Recursive Types and Subtyping
One-Slide Summary

- **Recursive types** (e.g., $\tau$ list) make the typed lambda calculus as powerful as the untyped lambda calculus.
- If $\tau$ is a **subtype** of $\sigma$ then any expression of type $\tau$ can be used in a context that expects a $\sigma$; this is called **subsumption**.
- A **conversion** is a function that converts between types.
- A subtyping system should be **coherent**.
Recursive Types: Lists

• We want to define recursive data structures
• Example: lists
  - A list of elements of type \( \tau \) (a \( \tau \) list) is either empty or it is a pair of a \( \tau \) and a \( \tau \) 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 \( \tau \)
- Another interpretation is that the recursive equation is taken up-to (modulo) set isomorphism
Recursive Types

• We introduce a **recursive type constructor** \( \mu \) (mu): 

\[
\mu t. \tau
\]

- The **type variable** \( t \) is bound in \( \tau \)
- This stands for the solution to the equation 
  \[
t \simeq \tau \quad (t \text{ is isomorphic with } \tau)
\]
- Example: \( \tau \text{ list} = \mu t. (\text{unit} + \tau \times t) \)
- 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 “\( \tau \text{ list} \)” and “\( \text{unit} + (\tau \times \tau \text{ list}) \)”

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

\[
\tau ::= \ldots \quad | \quad t \quad | \quad \mu t. \tau
\]
Example with Recursive Types

• Lists
  \[ \tau \text{ list} = \mu t. \ (\text{unit} + \tau \times t) \]
  \[ \text{nil}_\tau = \text{fold}_{\tau \text{ list}} (\text{injl } \ast) \]
  \[ \text{cons}_\tau = \lambda x: \tau. \lambda L: \tau \text{ list}. \ \text{fold}_{\tau \text{ list}} \text{ injr } (x, L) \]

• A list length function
  \[ \text{length}_\tau = \lambda L: \tau \text{ list}. \]
  \[ \text{case } \text{unfold}_{\tau \text{ list}} L \text{ of } \begin{align*}
    \text{injl } x & \Rightarrow 0 \\
    \text{injr } y & \Rightarrow 1 + \text{length}_\tau (\text{snd } y)
  \end{align*} \]

• (At home ...) Verify that
  - \( \text{nil}_\tau : \tau \text{ list} \)
  - \( \text{cons}_\tau : \tau \to \tau \text{ list} \to \tau \text{ list} \)
  - \( \text{length}_\tau : \tau \text{ list} \to \text{int} \)
Type Rules for Recursive Types

\[ \Gamma \vdash e : \mu t.\tau \]

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

\[ \Gamma \vdash e : [\mu t.\tau/t]\tau \]

\[ \Gamma \vdash \text{fold}_{\mu t.\tau} e : \mu t.\tau \]

- 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

\[ v ::= \ldots | \text{fold}_{\mu t.\tau} v \]

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

• The evaluation rules:

\[
\frac{e \Downarrow v}{\text{fold}_{\mu t.\tau} e \Downarrow \text{fold}_{\mu t.\tau} v}
\]

\[
\frac{e \Downarrow \text{fold}_{\mu t.\tau} v}{\text{unfold}_{\mu t.\tau} e \Downarrow v}
\]

• 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**

\[
\text{type } t = C_1 \text{ of } \tau_1 \mid C_2 \text{ of } \tau_2 \mid \ldots \mid C_n \text{ of } \tau_n
\]

\[(*) \text{ t can appear in } \tau_i (*)\]

- e.g., “type intlist = Nil of unit | Cons of int * intlist”

• When the programmer writes \(\text{Cons (5, l)}\)
  - the compiler treats it as \(\text{fold}_{\text{intlist}} (\text{injr (5, l)})\)

• When the programmer writes
  - case e of Nil ⇒ … | Cons (h, t) ⇒ …
  the compiler treats it as
  - case unfold_{intlist} e of Nil ⇒ … | Cons (h,t) ⇒ …
Encoding Call-by-Value

