.
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
  - Note: might want to add types for "NonZero", "NonNegative", and "NonPositive" so that set union on values corresponds to taking LUBs on types.

"If" Typing Rule Revisited

- For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

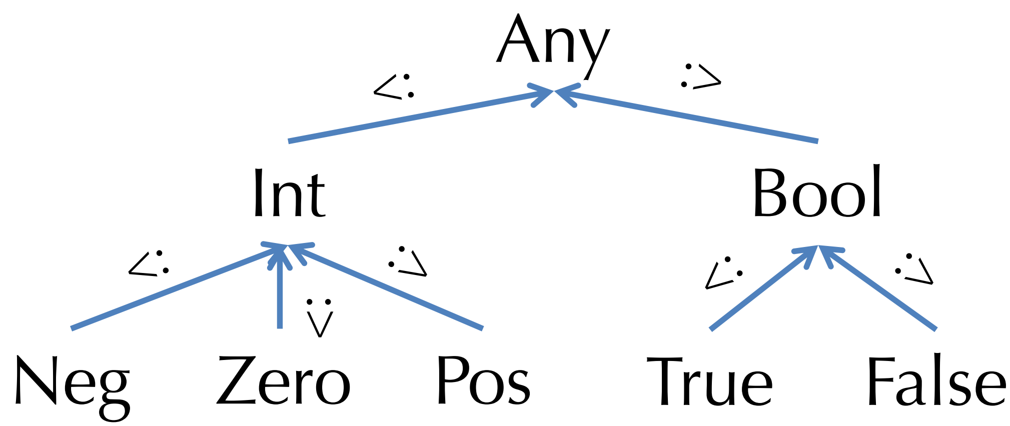
IF-BOOL

$$\frac{G \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \qquad G \vdash e_3 : T_2}{G \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \text{LUB}(T_1, T_2)}$$

- Note that LUB(T_1, T_2) is the most precise type (according to the hierarchy) that is able to describe any value that has either type T_1 or type T_2.
- In math notation, LUB(T1, T2) is sometimes written $T_1 \vee T_2$
- LUB is also called the *join* operation.

Subtyping Hierarchy

- A *subtyping hierarchy*:



- The subtyping relation is a *partial order*:
  - Reflexive: T <: T   for any type T
  - Transitive: T_1 <: T_2 and T_2 <: T_3 then T_1 <: T_3
  - Antisymmetric: It T_1 <: T_2 and T_2 <: T_1 then $T_1 = T_2$

Soundness of Subtyping Relations

- We don't have to treat *every* subset of the integers as a type.
  - e.g., we left out the type NonNeg

- A subtyping relation T_1 <: T_2 is *sound* if it approximates the underlying semantic subset relation.
- Formally: write $[\![T]\!]$ for the subset of (closed) values of type T
  - i.e. $[\![T]\!] = \{v \mid \vdash v : T\}$
  - e.g. $[\![Zero]\!] = \{0\}$, $[\![Pos]\!] = \{1, 2, 3, \ldots\}$

- If T_1 <: T_2 implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then T_1 <: T_2 is sound.
  - e.g. Pos <: Int is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  - e.g. Int <: Pos is not sound, since it is *not* the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$

Soundness of LUBs

- Whenever you have a sound subtyping relation, it follows that:

  $$[\![\text{LUB}(T_1, T_2)]\!] \supseteq [\![T_1]\!] \cup [\![T_2]\!]$$

  – Note that the LUB is an over approximation of the “semantic union”
  – Example: $[\![\text{LUB(Zero, Pos)}]\!] = [\![\text{Int}]\!] = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq$
  $\{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\![\text{Zero}]\!] \cup [\![\text{Pos}]\!]$

- Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule).
- It just so happens that LUBs on subtypes of Int are *sound* for +

