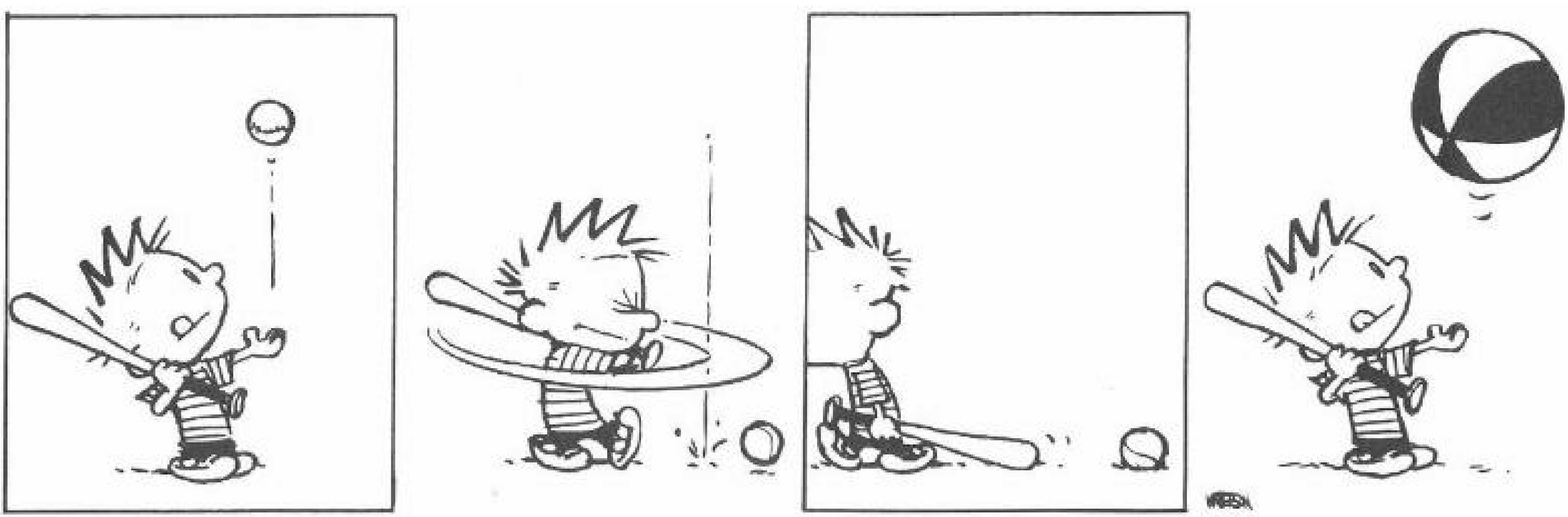
Recursive Types and Subtyping



Recursive Types: Lists

- We want to define recursive data structures
- Example: lists
  - A list of elements of type τ (a τ list) is *either* empty *or* it is a pair of a τ and a τ list

$$\tau \text{ list} = \text{unit} + (\tau \times \tau \text{ list})$$

  - This is a recursive equation. We take its solution to be the smallest set of values L that satisfies the equation

$$L = \{ * \} \cup (T \times L)$$

    where T is the set of values of type τ
  - Another interpretation is that the recursive equation is taken up-to (modulo) set isomorphism

Recursive Types

- We introduce a recursive type constructor μ (mu):

$$\mu t.\ \tau$$

  - The type variable t is bound in τ
  - This stands for the solution to the equation

    $t \simeq \tau$ (t is isomorphic with τ)
  - Example: **τ list** = **$\mu t.\ (\text{unit} + \tau \times t)$**
  - This also allows "unnamed" recursive types
- We introduce syntactic (sugary) operations for the conversion between $\mu t.\tau$ and $[\mu t.\tau/t]\tau$
- e.g. between "τ list" and "unit + ($\tau \times \tau$ list)"

$$e ::= \ldots \quad | \ \text{fold}_{\mu t.\tau}\ e \ | \ \text{unfold}_{\mu t.\tau}\ e$$

$$\tau ::= \ldots \quad | \ t \ | \ \mu t.\tau$$

Example with Recursive Types

- Lists

  τ list = μt. (unit + $\tau \times$ t)

  nil_τ = $fold_{\tau\ list}$ (injl *)

  $cons_\tau$ = $\lambda x{:}\tau.\lambda L{:}\tau$ list. $fold_{\tau\ list}$ injr (x, L)

- A list length function

  $length_\tau$ = $\lambda L{:}\tau$ list.

  case ($unfold_{\tau\ list}$ L) of injl x $\Rightarrow$ 0

  | injr y $\Rightarrow$ 1 + $length_\tau$ (snd y)

- (At home …) Verify that
  - nil_τ : τ list
  - $cons_\tau$: $\tau \to \tau$ list $\to \tau$ list
  - $length_\tau$: τ list $\to$ int