\( \lambda \)-calculus in F_1^{\mu}

- 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 \( \lambda \)-calculus as expressive as untyped \( \lambda \)-calculus!
- We could show a conversion algorithm from call-by-value untyped \( \lambda \)-calculus to call-by-value F_1^{\mu}
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 $\tau$ is a subtype of $\sigma$ then any expression with type $\tau$ also has type $\sigma$ (e.g., $\mathbb{Z} \subseteq \mathbb{R}$, $1 \in \mathbb{Z} \Rightarrow 1 \in \mathbb{R}$)
  - If $\tau$ is a subtype of $\sigma$ then any expression of type $\tau$ can be used in a context that expects a $\sigma$
  - We write $\tau < \sigma$ to say that $\tau$ is a subtype of $\sigma$
  - 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 \( \text{int} < \text{real} \)
  - \( 5 + 1.5 \) is well-typed in many languages

• PASCAL had \([1..10] < [0..15] < \text{int}\)

• 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
  - “\text{subclassing } \Rightarrow \text{ subtyping}” philosophy
Subsumption

• Formalize the requirements on subtyping

• Rule of subsumption
  - If $\tau < \sigma$ then an expression of type $\tau$ has type $\sigma$

\[
\Gamma \vdash e : \tau \quad \tau < \sigma \\
\hline
\Gamma \vdash e : \sigma
\]

• But now type safety may be in danger:
  • If we say that int $<$ (int $\rightarrow$ int)
  • Then we can prove that “11 8” is well typed!

• There is a way to construct the subtyping relation to preserve type safety
Subtyping in POPL/PLDI 14

- Backpack: Retrofitting Haskell with Interfaces
- Getting F-Bounded Polymorphism into Shape
- Optimal Inference of Fields in Row-Polymorphic Records
- Polymorphic Functions with Set-Theoretic Types (Part 1: Syntax, Semantics, and Evaluation)
- ... (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)

$$
\begin{align*}
\tau_1 < \tau_2 & \quad \tau_2 < \tau_3 \\
\tau_1 < \tau_3 & \\
\tau_1 < \tau & \\
\tau < \tau &
\end{align*}
$$

• 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 \( s \times s' \) can be used, a \( t \times t' \) can also be used:

  - Consider the context \( H = H'[\text{fst } \bullet] \) expecting a \( s \times s' \)
    - Then \( H' \) expects a \( s \)
    - Because \( t < s \) then \( H' \) accepts a \( t \)
    - Take \( e : t \times t' \). Then \( \text{fst } e : t \) so it works in \( H' \)
    - Thus \( e \) works in \( H \)
  - The case of “\( \text{snd } \bullet \)” is similar
Subtyping for Records

• Several subtyping relations for records

  • **Depth** subtyping
    \[ \tau_i < \tau'_i \]
    \[
    \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} < \{ l_1 : \tau'_1, \ldots, l_n : \tau'_n \}
    \]
    
    • e.g., \{f1 = int, f2 = int\} < \{f1 = real, f2 = int\}

  • **Width** subtyping
    \[ n \geq m \]
    \[
    \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} < \{ l_1 : \tau_1, \ldots, l_m : \tau_m \}
    \]
    
    • E.g., \{f1 = int, f2 = int\} < \{f2 = int\}
    • Models subclassing in OO languages

• Or, a **combination** of the two
Subtyping for Functions

\[ \tau < \sigma \quad \tau' < \sigma' \]

\[ \tau \rightarrow \tau' < \sigma \rightarrow \sigma' \]

Example Use:

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

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

Since \( \mathbb{Z} < \mathbb{R} \), \( \text{rounded}_\text{sqrt} < \text{actual}_\text{sqrt} \)

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.
This numerical technique for finding solutions to boundary-value problems was initially developed for use in structural analysis in the 1940's. The subject is represented by a model consisting of a number of linked simplified representations of discrete regions. It is often used to determine stress and displacement in mechanical systems.
Computer Science

• This American Turing-award winner is known for his visionary and pioneering contributions to Computer Graphics, and for Sketchpad, an early predecessor to the GUI. He created the first virtual reality display, and a graphics line clipping algorithm. His students include Alan Kay (Smalltalk), Henri Gouraud (shading), Frank Crow (anti-aliasing), and Edwin Catmull (Pixar). When asked, "How could you possibly have done the first interactive graphics program, the first non-procedural programming language, the first object oriented software system, all in one year?" He replied: "Well, I didn't know it was hard."
Subtyping for Functions

\[ \tau < \sigma \quad \tau' < \sigma' \]

\[ \tau \to \tau' < \sigma \to \sigma' \]

