Subtyping

Meeting 27, CSCI 5535, Spring 2010

Announcements

• Homework 6
  - 42.0 mean, 9.4 stddev
  - 44 median, 48.25 3rd quartile, 50 max

• Homework 7
  - 35.9 mean, 11.3 stddev
  - 38 median, 40.5 3rd quartile, 48 max

• Overall Homeworks
  - 81.0 mean, 20.3 stddev
  - 84.8 median, 90.1 3rd quartile, 94.5 max

Announcements

• FCQ This Thu!
  - Please come give me feedback
  - Volunteer to administer?

• CAETE students
  - Online FCQ

Poll: What do you want to discuss Thu?

• Continue with Types
  - Imperative Features: References 0
  - Exceptions and Continuations 2
  - Parametric Polymorphism / Generics / Universal Types 5
  - Data Abstraction / Modules / Existential Types and Dependent Types 4

• Shape Analysis
  - Precise program analysis for the heap 7

Review of Monomorphic Type Systems
### General PL Feature Plan

- The general plan for language feature design
- You **invent** a new feature (e.g., sums)
- You **add** it to the **lambda calculus**
- You **invent** typing rules and opsem rules
- You **extend** the basic **proof of type safety**
- You declare moral victory, and milling throngs of cheering admirers wait to carry you on their shoulders to be knighted by the Queen, etc.

### Sums (Tagged Unions)

- **We need disjoint union types** of the form:
  - either an int or a float
  - either 0 or a pointer
  - either a (binary tree node with two children) or a (leaf)
- **New expressions and types**
  
  \[
  e ::= \ldots \mid \text{injl } e \mid \text{injr } e \mid \tag
  \]

  \[
  \tau ::= \ldots \mid \tau_1 + \tau_2
  \]

  - A value of type \(\tau_1 + \tau_2\) is either \(\tau_1\) or \(\tau_2\)

### Static Semantics of Sums

- **New typing rules**

  \[
  \begin{align*}
  & \Gamma \vdash e : \tau_1 \\
  & \Gamma \vdash \text{injl } e : \tau_1 + \tau_2 \\
  & \Gamma \vdash \text{injr } e : \tau_1 + \tau_2 \\
  & \Gamma, x : \tau_1 + \tau_2 \vdash e : \tau \\
  & \Gamma, y : \tau_1 + \tau_2 \vdash e : \tau \\
  & \Gamma \vdash \text{case } e \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_2
  \end{align*}
  \]

- **Types are not unique anymore**
  - \(\text{injl } 1 : \text{int + bool}\)
  - \(\text{injl } 1 : \text{int + (int \to int)}\)

  - this complicates type checking, but it is still doable

### Dynamic Semantics of Sums

- **New values**
  
  \[
  v ::= \ldots \mid \text{injl } v \mid \text{injr } v
  \]

- **New evaluation rules**

  \[
  \begin{align*}
  & e \Downarrow v \\
  & \text{injl } e \Downarrow \text{injl } v \\
  & \text{injr } e \Downarrow \text{injr } v \\
  & e \Downarrow \text{injl } v \\
  & [v/x]e_l \Downarrow v' \\
  & \text{case } e \text{ of } \text{injl } x \Rightarrow e_l \mid \text{injr } y \Rightarrow e_r \Downarrow v' \\
  & e \Downarrow \text{injr } v \\
  & [v/y]e_r \Downarrow v' \\
  & \text{case } e \text{ of } \text{injl } x \Rightarrow e_l \mid \text{injr } y \Rightarrow e_r \Downarrow v'
  \end{align*}
  \]
Type Soundness for \( F_{\tau} \)

- Type soundness **still holds**
- No way to use a \( \tau_1 + \tau_2 \) inappropriately
- The key is that the **only way** to use a \( \tau_1 + \tau_2 \) is with case, which ensures that you are not using a \( \tau_1 \) as a \( \tau_2 \)
- In C or Pascal checking the tag is the responsibility of the programmer!
  - Unsafe

End of Review

On to Subtyping

What is subtyping?

- Subtyping: 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 \( \sigma \)
  - 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 \( \tau \leq \sigma \) to say that \( \tau \) is a subtype of \( \sigma \)

One-Slide Summary

- 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**.

