Lecture 17

CIS 4521/5521: COMPILERS

Announcements

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: Wednesday, March 26th**
  - **Test case Due: TONIGHT at 10:00PM**

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Wednesday, April 9th
  - Available on Thursday
  - Start early!

Inference Rules

- We can read a judgment $G \vdash e$ as
  "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements s_1, s_2.

$$G \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2$$

  holds if $G \vdash e$ and $G \vdash s_1$ and $G \vdash s_2$ all hold.
- More succinctly: we summarize these constraints as an ***inference rule***:

Premises

Conclusion

$$\frac{G \vdash e \qquad G \vdash s_1 \qquad G \vdash s_2}{G \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

- Such a rule can be used for *any* substitution of the syntactic metavariables G, e, s_1 and s_2.

CBV Operational Semantics

- This is *call-by-value* semantics:
  function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\texttt{fun}\ x \rightarrow exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2\ \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

CBN Operational Semantics

- This is *call-by-name* semantics:
  function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\texttt{fun}\ x \rightarrow exp_3) \qquad exp_3\{exp_2/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, substitute the argument into the function body, and then keep evaluating. "

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

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

APP

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

- Note how these rules correspond to the code.
- By convention, if G is empty we leave that spot blank.

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:

⊢ `(fun (x:int) → x + 3) 5` : int

Example Derivation Tree

$$\text{APP}\;\dfrac{\text{FUN}\;\dfrac{\text{ADD}\;\dfrac{\text{VAR}\;\dfrac{x : \text{int} \;\in\; x : \text{int}}{x : \text{int} \vdash x \;:\; \text{int}} \qquad \text{INT}\;\dfrac{}{x : \text{int} \vdash 3 \;:\; \text{int}}}{x : \text{int} \vdash x + 3 : \text{int}}}{\vdash (\texttt{fun (x:int)} \to \texttt{x + 3}) : \text{int} \longrightarrow \text{int}} \qquad \text{INT}\;\dfrac{}{\vdash 5 : \text{int}}}{\vdash (\texttt{fun (x:int)} \to \texttt{x + 3) 5} \;:\; \text{int}}$$

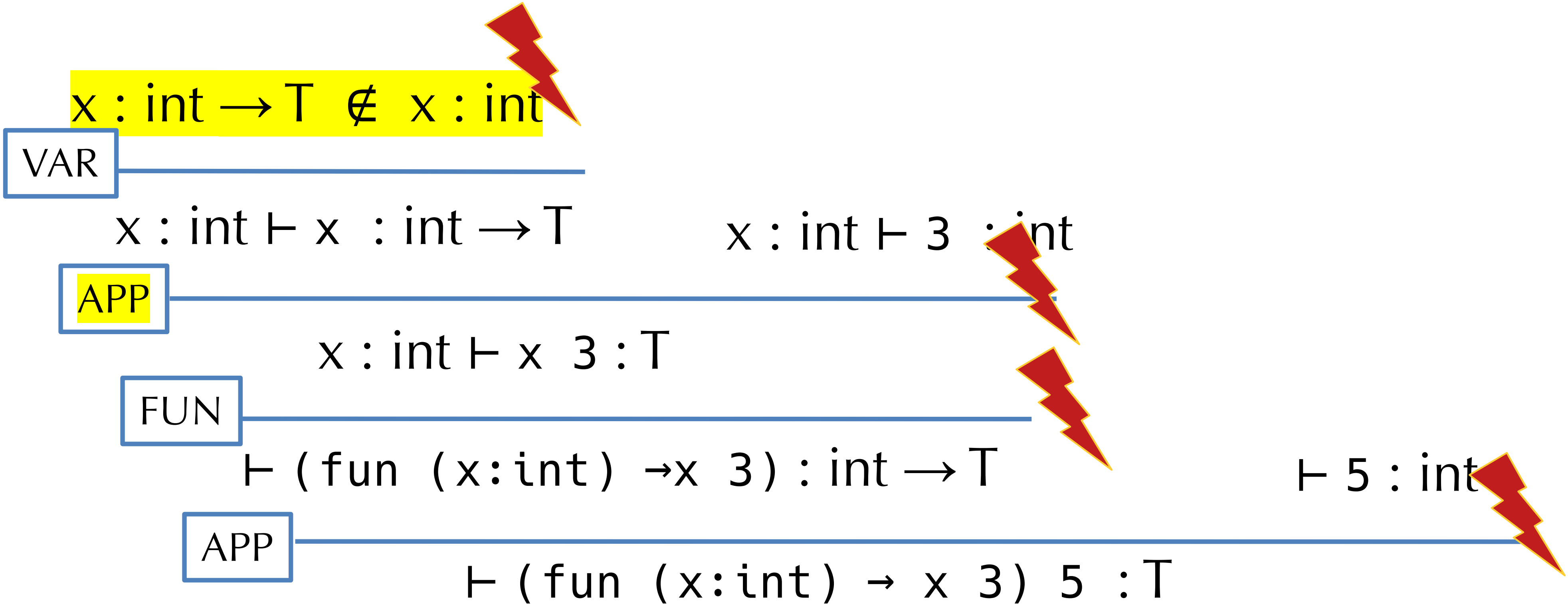
- Note: the OCaml function `typecheck` verifies the existence of this tree. The structure of the recursive calls when running `typecheck` is the same shape as this tree!
- Note that $x : \text{int} \in E$ is implemented by the function `lookup`