• What do you think of this rule?
Subtyping for Functions

\[ \tau < \sigma \quad \tau' < \sigma' \]

\[ \tau \rightarrow \tau' < \sigma \rightarrow \sigma' \]

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

\[
\Gamma \vdash f : \text{int} \rightarrow \text{bool} \quad \text{int} \rightarrow \text{bool} < \text{real} \rightarrow \text{bool} \\
\Gamma \vdash f : \text{real} \rightarrow \text{bool} \\
\Gamma \vdash 5.0 : \text{real} \\
\Gamma \vdash f \ 5.0 : \text{bool}
\]
Correct Function Subtyping

\[ \sigma < \tau \quad \tau' < \sigma' \]

\[ \tau \rightarrow \tau' < \sigma \rightarrow \sigma' \]

- We say that \( \rightarrow \) is **covariant** in the result type and **contravariant** in the argument type.

- Informal correctness argument:
  - Pick \( f : \tau \rightarrow \tau' \)
  - \( f \) expects an argument of type \( \tau \)
  - It also accepts an argument of type \( \sigma < \tau \)
  - \( f \) returns a value of type \( \tau' \)
  - Which can also be viewed as a \( \sigma' \) (since \( \tau' < \sigma' \))
  - Hence \( f \) can be used as \( \sigma \rightarrow \sigma' \)
More on Contravariance

• Consider the subtype relationships:

\[
\begin{align*}
\text{int} &\rightarrow \text{real} \\
\text{real} &\rightarrow \text{real} \\
\text{real} &\rightarrow \text{int} \\
\text{int} &\rightarrow \text{int}
\end{align*}
\]

• In what sense \((f \in \text{real} \rightarrow \text{int}) \Rightarrow (f \in \text{int} \rightarrow \text{int})\) ?
  • “\text{real} \rightarrow \text{int}” has a \emph{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

\[
\tau < \sigma \\
\frac{\tau \text{ ref} < \sigma \text{ ref}}{\text{Wrong!}}
\]

- Example: assume \( \tau < \sigma \)
- The following holds (if we assume the above rule):
  \[
x : \sigma, \ y : \tau \text{ ref}, \ f : \tau \rightarrow \text{int} \leftarrow y := x; f (! y)
\]
- Unsound: \( f \) is called on a \( \sigma \) but is defined only on \( \tau \)
- 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 \textit{bad design}
Subtyping References (Part 2)

- Contravariance?
  \[ \tau < \sigma \]
  \[ \sigma \text{ ref} < \tau \text{ ref} \]
  
  - Example: assume \( \tau < \sigma \)
  - The following holds (if we assume the above rule):
    \[ x : \sigma, y : \sigma \text{ ref}, f : \tau \rightarrow \text{int} \vdash y := x; f (! y) \]
  - Unsound: \( f \) is called on a \( \sigma \) but is defined only on \( \tau \)

- 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 $\tau \text{ list} = \mu t.(\text{unit} + \tau \times t)$
  - We would like $\tau \text{ list} < \sigma \text{ list}$ whenever $\tau < \sigma$
- Covariance?
  $\tau < \sigma$  
  $\mu t.\tau < \mu t.\sigma$

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

• The correct rule

\[ t < s \]
\[ \vdash \]
\[ \tau < \sigma \]
\[ \mu t.\tau < \mu s.\sigma \]

\( \begin{cases} \text{Means assume } t < s \\ \text{and use that to} \\ \text{prove } \tau < \sigma \end{cases} \)

• 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 \( \mu t.t \rightarrow \text{int} \) and \( \mu t.t \rightarrow \text{real} \) (recall:

\[ \tau < \sigma \]
\[ \mu t.\tau < \mu t.\sigma \]

