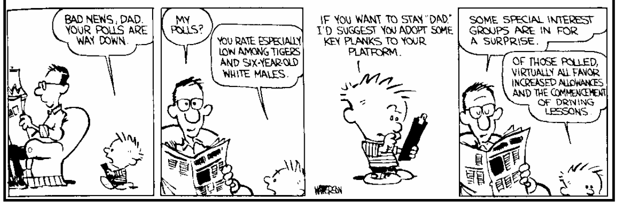
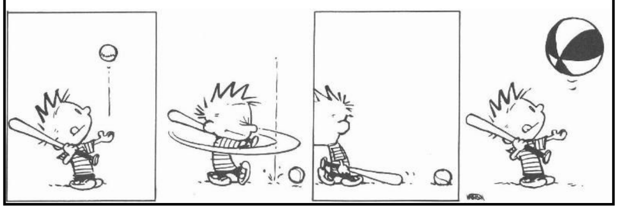
Class Survey #2 Out Today



Subtyping



Introduction to Subtyping

- We can view types as denoting *sets of values*
- Subtyping is a relation between types induced by the *subset relation between value sets*
- Informal intuition:
  - If τ is a subtype of σ then any expression with type τ also has type σ (e.g., $\mathbb{Z} \subseteq \mathbb{R}$, $1 \in \mathbb{Z} \Rightarrow 1 \in \mathbb{R}$)
  - If τ is a subtype of σ then any expression of type τ can be used in a context that expects a σ
  - We write $\tau < \sigma$ to say that τ is a subtype of σ
  - Subtyping is reflexive and transitive

Plan For This Lecture

- Bonus Lecture #2 on Tue Mar 28
  - Usual Suspects get food and drinks?
- Formalize Subtyping Requirements
  - Subsumption
- Create Safe Subtyping Rules
  - Pairs, functions, references, etc.
  - Most easy thing we try will be wrong
- Subtyping Coercions

Subtyping Examples

- FORTRAN introduced int < real
  - 5 + 1.5 is well-typed in many languages
- PASCAL had [1..10] < [0..15] < int
- Subtyping is a fundamental property of object-oriented languages
  - If S is a subclass of C then an instance of S can be used where an instance of C is expected
  - "subclassing ⇒ subtyping" philosophy

Subsumption

- Formalize the requirements on subtyping
- Rule of subsumption
  - If $\tau < \sigma$ then an expression of type τ has type σ

$$\frac{\Gamma \vdash e : \tau \qquad \tau < \sigma}{\Gamma \vdash e : \sigma}$$

- But now type safety may be in danger:
  - If we say that int < (int → int)
  - Then we can prove that "5 5" is well typed!
- There is a way to construct the subtyping relation to preserve type safety

Defining Subtyping

- The formal definition of subtyping is by derivation rules for the judgment $\tau < \sigma$
- We start with subtyping on the base types
  - e.g. int < real or nat < int
  - These rules are language dependent and are typically based directly on types-as-sets arguments
- We then make subtyping a preorder (reflexive and transitive)

$$\frac{}{\tau < \tau} \qquad \frac{\tau_1 < \tau_2 \quad \tau_2 < \tau_3}{\tau_1 < \tau_3}$$

- Then we build-up subtyping for "larger" types

#7

Subtyping for Pairs

- Try $$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \times \tau' < \sigma \times \sigma'}$$
  - Show (informally) that whenever a s × s' can be used, a t × t' can also be used:
  - Consider the context H = H'[fst •] expecting a s × s'
    - Then H' expects a s
    - Because t < s then H' accepts a t
    - Take e : t × t'. Then fst e : t so it works in H'
    - Thus e works in H
  - The case of "snd •" is similar

#8

Subtyping for Records

- Several subtyping relations for records

1. Depth subtyping

$$\frac{\tau_i < \tau_i'}{\{\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\} < \{\, l_1 : \tau_1', \ldots, l_n : \tau_n' \,\}}$$

- e.g., {f1 = int, f2 = int} < {f1 = real, f2 = int}

2. Width subtyping

$$\frac{n \geq m}{\{\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\} < \{\, l_1 : \tau_1, \ldots, l_m : \tau_m \,\}}$$

- E.g., {f1 = int, f2 = int} < {f2 = int}
- Models subclassing in OO languages

3. Or, a combination of the two

#9

Subtyping for Functions

$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

Example Use:
rounded_sqrt : $\mathbb{R} \to \mathbb{Z}$
actual_sqrt : $\mathbb{R} \to \mathbb{R}$
Since $\mathbb{Z} < \mathbb{R}$, rounded_sqrt < actual_sqrt
So if I have code like this:

```
float result = rounded_sqrt(5); // 2
```

… I can replace it like this:

```
float result = actual_sqrt(5); // 2.23
```

… and everything will be fine.

#10

Subtyping for Functions

$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

- What do you think of this rule?