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



$x : int \to T \notin x : int$

VAR

$x : int \vdash x : int \to T$ $\qquad$ $x : int \vdash 3 : int$

APP

$x : int \vdash x\ 3 : T$

FUN

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

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`?

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…

B

MORE TYPES

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

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

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…

Beyond describing “structure”… describing “properties”

Types as sets

Subsumption

TYPES, MORE GENERALLY

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:

$$\boxed{\text{P-INT}}\quad \frac{i > 0}{G \vdash i : \text{Pos}} \qquad \boxed{\text{N-INT}}\quad \frac{i < 0}{G \vdash i : \text{Neg}} \qquad \boxed{\text{ZERO}}\quad \frac{}{G \vdash 0 : \text{Zero}}$$

- Same for booleans:

$$\boxed{\text{TRUE}}\quad \frac{}{G \vdash \text{true} : \text{True}} \qquad \boxed{\text{FALSE}}\quad \frac{}{G \vdash \text{false} : \text{False}}$$

What about "if"?

- Two cases are easy:

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

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

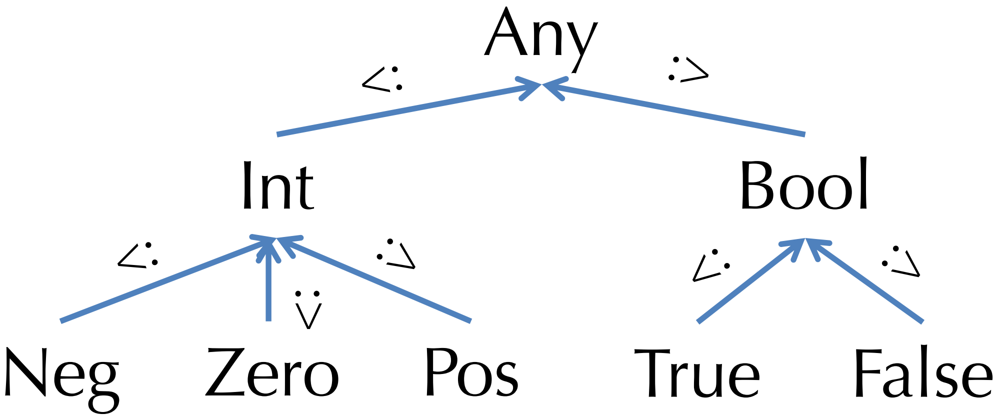
- What happens when we don't know statically which branch will be taken?
- Consider the typechecking problem:

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

- 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 $\subseteq$ 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:

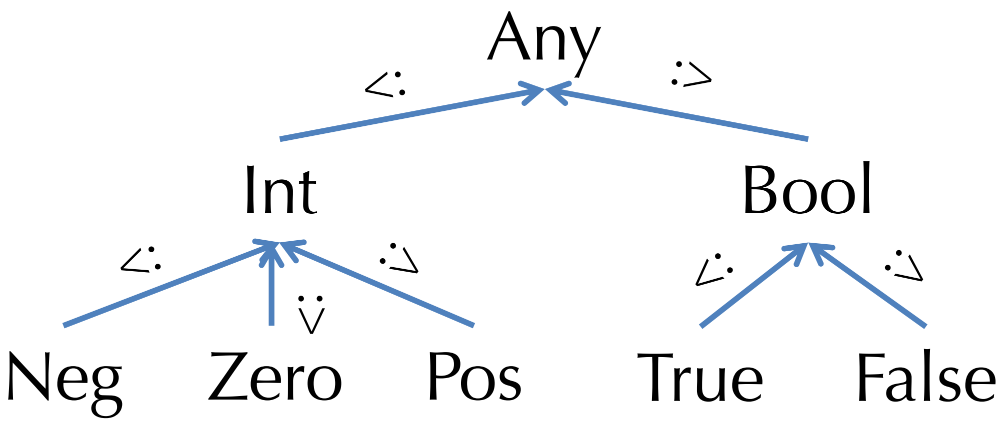
IF-BOOL

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

- 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. $[\![\text{Zero}]\!] = \{0\}$, $[\![\text{Pos}]\!] = \{1, 2, 3, \ldots\}$

- 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: $[\![\text{LUB(Zero, Pos)}]\!] = [\![\text{Int}]\!] = \{\dots,-3,-2,-1,0,1,2,3,\dots\} \supseteq$
    $\{0,1,2,3,\dots\} = \{0\} \cup \{1,2,3,\dots\} = [\![\text{Zero}]\!] \cup [\![\text{Pos}]\!]$

- 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"

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

- 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: $\mathrm{Int} \rightarrow \mathrm{NonZero} \rightarrow \mathrm{Int}$

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)

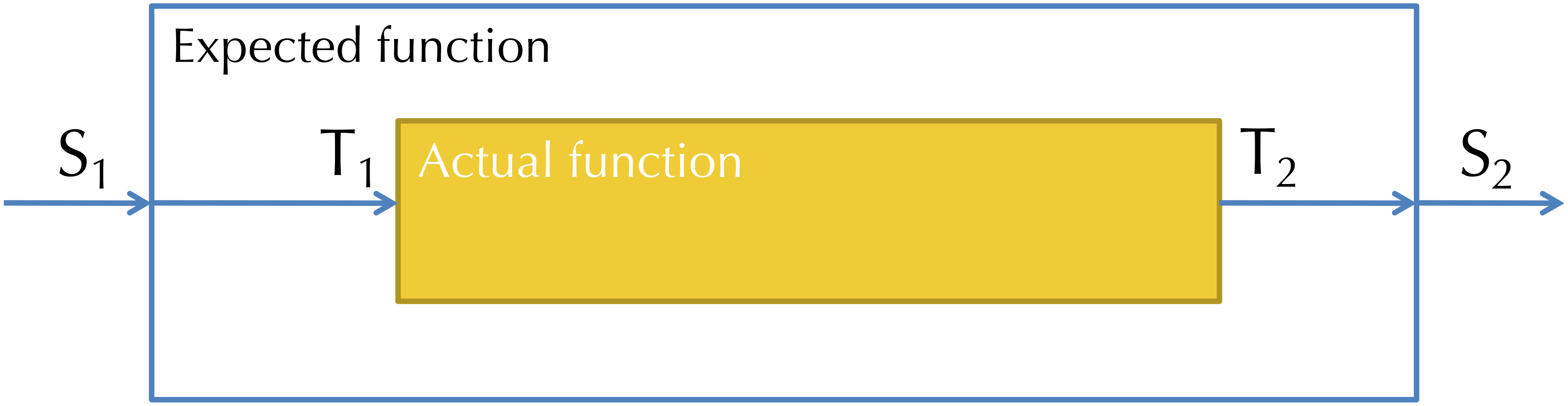
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- 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:

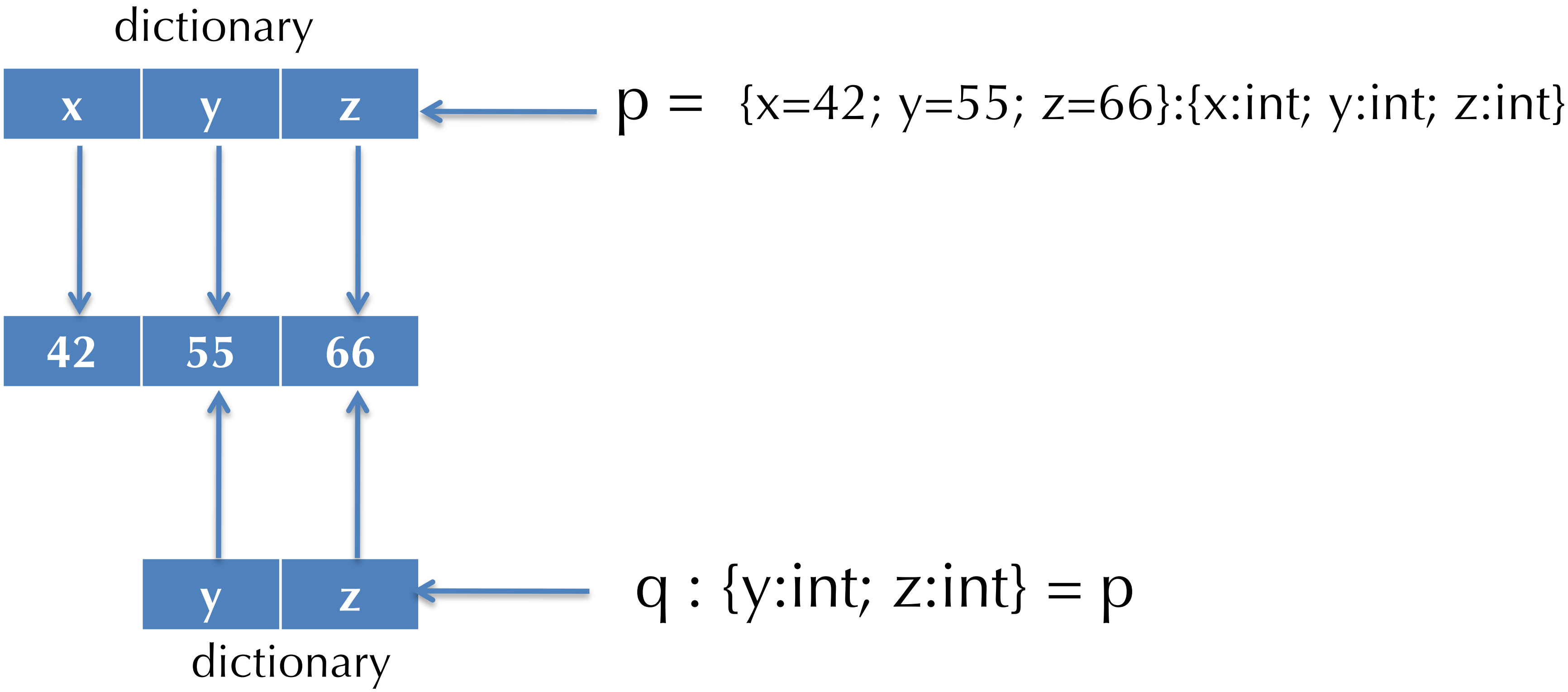
  {x:int; y:int} ≠ {y:int; x:int}
- But: {x:int; y:int; z:int} <: {x:int; y:int}
  - Implementation: a record is a struct, subtypes just add fields at the *end* of the struct.
- Alternative: allow permutation of record fields:

  {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:



dictionary

x y z

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

42 55 66

y z

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

dictionary

MUTABILITY & SUBTYPING

NULL

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

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

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

- 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 4521/5521;}
}
```

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