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., \( \mathbb{Z} \subseteq \mathbb{R}, 1 \in \mathbb{Z} \Rightarrow 1 \in \mathbb{R} \))
  - We write \( \tau \leq \sigma \) to say that \( \tau \) is a subtype of \( \sigma \)
- Subtyping is reflexive and transitive

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
  - "subclassing ⇒ subtyping" philosophy

Subsumption

- Formalize the requirements on subtyping
- Rule of subsumption
  - If \( : \sigma \) then an expression of type \( \tau \) has type \( \sigma \)
    \[
    \Gamma \vdash e : \tau \\
    \tau <: \sigma \\
    \Gamma \vdash e : \sigma
    \]
  - But now type safety may be in danger:
    - If we say that \( \text{int} <: (\text{int} \to \text{int}) \)
    - Then we can prove that "11 8" 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 inference rules for the judgment \( \tau <: \sigma \)
- We start with subtyping on the base types
  - e.g. \( \text{int} <: \text{real} \) or \( \text{nat} <: \text{int} \)
  - These rules are language dependent and are typically based directly on types-as-sets arguments
- We then make subtyping a preorder (reflexive and transitive)
  \[
  \begin{align*}
  \tau & <: \tau \\
  \tau_1 & <: \tau_2 \\
  \tau_2 & <: \tau_3 \\
  \tau_1 & <: \tau_3
  \end{align*}
  \]
- Then we build-up subtyping for "larger" types

Subtyping for Pairs

- Why is it the case that whenever a \( \sigma \times \sigma' \) can be used, a \( \tau \times \tau' \) can also be used?
  - Consider the context \( H = H'(\text{fst} *) \) expecting a \( \sigma \times \sigma' \)
    - Then \( H' \) expects a \( \sigma \)
    - Because \( : \sigma \) then \( H' \) accepts a \( \tau \)
    - Take \( e : \tau \times \tau' \). Then \( \text{fst} e : \sigma \) so it works in \( H' \)
    - Thus \( e \) works in \( H \)
  - The case of "snd *" is similar

Subtyping for Records

- Thoughts?

\[
\begin{align*}
\xi, \eta : \tau_1, \ldots, \xi_m : \tau_m \quad \xi, \eta : \xi_1, \ldots, \xi_m : \xi_m \quad \xi_1 : \tau_1, \ldots, \xi_m : \tau_m
\end{align*}
\]
Subtyping for Records

- Several subtyping relations for records
  - **Depth** subtyping
    \[ \{l_1 : \tau_1, \ldots, l_n : \tau_n\} < \{l_1 : \tau'_1, \ldots, l_n : \tau'_n\} \]
    - e.g., \{f1 : int, f2 : int\} < \{f1 : real, f2 : int\}
  - **Width** subtyping
    \[ \{l_1 : \tau_1, \ldots, l_n : \tau_n\} < \{l_1 : \tau_1', \ldots, l_m : \tau_m\} \]
    - e.g., \{f1 : int, f2 : int\} < \{f2 : int\}
    - Models subclassing in OO languages
  - Or, a combination of the two

Subtyping for Functions

- Example Use:
  - `rounded_sqrt : R → Z`
  - `actual_sqrt : R → R`
  - Since \(Z <: R\), `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".

Correct Function Subtyping

- 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
  \[
  \begin{array}{ccc}
  \text{int} & \rightarrow & \text{real} \\
  \text{real} & \rightarrow & \text{int} \\
  \text{int} & \rightarrow & \text{int}
  \end{array}
  \]
- In what sense \((f : \text{real} \rightarrow \text{int}) \Rightarrow (f : \text{int} \rightarrow \text{int})\)?
  - "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 ...)
References

- Such types are used for mutable memory cells
- Syntax (as in ML)
  
  \[
  \begin{align*}
  e ::= & \ldots & | \text{ref} \; e :: \tau | e_1 := e_2 | ! e \\
  \tau ::= & \ldots & | \text{ref} \tau
  \end{align*}
  \]

  - ref e :: \tau evaluates e, allocates a new memory cell, stores the value of e in it and returns the address of the memory cell.
  - e_1 := e_2 evaluates e_1 to a memory cell and updates its value with the value of e_2.
  - ! e evaluates e to a memory cell and returns its contents.