#11

Subtyping for Functions

$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

- This rule is unsound
  - Let Γ = f : int → bool (and assume int < real)
  - We show using the above rule that $\Gamma \vdash$ f 5.0 : bool
  - But this is wrong since 5.0 is *not a valid argument* of f

$$\frac{\dfrac{\Gamma \vdash f : \texttt{int} \to \texttt{bool} \qquad \dfrac{\texttt{int} < \texttt{real} \quad \texttt{bool} < \texttt{bool}}{\texttt{int} \to \texttt{bool} < \texttt{real} \to \texttt{bool}}}{\Gamma \vdash f : \texttt{real} \to \texttt{bool}} \qquad \Gamma \vdash 5.0 : \texttt{real}}{\Gamma \vdash f\ 5.0 : \texttt{bool}}$$

#12

Correct Function Subtyping

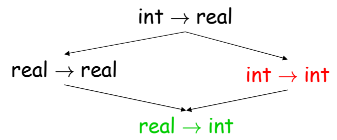
$$\frac{\sigma < \tau \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

- We say that $\to$ is covariant in the result type and contravariant in the argument type
- Informal correctness argument:
  - Pick f : $\tau \to \tau'$
  - f expects an argument of type τ
  - It also accepts an argument of type $\sigma < \tau$
  - f returns a value of type τ'
  - Which can also be viewed as a σ' (since $\tau' < \sigma'$)
  - Hence f can be used as $\sigma \to \sigma'$

#13

More on Contravariance

- Consider the subtype relationships:



- In what sense (f ∈ real → int) ⇒ (f ∈ int → int) ?
  - "real → int" has a *larger domain*!
  - (recall the set theory (arg,result) pair encoding for functions)
- This suggests that "subtype-as-subset" interpretation is not straightforward
  - We'll return to this issue (after these commercial messages ...)

#14

Subtyping References

- Try covariance
$$\frac{\tau < \sigma}{\tau\ \texttt{ref} < \sigma\ \texttt{ref}}$$
Wrong!
  - Example: assume $\tau < \sigma$
  - The following holds (if we assume the above rule):
    x : σ, y : τ ref, f : τ → int ⊢ y := x; f (! y)
  - Unsound: f is called on a σ but is defined only on τ
  - Java has covariant arrays!
- If we want covariance of references we can recover type safety with a runtime check for each y := x
  - The actual type of x matches the actual type of y
  - But this is generally considered a *bad design*

#15

Subtyping References (Part 2)

- Contravariance?
$$\frac{\tau < \sigma}{\sigma\ \texttt{ref} < \tau\ \texttt{ref}}$$
Also Wrong!
  - Example: assume $\tau < \sigma$
  - The following holds (if we assume the above rule):
    x : σ, y : σ ref, f : τ → int ⊢ y := x; f (! y)
  - Unsound: f is called on a σ but is defined only on τ
- References are invariant
  - *No subtyping for references* (unless we are prepared to add run-time checks)
  - hence, *arrays* should be invariant
  - hence, *mutable records* should be invariant

#16

Subtyping Recursive Types

- Recall τ list = μt.(unit + τ×t)
  - We would like τ list < σ list whenever τ < σ
- Covariance?
$$\frac{\tau < \sigma}{\mu t.\tau < \mu t.\sigma}$$
Wrong!
- This is *wrong if t occurs contravariantly in τ*
- Take τ = μt.t→int and σ = μt.t→real
- Above rule says that τ < σ
- We have τ≃τ→int and σ≃σ→real
- τ<σ would mean covariant function type!
- How can we get safe subtyping for lists?

#17

Subtyping Recursive Types

- The correct rule
$$\frac{\begin{array}{c} t < s \\ \vdots \\ \tau < \sigma \end{array}}{\mu t.\tau < \mu s.\sigma}$$
- We add as an *assumption* that the type variables stand for types with the desired subtype relationship
  - Before we assumed they stood for the *same* type!
- Verify that now subtyping works properly for lists
- There is no subtyping between μt.t→int and μt.t→real (recall: $\frac{\tau < \sigma}{\mu t.\tau < \mu t.\sigma}$ Wrong!

#18

Conversion Interpretation

- The subset interpretation of types leads to an abstract modeling of the operational behavior
  - e.g., we say int < real even though an int could not be directly used as a real in the concrete x86 implementation (cf. IEEE 754 bit patterns)
  - The int needs to be converted to a real
- We can get closer to the "machine" with a conversion interpretation of subtyping
  - We say that τ < σ when there is a conversion function that converts values of type τ to values of type σ
  - Conversions also help explain issues such as contravariance
  - Must be careful with conversions (cf. Afghanistan)

#19

Conversions

- Examples:
  - nat < int with conversion λx.x
  - int < real with conversion 2's comp → IEEE
- The subset interpretation is a *special case* when all conversions are *identity functions*
- Write "τ < σ ⇒ C(τ, σ)" to say that C(τ,σ) is the conversion function from subtype τ to σ
  - If C(τ, σ) is expressed in F_1 then C(τ,σ) : τ → σ

#20

Issues with Conversions

- Consider the expression "printreal 1" typed as follows:

$$\dfrac{\texttt{printreal} : \texttt{real} \to \texttt{unit} \qquad \dfrac{1 : \texttt{int} \quad \texttt{int} < \texttt{real}}{1 : \texttt{real}}}{\texttt{printreal}\ 1 : \texttt{unit}}$$

we convert 1 to real: printreal (C(int,real) 1)

- But we can also have another type derivation:

$$\dfrac{\dfrac{\texttt{printreal} : \texttt{real} \to \texttt{unit} \quad \texttt{real} \to \texttt{unit} < \texttt{int} \to \texttt{unit}}{\texttt{printreal} : \texttt{int} \to \texttt{unit}} \qquad 1 : \texttt{int}}{\texttt{printreal}\ 1 : \texttt{unit}}$$

with conversion "(C(real -> unit, int -> unit) printreal) 1"

- Which one is right? What do they mean?