ADD

$$\frac{G \vdash e_1 : T_1 \qquad G \vdash e_2 : T_2 \qquad T_1 <: \text{Int} \quad T_2 <: \text{Int}}{G \vdash e_1 + e_2 : T_1 \vee T_2}$$

Subsumption Rule

- When we add subtyping judgments of the form T <: S we can uniformly integrate it into the type system generically:

SUBSUMPTION

$$\frac{G \vdash e : T \quad T <: S}{G \vdash e : S}$$

- Subsumption allows any value of type T to be treated as an S whenever T <: S.
- Adding this rule makes the search for typing derivations more difficult – this rule can be applied anywhere, since T <: T.
  - But careful engineering of the typing system can incorporate the subsumption rule into a deterministic algorithm. (See, e.g., the OAT type system)

Downcasting

- What happens if we have an Int but need something of type Pos?
  - At compile time, we don't know whether the Int is greater than zero.
  - At run time, we do.

- Add a "checked downcast"

$$\frac{G \vdash e_1 : \text{Int} \qquad G, x : \text{Pos} \vdash e_2 : T_2 \qquad G \vdash e_3 : T_3}{G \vdash \text{ifPos } (x = e_1)\ e_2 \text{ else } e_3 : T_2 \vee T_3}$$

- At runtime, ifPos checks whether e_1 is > 0. If so, branches to e_2 and otherwise branches to e_3.
- Inside the expression e_2, x is the name for e_1's value, which is known to be strictly positive because of the dynamic check.
- Note that such rules force the programmer to add the appropriate checks
  - We could give integer division the type: Int → NonZero → Int

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)

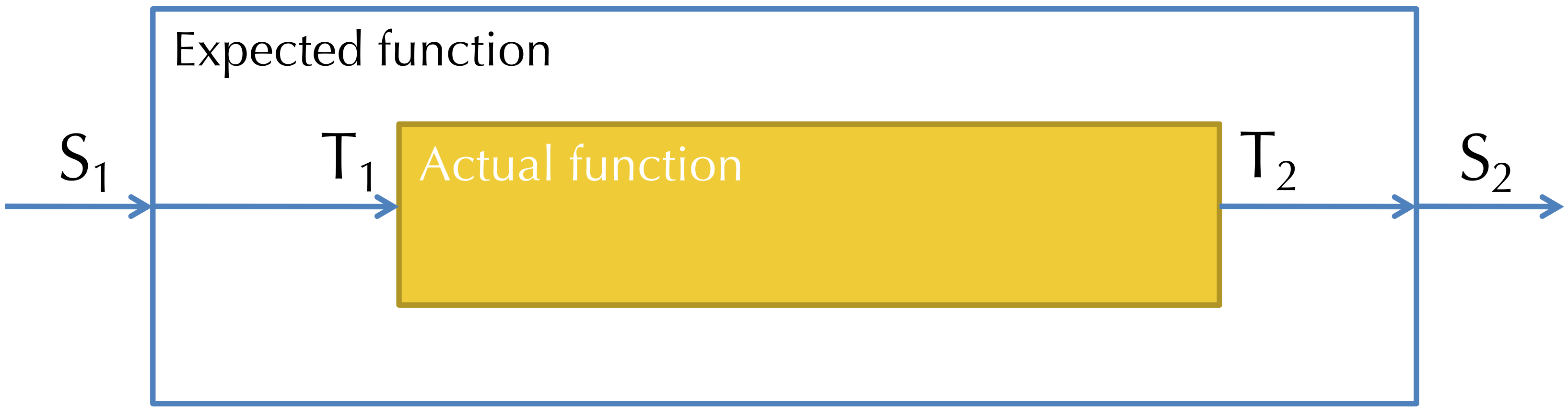
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- 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:

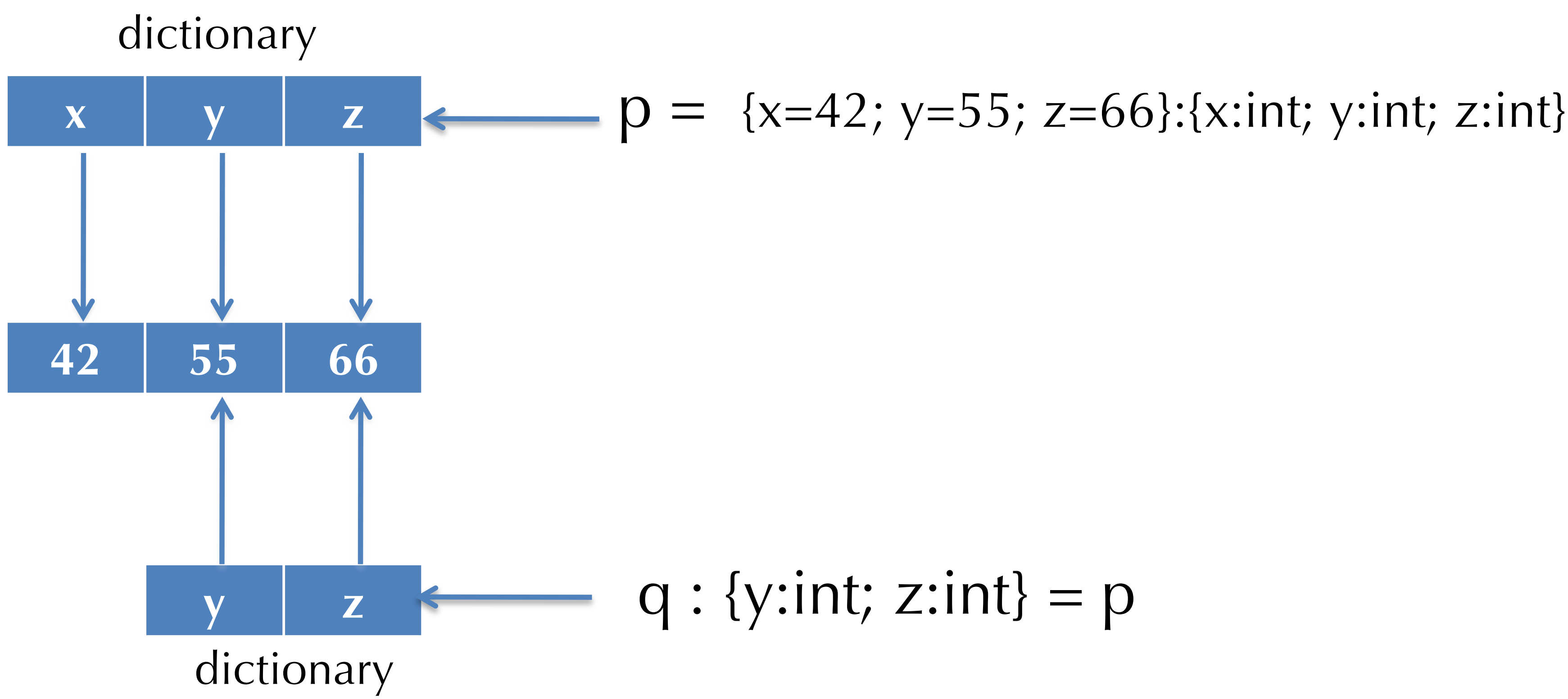
  {x:int; y:int} = {y:int; x:int}

  - 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:



MUTABILITY & SUBTYPING

NULL

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

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

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:

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

  - 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.
- $\{\text{get: unit} \to T; \text{ set: } T \to \text{unit}\} <: \{\text{get: unit} \to S; \text{ set: } S \to \text{unit}\}$
  - get components are subtypes: $\text{unit} \to T \;<:\; \text{unit} \to S$
    set components are subtypes: $T \to \text{unit} \;<:\; S \to \text{unit}$
- 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 341;}
}
```

- 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

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?