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 $\tau$ to values of type $\sigma$
  - 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 \( \tau \) to \( \sigma \)
  - 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:

\[
\begin{align*}
\text{printreal} & : \text{real} \rightarrow \text{unit} \\
1 &: \text{real} \\
\text{printreal} \ 1 &: \text{unit}
\end{align*}
\]

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

• But we can also have another type derivation:

\[
\begin{align*}
\text{printreal} & : \text{real} \rightarrow \text{unit} \\
\text{real} \rightarrow \text{unit} & < \text{int} \rightarrow \text{unit} \\
\text{printreal} & : \text{int} \rightarrow \text{unit} \\
1 &: \text{int} \\
\text{printreal} \ 1 &: \text{unit}
\end{align*}
\]

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

\[ \Gamma \vdash e : \tau \Rightarrow e \]

- Expression e has type \( \tau \) and its conversion is \( e \)

• Rules for the conversion process:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \Rightarrow e_1 \\
\Gamma \vdash e_2 : \tau_2 \Rightarrow e_2 \\
\hline
\Gamma \vdash e_1 e_2 : \tau \Rightarrow e_1 e_2
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau \Rightarrow e \\
\tau < \sigma \Rightarrow C'(\tau, \sigma) \\
\hline
\Gamma \vdash e : \sigma \Rightarrow C'(\tau, \sigma)e
\end{align*}
\]
Coherence of Conversions

• Questions and Concerns:
  - Can we build *arbitrary subtype relations* just because we can write conversion functions?
  - Is \( \text{real} < \text{int} \) 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 \( \tau < \tau' < \sigma \) then
  
  \[
  \begin{align*}
  C(\tau, \tau) &= \lambda x. x \\
  C(\tau, \sigma) &= C(\tau', \sigma) \circ C(\tau, \tau') \quad (= \text{composed with})
  \end{align*}
  \]

  - Example: if \( b \) is a \text{bool} then \((\text{float})b == (\text{float})((\text{int})b)\)
  - otherwise we end up with confusing uses of subsumption
Example of Coherence

• We want the following subtyping relations:
  - int < real ⇒ \( \lambda x:\text{int}. \text{toIEEE} \ x \)
  - real < int ⇒ \( \lambda x:\text{real}. \text{floor} \ x \)

• For this system to be coherent we need
  - \( C(\text{int, real}) \circ C(\text{real, int}) = \lambda x. x \), and
  - \( C(\text{real, int}) \circ C(\text{int, real}) = \lambda x. x \)

• This requires that
  - \( \forall x : \text{real} \ . \ ( \text{toIEEE} (\text{floor} x) = x ) \)
  - which is not true
Building Conversions

• We start from conversions on basic types

\[
\begin{align*}
\tau < \tau & \Rightarrow \lambda x : \tau. x \\
\tau_1 < \tau_2 & \Rightarrow C(\tau_1, \tau_2) \quad \tau_2 < \tau_3 & \Rightarrow C(\tau_2, \tau_3) \\
\tau_1 < \tau_3 & \Rightarrow C(\tau_2, \tau_3) \circ C(\tau_1, \tau_2) \\
\tau_1 < \sigma_1 & \Rightarrow C'(\tau_1, \sigma_1) \quad \tau_2 < \sigma_2 & \Rightarrow C'(\tau_2, \sigma_2) \\
\tau_1 \times \tau_2 < \sigma_1 \times \sigma_2 & \Rightarrow \lambda x : \tau_1 \times \tau_2. (C(\tau_1, \sigma_1)(\text{fst}(x)), C(\tau_2, \sigma_2)(\text{snd}(x))) \\
\tau_1 \times \tau_2 < \tau_1 & \Rightarrow \lambda x : \tau_1 \times \tau_2. \text{fst}(x) \\
\sigma_1 < \tau_1 & \Rightarrow C'(\sigma_1, \tau_1) \quad \tau_2 < \sigma_2 & \Rightarrow C'(\tau_2, \sigma_2) \\
\tau_1 \rightarrow \tau_2 < \sigma_1 \rightarrow \sigma_2 & \Rightarrow \lambda f : \tau_1 \rightarrow \tau_2. \lambda x : \sigma_1. C(\tau_2, \sigma_2)(f(C(\sigma_1, \tau_1)(x)))
\end{align*}
\]
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?