HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 15: Subtyping

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic

Announcements

- HW4 Oat v1 out
  - Due Tuesday Oct 29 (7 days)
- Reference solns
  - Will be released on Canvas
  - HW2 later today
  - HW3 later this week

Today

- Types as sets of values
- Subtyping
  - Subsumption
  - Downcasting
  - Functions
  - Records
  - References

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
$$\frac{i > 0}{E \vdash i : \text{Pos}}$$

N-INT
$$\frac{i < 0}{E \vdash i : \text{Neg}}$$

ZERO
$$\frac{}{E \vdash 0 : \text{Zero}}$$

- Same for booleans:

TRUE
$$\frac{}{E \vdash \text{true} : \text{True}}$$

FALSE
$$\frac{}{E \vdash \text{false} : \text{False}}$$

What about "if"?

- Two cases are easy:

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

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

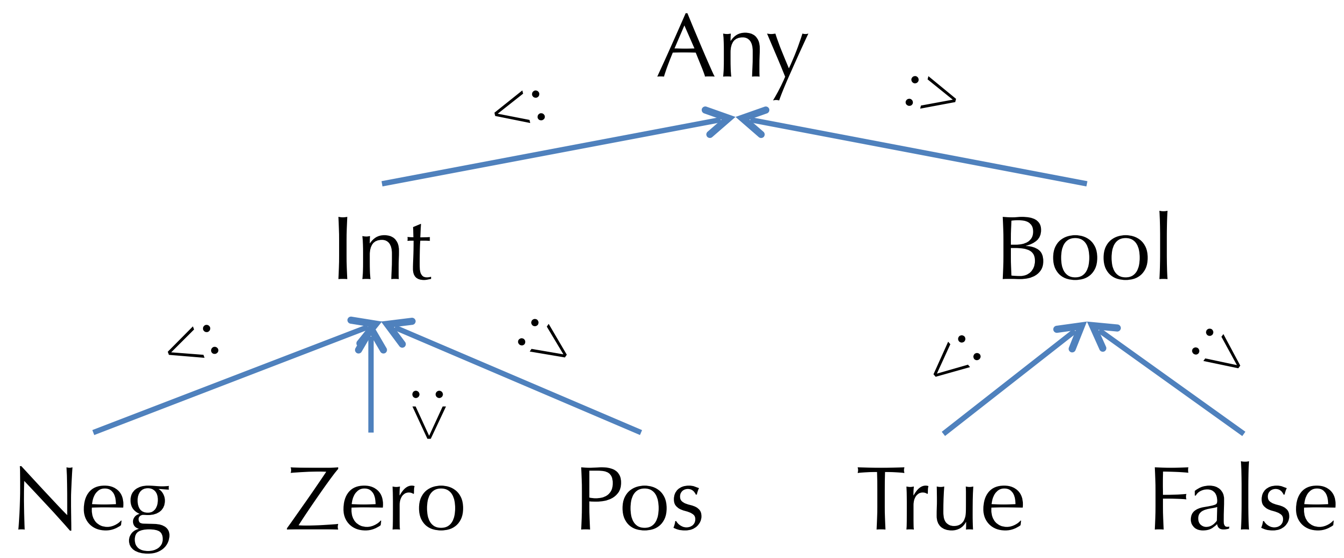
- What if we don't know statically which branch will be taken?
- Consider the typechecking problem:

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

- 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 T1 and T2, 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:

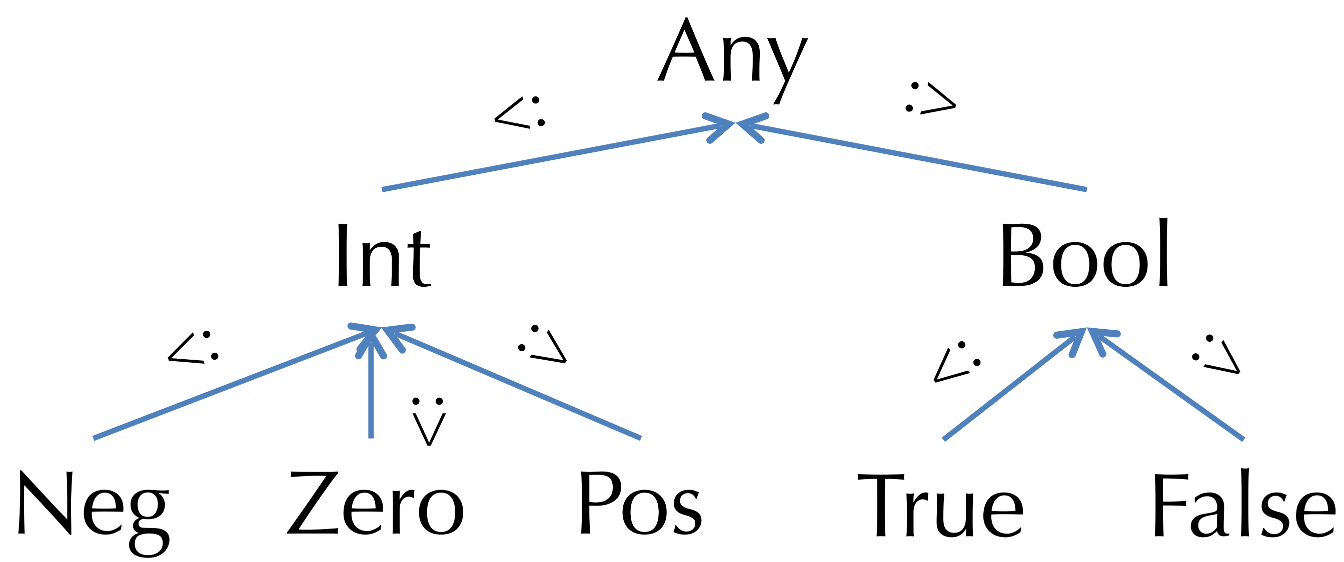
IF-BOOL

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

- Note: LUB(T1, T2) is the most precise type (according to the hierarchy) that describes any value with either type T1 or type T2
- Math notation: LUB(T1, T2) is sometimes written $T1 \vee T2$ or $T1 \sqcup T2$
    - 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: $T1 <: T2$ and $T2 <: T3$ then $T1 <: T3$
    - Antisymmetric: $T1 <: T2$ and $T2 <: T1$ then $T1 = T2$

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 T1 <: T2 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 T1 <: T2 implies $[\![T1]\!] \subseteq [\![T2]\!]$, then T1 <: T2 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:

$$[\![T1]\!] \cup [\![T2]\!] \subseteq [\![LUB(T1, T2)]\!]$$

  - Note that the LUB is an over approximation of the "semantic union"
- Example: $[\![Zero]\!] \cup [\![Pos]\!] = \{0\} \cup \{1,2,3,\ldots\}$

$$= \{0,1,2,3,\ldots\}$$
$$\subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$$
$$= [\![Int]\!] = [\![LUB(Zero, Pos)]\!]$$

- Using LUBs in the typing rules yields sound approximations of the program behavior (as in the IF-B rule).

IF-BOOL

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

Subsumption Rule

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

SUBSUMPTION

$$\frac{E \vdash e : T \qquad T <: S}{E \vdash e : S}$$

- **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.

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"

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

- At runtime, ifPos checks whether e1 is > 0. If so, branches to e2 and otherwise branches to e3
- Inside expression e2, x is e1'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

Extending Subtyping to Other Types

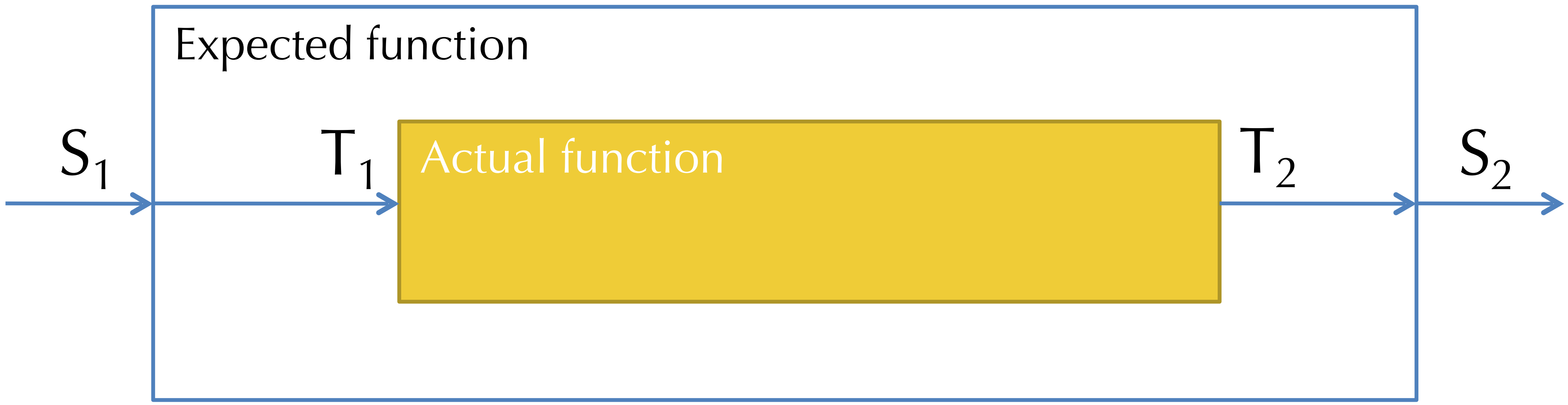
- What about subtyping for tuples?
  - When a program expects a value of type S1 * S2, when is sound to give it a T1 * T2?