#21

Introducing Conversions

- We can compile a language with subtyping into one without subtyping by introducing conversions
- The process follows closely that of type checking

$$\Gamma \vdash e : \tau \Rightarrow \underline{e}$$

  - Expression e has type τ and its conversion is e
- Rules for the conversion process:

$$\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \Rightarrow \underline{e_1} \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow \underline{e_2}}{\Gamma \vdash e_1\ e_2 : \tau \Rightarrow \underline{e_1}\ \underline{e_2}}$$

$$\dfrac{\Gamma \vdash e : \tau \Rightarrow \underline{e} \qquad \tau < \sigma \Rightarrow C(\tau, \sigma)}{\Gamma \vdash e : \sigma \Rightarrow C(\tau, \sigma)\underline{e}}$$

#22

Coherence of Conversions

- Questions and Concerns:
  - Can we build *arbitrary subtype relations* just because we can write conversion functions?
  - Is real < int just because the "floor" function is a conversion?
  - *What is the conversion* from "real→int" to "int→int"?
- What are the restrictions on conversion functions?
- A system of conversion functions is coherent if whenever we have τ < τ' < σ then
  - C(τ, τ) = λx.x
  - C(τ,σ) = C(τ', σ) ○ C(τ, τ') *(= composed with)*
  - otherwise we end up with confusing uses of subsumption

#23

Example of Coherence

- We want the following subtyping relations:
  - int < real ⇒ λx:int. toIEEE x
  - real < int ⇒ λx:real. floor x
- For this system to be coherent we need
  - C(int, real) ○ C(real, int) = λx.x, and
  - C(real, int) ○ C(int, real) = λx.x
- This means that
  - ∀x : real . (toIEEE (floor x) = x)
  - which is *not true*

#24

Building Conversions

- We start from conversions on basic types

$$\frac{}{\tau < \tau \Rightarrow \lambda x : \tau . x}$$

$$\frac{\tau_1 < \tau_2 \Rightarrow C(\tau_1, \tau_2) \quad \tau_2 < \tau_3 \Rightarrow C(\tau_2, \tau_3)}{\tau_1 < \tau_3 \Rightarrow C(\tau_2, \tau_3) \circ C(\tau_1, \tau_2)}$$

$$\frac{\tau_1 < \sigma_1 \Rightarrow C(\tau_1, \sigma_1) \quad \tau_2 < \sigma_2 \Rightarrow C(\tau_2, \sigma_2)}{\tau_1 \times \tau_2 < \sigma_1 \times \sigma_2 \Rightarrow \lambda x : \tau_1 \times \tau_2 . (C(\tau_1, \sigma_1)(\texttt{fst}(x)), C(\tau_2, \sigma_2)(\texttt{snd}(x)))}$$

$$\frac{}{\tau_1 \times \tau_2 < \tau_1 \Rightarrow \lambda x : \tau_1 \times \tau_2 .\, \texttt{fst}(x)}$$

$$\frac{\sigma_1 < \tau_1 \Rightarrow C(\sigma_1, \tau_1) \quad \tau_2 < \sigma_2 \Rightarrow C(\tau_2, \sigma_2)}{\tau_1 \to \tau_2 < \sigma_1 \to \sigma_2 \Rightarrow \lambda f : \tau_1 \to \tau_2 .\, \lambda x : \sigma_1 .\, C(\tau_2, \sigma_2)(f(C(\sigma_1, \tau_1)(x)))}$$

#25

Comments

- With the conversion view we see why we do not necessarily want to impose antisymmetry for subtyping
  - Can have multiple representations of a type
  - We want to reserve type equality for representation equality
  - $\tau < \tau'$ and also $\tau' < \tau$ (are interconvertible) but not necessarily $\tau = \tau'$
  - e.g., Modula-3 has packed and unpacked records
- We'll encounter subtyping again for object-oriented languages
  - Serious difficulties there due to recursive types

#26

Subtyping in POPL and PLDI 2005

- *A simple typed intermediate language for object-oriented languages*
- *Checking type safety of foreign function calls*
- *Essential language support for generic programming*
- *Semantic type qualifiers*
- *Permission-based ownership*
- *... (out of space on slide)*

#27

Homework

- Project Status Update Due Today
- Class Survey #2 Out Today
- Bonus Lecture #2 On Tuesday

#28