Recursive Types
Subtyping

Lecture 17
CS263

Recursive Types

- It is useful to be able 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
    τ list = unit + ($\tau \times \tau$ list)
  - This is a recursive equation. We take its solution to be the smallest set of values L that satisfy the equation
    $L = \{*\} \cup (T \times L)$
    where T is the set of values of type τ
  - Note: this interpretation can be troublesome
    - E.g. $\tau = \tau \to \tau$, but only for trivial sets we have $T = T \to T$
  - Another interpretation is that the recursive equation is up-to set isomorphism

Recursive Types

- We introduce a recursive type constructor
  $\mu t.\ \tau$
  - The type variable t is bound in τ
  - This is the solution to the equation
    $t \simeq \tau$ (t is isomorphic with τ)
  - E.g., τ list = $\mu t.$ (unit + $\tau \times t$)
  - This allows "unnamed" recursive types
- We introduce syntactic 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 ::= … | $fold_{\mu t.\tau}$ e | $unfold_{\mu t.\tau}$ e
  τ ::= … | t | $\mu t.\tau$

Example with Recursive Types

- Lists
  τ list = $\mu t.$ (unit + $\tau \times t$)
  $nil_\tau = fold_{\tau\ list}$ (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)
- Verify that
  - nil_τ : τ list
  - $cons_\tau$: $\tau \to \tau$ list $\to \tau$ list
  - $length_\tau$: τ list $\to$ int

Static Semantics of 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 ::= … | $fold_{\mu t.\tau}$ 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
  datatype t = C_1 of τ_1 | C_2 of τ_2 | ... | C_n of τ_n (t can appear in τ_i)
  - E.g., datatype intlist = Nil of unit | Cons of int $\times$ intlist
- When the programmer writes
  Cons (5, l)
  - the compiler treats it as
    $fold_{intlist}$ (injlr (5, l))
- When the programmer writes
  - case e of Nil $\Rightarrow$ ... | Cons (h, t) $\Rightarrow$ ...
  the compiler treats it as
  - case $unfold_{intlist}$ e of Nil $\Rightarrow$ ... | Cons (h,t) $\Rightarrow$ ...

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 can 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 $V = \mu t.\ t \rightarrow t$
- The conversion rules
  $\underline{x}$ = x
  $\underline{\lambda x.\ e}$ = $fold_V\ (\lambda x{:}V.\ \underline{e})$
  $\underline{e_1\ e_2}$ = $(unfold_V\ \underline{e_1})\ \underline{e_2}$
- Verify that
  1. $\cdot \vdash \underline{e} : V$
  2. $e \Downarrow v$ if and only if $\underline{e} \Downarrow \underline{v}$
- We can express non-terminating computation
  $D = (unfold_V\ (fold_V\ (\lambda x{:}V.\ (unfold_V\ x)\ x)))\ (fold_V\ (\lambda x{:}V.\ (unfold_V\ x)\ x)))$
  or, equivalently
  $D = (\lambda x{:}V.\ (unfold_V\ x)\ x)\ (fold_V\ (\lambda x{:}V.\ (unfold_V\ x)\ x)))$

Subtyping

Introduction to Subtyping

- Viewing types as denoting sets of values, it is natural to consider a subtyping relation between types as induced by the subset relation between sets
- Informal intuition:
  - If τ is a subtype of σ then any expression with type τ also has type σ
  - If τ is a subtype of σ then any expression of type τ can be used in a context that expects a σ
  - Subtyping is reflexive and transitive
  - We write $\tau < \sigma$ to say that τ is a subtype of σ

Subtyping Examples

- FORTRAN introduced int < real
  - 5 + 1.5 is well-typed in many languages
- PASCAL had [1..10] < [0..15] < int
- It is generally accepted that subtyping is a fundamental property of object-oriented languages
  - Let S be a subclass of C. Then an instance of S can be used where an instance of C is expected
  - This is "subclassing $\Rightarrow$ subtyping" philosophy

Subsumption

- We formalize the informal requirement on subtyping
- Rule of subsumption
  - If $\tau < \sigma$ then an expression of type τ also has type σ

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

- But now type safety is in danger:
  - If we say that int < int $\rightarrow$ 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

Subtyping for Pairs

- Try $$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \times \tau' < \sigma \times \sigma'}$$
- Show (informally) that whenever a $\sigma \times \sigma'$ can be used, a $\tau \times \tau'$ can also be used:
- Consider the context H = H'[fst •] expecting a $\sigma \times \sigma'$
  - Then H' expects a σ
  - Because $\tau < \sigma$ then H' accepts a τ
  - Take e : $\tau \times \tau'$. Then fst e : τ so it works in H'
  - Thus e works in H
- The case of "snd •" is similar

Subtyping for Records

- Several subtyping relations make sense for records

1. 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}

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

Subtyping for Functions

- Try the (naive) rule

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

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

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

Subtyping for Functions (Cont.)

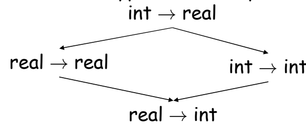
- The correct rule

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

- We say that $\rightarrow$ is covariant in the result type and contravariant in the argument type
- Informal correctness argument:
  - Pick f : $\tau \rightarrow \tau'$ and show it can be used with type $\sigma \rightarrow \sigma'$
  - 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 $\sigma \rightarrow \sigma'$

More on Contravariance

- Consider the subtype relationships



- In what sense f ∈ real → int ⇒ f ∈ int → int?
  - "real → int" has a larger domain!
- This suggests that "subtype-as-subset" interpretation is not straightforward
  - We'll get back to this issue

Subtyping References

- Try covariance

$$\frac{\tau < \sigma}{\tau\ \texttt{ref} < \sigma\ \texttt{ref}} \qquad \text{Wrong!}$$

  - Example: assume τ < σ
  - 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
  - Some consider this a bad design

Subtyping References (Cont.)

- Try contravariance:

$$\frac{\tau < \sigma}{\sigma\ \texttt{ref} < \tau\ \texttt{ref}} \qquad \text{Also Wrong!}$$

  - Example: assume τ < σ
  - 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 <u>invariant</u> with respect to subtyping
  - 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 + τ×t)
  - We would like τ list < σ list whenever τ < σ
- Try simple covariance:

$$\frac{\tau < \sigma}{\mu t.\tau < \mu t.\sigma} \qquad \text{Wrong!}$$

- 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 still have the subtyping for lists?

Subtyping Recursive Types (Cont.)

- 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 that they stand for the <u>same</u> type!
- Verify that subtyping now works properly for lists
- There is no subtyping between μt.t→int and μt.t→real

Conversion Interpretation of Subtyping

- 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 implementation
  - The int needs to be <u>converted</u> 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

Conversions

- Examples:
  - nat < int with conversion λx.x
  - int < real with conversion from 2's complement to IEEE format
- The subset interpretation is a special case when all conversions are the identity functions
- Definition: we 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 $C(\tau,\sigma) : \tau \to \sigma$

Issues with Conversions

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

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

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

- But we can also have another type derivation:

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

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

- Which one is right ?

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 $\underline{e}$
- Rules for the conversion process:

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \Rightarrow \underline{e_1} \quad \Gamma \vdash e_2 : \tau_2 \Rightarrow \underline{e_2}}{\Gamma \vdash e_1\ e_2 : \tau \Rightarrow \underline{e_1}\ \underline{e_2}}$$

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

Coherence of Conversions

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

Coherence of Conversions. Example

- Say that 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 means 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 might want to allow both $\tau < \sigma$ and $\sigma < \tau$
  - 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