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

  - Example: (Pos * Neg) <: (Int * Int)
- What about functions?
  - When is T1 -> T2 <: S1 -> S2 ?
  - When a program expects a function of type S1 -> S2, when can we give it a function of type T1 -> T2 ?

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: {lab1:T1; lab2:T2; … ; labn:Tn}
  - Each labi is a label drawn from a set of identifiers.

RECORD

$$\frac{E \vdash e_1 : T_1 \qquad E \vdash e_2 : T_2 \quad \dots \quad E \vdash e_n : T_n}{E \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \dots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \dots\ ;\ lab_n{:}T_n\}}$$

PROJECTION

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

Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

DEPTH

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

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

WIDTH

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

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. Why?

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:

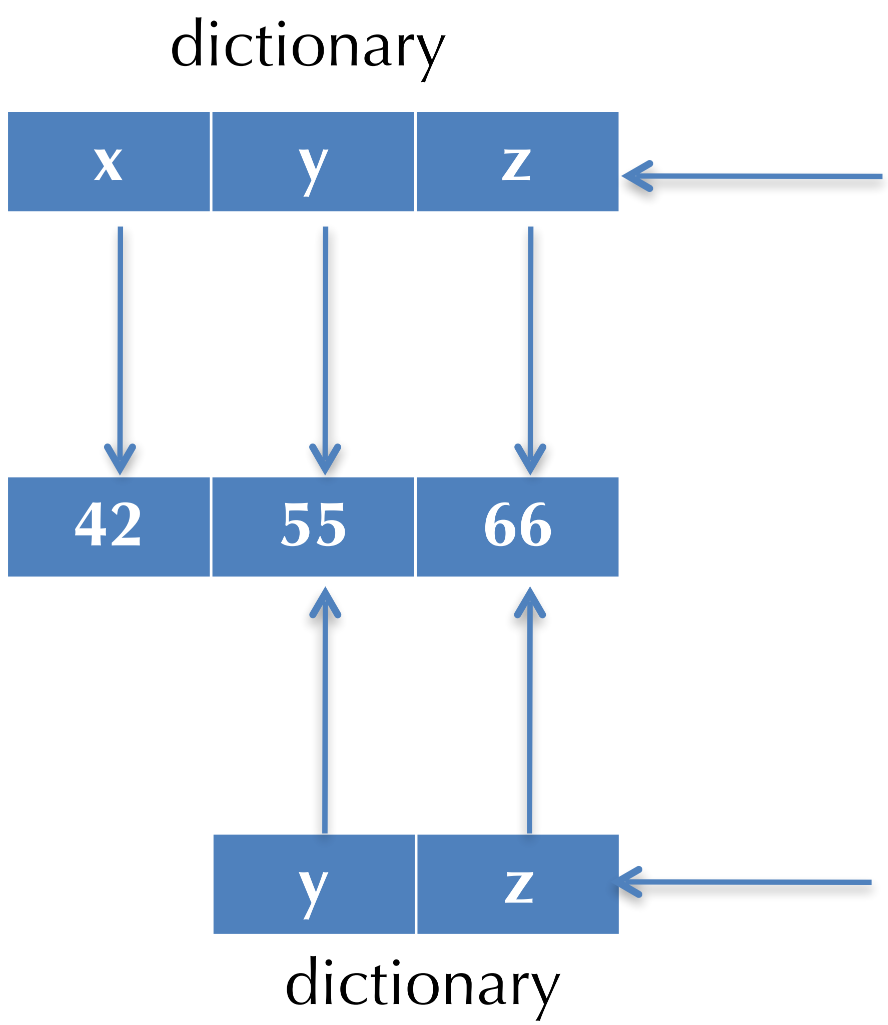
  {x:int; y:int} = {y:int; x:int}

  - 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 and Subtyping

- What about when we add mutable locations?
  - References, arrays, ...

NULL

- What is the type of `null`?
- Consider:
  - `int[] a  = null;` // OK?
  - `int x    = null;` // not OK?
  - `string s = null;` // OK?

$$\frac{}{E \vdash \text{null} : r} \text{ NULL}$$

- 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 T1 <: T2 then ref T2 <: ref T1 is also unsound
  - Exercise: construct a program that breaks contravariant references.
- Moral: Mutable structures are invariant:

  T1 ref <: T2 ref implies T1 = T2
- Same holds for arrays, 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 ref ≃ {get: unit -> T; set: T -> 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.
- {get: unit -> T; set: T -> unit} <: {get: unit -> S; set: S -> unit}
  - get components are subtypes: unit -> T <: unit -> S
  - set components are subtypes: T -> unit <: S -> 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 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 are
                                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 42;}
}
```

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