  - Like malloc + initialization in C, or new in C++ and Java.

Subtyping References

Contravariance? \( \tau <: \sigma \) \[ \sigma \text{ ref} \text{ ref} \) \[ \text{Wrong!} \]

- Example: assume \( \tau <: \sigma \)
- The following holds (if we assume the above rule):
  
  \[
  \begin{align*}
  x : \sigma, y : \text{ref}, f : \tau \rightarrow \text{int} & \vdash y := x; f(y) : \text{int} \\
  \end{align*}
  \]

  - 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 bad design.

Conversions

- Examples:
  
  - nat <: int with conversion \( \lambda x. x \)
  - int <: real with conversion \( 2s \text{ comp} \rightarrow \text{IEEE} \)

  - The subset interpretation of types leads to an abstract modeling of the operational behavior.
  - For example, 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.

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.

Subtyping References

- Example: assume \( \tau <: \sigma \)
- The following holds (if we assume the above rule):
  
  \[
  \begin{align*}
  x : \sigma, y : \text{ref}, f : \tau \rightarrow \text{int} & \vdash y := x; f(y) : \text{int} \\
  \end{align*}
  \]

  - 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 bad design.

Conversions

- Examples:
  
  - nat <: int with conversion \( \lambda x. x \)
  - int <: real with conversion \( 2s \text{ comp} \rightarrow \text{IEEE} \)

  - 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:
  \[
  \text{printreal : real } \rightarrow \text{unit} \\
  \text{printreal 1 : unit}
  \]
  we convert 1 to real: printreal (C(int,real) 1)

- But we can also have another type derivation:
  \[
  \text{printreal : real } \rightarrow \text{unit} \rightarrow \text{unit} \\
  \text{printreal : int } \rightarrow \text{unit} \rightarrow \text{unit} \\
  \text{printreal 1 : unit}
  \]
  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

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

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

- Rules for the conversion process:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \Rightarrow e_1 \\
\Gamma \vdash e_2 : \tau_2 & \Rightarrow e_2 \\
\Gamma \vdash e_1 e_2 : \tau & \Rightarrow e_1 e_2 \\
\Gamma \vdash e : \tau & \Rightarrow e \\
\tau & \ll \sigma \Rightarrow C(\tau, \sigma)
\end{align*}
\]

Coherence of Conversions

- Questions and Concerns:
  - Can we build arbitrary subtype relations just because we can write conversion functions?
  - Is \( \text{real <: int} \) 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?

Example of Coherence

- Consider the following subtyping relations:
  - \text{int <: real } \Rightarrow \lambda x : \text{int}. \text{toIEEE } x
  - \text{real <: int } \Rightarrow \lambda x : \text{real}. \text{floor } x

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

- 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

\[
\begin{align*}
\tau & \ll \tau \Rightarrow \lambda x : \tau. x \\
\tau_1 & \ll \tau_2 \Rightarrow \lambda x : \tau_1. \text{toIEEE } x \\
\tau_1 & \ll \tau_2 \Rightarrow \lambda x : \tau_1. \text{floor } x \\
\tau_1 \times \tau_2 & \ll \tau_1 \times \tau_2 \Rightarrow \lambda x : \tau_1 \times \tau_2. \text{floor}(x) \\
\tau_1 \times \tau_2 & \ll \tau_1 \times \tau_2 \Rightarrow \lambda x : \tau_1 \times \tau_2. \text{toIEEE}(x)
\end{align*}
\]
Comments

- With the conversion view we see why we do not necessarily want to impose anti-symmetry 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 \( \tau = \tau' \)

- We’ll encounter subtyping again for object-oriented languages
  - Serious difficulties there due to recursive types

Next Time

- How’s the project going?