Type Rules for Recursive Types

$$\frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \texttt{unfold}_{\mu t.\tau}\ e : [\mu t.\tau/t]\tau}$$

$$\frac{\Gamma \vdash e : [\mu t.\tau/t]\tau}{\Gamma \vdash \texttt{fold}_{\mu t.\tau}\ e : \mu t.\tau}$$

- The typing rules are syntax directed
- Often, for syntactic simplicity, the fold and unfold operators are omitted
  - This makes type checking somewhat harder

Dynamics of Recursive Types

- We add a new form of values

  $v ::= \ldots \mid \text{fold}_{\mu t.\tau}\ v$

  – The purpose of fold is to ensure that the value has the recursive type and not its unfolding

- The evaluation rules:

$$\frac{e \Downarrow v}{\texttt{fold}_{\mu t.\tau}\ e \Downarrow \texttt{fold}_{\mu t.\tau}\ v} \qquad \frac{e \Downarrow \texttt{fold}_{\mu t.\tau}\ v}{\texttt{unfold}_{\mu t.\tau}\ e \Downarrow v}$$

  - The folding annotations are for type checking only
  - They can be dropped after type checking

Recursive Types in ML

- The language ML uses a simple syntactic trick to avoid having to write the explicit fold and unfold
- In ML recursive types are *bundled with union types*

  type t = C_1 of τ_1 | C_2 of τ_2 | ... | C_n of τ_n

  (* t can appear in τ_i *)

  - e.g., "type intlist = Nil of unit | Cons of int * intlist"
- When the programmer writes Cons (5, l)
  - the compiler treats it as $\text{fold}_{\text{intlist}}$ (injr (5, l))
- When the programmer writes
  - case e of Nil ⇒ ... | Cons (h, t) ⇒ ...

  the compiler treats it as
  - case $\text{unfold}_{\text{intlist}}$ e of Nil ⇒ ... | Cons (h,t) ⇒ ...

Encoding Call-by-Value λ-calculus in F_1^μ

- So far, F_1 was so weak that we could not encode non-terminating computations
  - Cannot encode recursion
  - Cannot write the $\lambda x.x\ x$ (self-application)
- The addition of recursive types makes typed λ-calculus *as expressive as untyped λ-calculus*!
- We could show a conversion algorithm from call-by-value untyped λ-calculus to call-by-value F_1^μ

Untyped Programming in F_1^{μ}

- We write $\underline{e}$ for the conversion of the term e to F_1^{μ}
  - The type of $\underline{e}$ is $V = \mu t.\ t \to t$
- The conversion rules

  $\underline{x} = x$

  $\underline{\lambda x.\ e} = \text{fold}_V\ (\lambda x{:}V.\ \underline{e})$

  $\underline{e_1\ \ e_2} = (\text{unfold}_V\ \underline{e_1})\ \underline{e_2}$
- Verify that
  - $\cdot \vdash \underline{e} : V$
  - $e \Downarrow v$ if and only if $\underline{e} \Downarrow \underline{v}$
- We can express non-terminating computation

  $D = (\text{unfold}_V\ (\text{fold}_V\ (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)))\ (\text{fold}_V\ (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)))$

  or, equivalently

  $D = (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)\ (\text{fold}_V\ (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)))$

Smooth Transition

- And now, on to 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

Cunning Plan For Subtyping

- Formalize Subtyping Requirements
  - Subsumption
- Create Safe Subtyping Rules
  - Pairs, functions, references, etc.
  - Most easy thing we try will be wrong
- Subtyping Coercions
  - When is a subtyping system correct?

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

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)*

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

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

Subtyping for Records

- Several subtyping relations for records
- Depth subtyping

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

  - e.g., {f1 = int, f2 = int} < {f1 = real, f2 = int}
- 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
- Or, a combination of the two

Subtyping for Functions

$$\frac{\tau < \sigma \qquad \tau' < \sigma'}{\tau \rightarrow \tau' < \sigma \rightarrow \sigma'}$$

Example Use:

rounded_sqrt : $\mathbb{R} \rightarrow \mathbb{Z}$

actual_sqrt : $\mathbb{R} \rightarrow \mathbb{R}$

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.

Subtyping for Functions

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

- What do you think of this rule?