Types for Imperative Features

- So far: types for pure functional languages
- Now: types for imperative features
- Such types are used to characterize non-local effects
  - assignments
  - exceptions
- Contextual semantics is useful here
  - Just when you thought it was safe to forget it …

References

- Such types are used for mutable memory cells
- Syntax (as in ML)
  
  \[
  e ::= \ldots | \text{ref } e :: \tau | e_1 := e_2 | ! e \\
  \tau ::= \ldots | \text{ref}
  \]
  
  - ref e :: \tau - evaluates e, allocates a new memory cell, stores the value of e in it and returns the address of the memory cell
  - like malloc + initialization in C, or new in C++ and Java
  - e_1 :: e_2, evaluates e_1 to a memory cell and updates its value with the value of e_2
  - ! e - evaluates e to a memory cell and returns its contents

Global Effects, Reference Cells

- A reference cell can escape the static scope where it was created
  \( \lambda f:\text{int} \to \text{int ref}. !(f 5) \) \( \lambda x:\text{int}. \text{ref } x : \text{int} \)
- The value stored in a reference cell must be visible from the entire program
- The "result" of an expression must now include the changes to the heap that it makes (cf. IMP’s opsem)
- To model reference cells we must extend the evaluation model
Modeling References

- A heap is a mapping from addresses to values
  \( h ::= \cdot | h, a \leftarrow v : \tau \)
- Addresses, tag the heap cells with their types
- Types are useful only for static semantics. They are not needed for the evaluation, that is, are not a part of the implementation

- We call a program an expression with a heap
  \( p ::= \text{heap } h \text{ in } e \)
- The initial program is “heap \( \cdot \) in e”
- Heap addresses act as bound variables in the expression
- This is a trick that allows easy reuse of properties of local variables for heap addresses
  - e.g., we can rename the address and its occurrences at will

Static Semantics of References

- Rules for expressions:
  \[ \Gamma \vdash e : \tau \]
  \[ \Gamma \vdash e : \tau \text{ ref} \]
  \[ \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref} \]
  \[ \Gamma \vdash e_1 : \tau \text{ ref} \]
  \[ \Gamma \vdash e_2 : \tau \]
- Rules for programs (new judgment):
  \[ \Gamma \vdash v_i : \tau_i \text{ (i = 1 . . . n)} \]
  \[ \Gamma \vdash e : \tau \]
  \[ \vdash \text{heap } h \text{ in } e : \tau \]
  where \( \Gamma = a_1 : \tau_1 \text{ ref}, \ldots, a_n : \tau_n \text{ ref} \)
  and \( h = a_1 \leftarrow v_1 : \tau_1, \ldots, a_n \leftarrow v_n : \tau_n \)

Contextual Semantics for References

- Addresses are values:
  \( v ::= \ldots | a \)
- New contexts:
  \[ H ::= \text{ref } H | H_1 ::= e_2 | a_2 ::= H_2 | H \]
- No new local reduction rules
- But some new global reduction rules
  - heap in \( H[\text{ref } v : \tau] \rightarrow \text{heap } h, a \leftarrow v : \tau \text{ in } H[a] \)
  - heap in \( H[v] \rightarrow \text{heap } h \text{ in } H[v] \)
  - where \( a \) is fresh (this models allocation - the heap is extended)
  - heap in \( H[a] \rightarrow \text{heap } h \text{ in } H[v] \)
  - where \( a \rightarrow v : \tau \text{ in } h \text{ is replaced by } a \leftarrow v : \tau \) (heap lookup - can we get stuck?)
  - heap in \( H[a] \rightarrow \text{heap } h \text{ in } H[v] \text{ in } H'[\tau] \)
  - where \( h[a \leftarrow v] \) means a heap like \( h \) except that the part \( a \leftarrow v : \tau \) in \( h \) is replaced by \( a \leftarrow v : \tau \) (memory update)
- Global rules are used to propagate the effects of a write to the entire program (eval order matters!)

Example with References

- Consider these (the redex is underlined)
  \[ \text{heap } \text{ in } (\lambda f : \text{int } \rightarrow \text{int} \text{ ref } ((f 5)) \text{ ref } x : \text{int}) \]
  \[ \rightarrow \text{heap } \text{ in } (((f \text{ ref } x : \text{int}) \text{ ref } 5)) \]
  \[ \rightarrow \text{heap } \text{ in } (\text{ref } 5 : \text{int}) \]
  \[ \rightarrow \text{heap } a = 5 : \text{int} \text{ in } H[a] \]
  \[ \rightarrow \text{heap } a = 5 : \text{int} \text{ in } H[a] \]
- The resulting program has a useless memory cell
- An equivalent result would be
  \[ \text{heap } \text{ in } 5 \]
- This is a simple way to model garbage collection