Subtyping for Functions

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

- This rule is unsound
  - Let Γ = f : int $\to$ 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

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

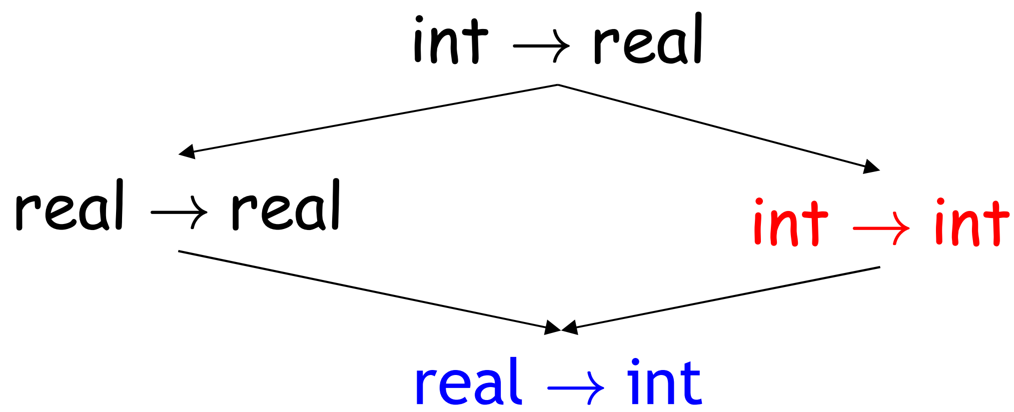
Correct Function Subtyping

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

- 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 $\sigma \to \sigma'$

More on Contravariance

- Consider the subtype relationships:



- In what sense $(f \in \text{real} \to \text{int}) \Rightarrow (f \in \text{int} \to \text{int})$?
  - "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 …)

Subtyping References

- Try covariance

$$\frac{\tau < \sigma}{\tau \ \mathtt{ref} < \sigma \ \mathtt{ref}}$$

Wrong!

  - Example: assume $\tau < \sigma$
  - The following holds (if we assume the above rule):

$$x : \sigma, y : \tau \ \text{ref}, f : \tau \to \text{int} \vdash y := x; f\ (!\ y)$$

  - 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*

Subtyping References (Part 2)

- Contravariance?

$$\frac{\tau < \sigma}{\sigma \ \mathtt{ref} < \tau \ \mathtt{ref}}$$

Also Wrong!

  - Example: assume $\tau < \sigma$
  - The following holds (if we assume the above rule):

$$x : \sigma, y : \sigma \ \text{ref}, f : \tau \to \text{int} \vdash y := x; f\ (!\ y)$$

  - 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

Subtyping Recursive Types

- Recall **τ list = μt.(unit + $\tau\times$t)**
  - We would like τ list < σ list whenever $\tau < \sigma$
- Covariance?

$$\frac{\tau < \sigma}{\mu t.\tau < \mu t.\sigma}$$

Wrong!

- This is *wrong if t occurs contravariantly in τ*
- Take $\tau = \mu t.t \to int$ and $\sigma = \mu t.t \to real$
- Above rule says that $\tau < \sigma$
- We have $\tau \simeq \tau \to int$ and $\sigma \simeq \sigma \to real$
- $\tau < \sigma$ would mean covariant function type!
- How can we get safe subtyping for lists?

Subtyping Recursive Types

- The correct rule

$$\frac{\begin{array}{c} t < s \\ \vdots \\ \tau < \sigma \end{array}}{\mu t.\tau < \mu s.\sigma}$$

Means assume t < s and use that to prove τ < σ

- 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 $\mu t.t \rightarrow int$ and $\mu t.t \rightarrow real$ (recall:

$$\frac{\tau < \sigma}{\mu t.\tau < \mu t.\sigma}$$

Wrong!

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 $\tau < \sigma$ 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
  - But: must be careful with conversions

Conversions

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

Issues with Conversions

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

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

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

- But we can also have another type derivation:

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

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

- Which one is right? What do they mean?

Introducing Conversions

- We can compile a language with subtyping into one without subtyping by introducing conversions
- The process is similar to type checking

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

  - Expression e has type τ and its conversion is $\underline{e}$
- Rules for the conversion process:

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \Rightarrow \underline{e_1} \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow \underline{e_2}}{\Gamma \vdash e_1\ e_2 : \tau \Rightarrow \underline{e_1}\ \underline{e_2}}$$

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

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$\rightarrow$int" to "int$\rightarrow$int"?
- What are the restrictions on conversion functions?
- A system of conversion functions is coherent if whenever we have $\tau < \tau' < \sigma$ then
    - $C(\tau, \tau) = \lambda x.x$
    - $C(\tau,\sigma) = C(\tau', \sigma) \circ C(\tau, \tau')$ *(= composed with)*
    - Example: if b is a bool then (float)b == (float)((int)b)
  - otherwise we end up with confusing uses of subsumption

Example of Coherence

- We want the following subtyping relations:
  - int < real $\Rightarrow$ λx:int. toIEEE x
  - real < int $\Rightarrow$ λx:real. floor x
- For this system to be coherent we need
  - C(int, real) $\circ$ C(real, int) = λx.x, and
  - C(real, int) $\circ$ C(int, real) = λx.x
- This requires that
  - $\forall$x : real . (toIEEE (floor x) = x)
  - which is *not true*

Building Conversions

- We start from conversions on basic types

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

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

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

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

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

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

Homework

- How's that project going?