Lecture 17

CIS 341: COMPILERS
Announcements

• HW5: OAT v. 2.0
  – records, function pointers, type checking, array-bounds checks, etc.
  – Due: Wednesday, April 11th
  – Available soon!
Beyond describing “structure”… describing “properties”
Types as sets
Subsumption

TYPES, MORE GENERALLY
What are types, anyway?

• A type is just a predicate on the set of values in a system.
  – For example, the type “int” can be thought of as a boolean function that
    returns “true” on integers and “false” otherwise.
  – Equivalently, we can think of a type as just a subset of all values.

• For efficiency and tractability, the predicates are usually taken to be
  very simple.
  – Types are an abstraction mechanism

• We can easily add new types that distinguish different subsets of
  values:

```plaintext
type tp =
  | IntT (* type of integers *)
  | Post | NegT | ZeroT (* refinements of ints *)
  | BoolT (* type of booleans *)
  | TrueT | FalseT (* subsets of booleans *)
  | AnyT (* any value *)
```
Modifying the typing rules

- We need to refine the typing rules too…
- Some easy cases:
  - Just split up the integers into their more refined cases:

  \[
  \begin{align*}
  &\text{P-INT} & i > 0 \\
  & & \quad \quad E \vdash i : \text{Pos} \\
  &\text{N-INT} & i < 0 \\
  & & \quad \quad E \vdash i : \text{Neg} \\
  &\text{ZERO} & \quad \quad E \vdash 0 : \text{Zero}
  \end{align*}
  \]

- Same for booleans:

  \[
  \begin{align*}
  &\text{TRUE} \\
  & \quad \quad E \vdash \text{true} : \text{True} \\
  &\text{FALSE} \\
  & \quad \quad E \vdash \text{false} : \text{False}
  \end{align*}
  \]
What about “if”?

- Two cases are easy:
  
  \[ \begin{align*}
  \text{IF-T} & \quad E \vdash e_1 : \text{True} \quad E \vdash e_2 : T \\
  \text{IF-F} & \quad E \vdash e_1 : \text{False} \quad E \vdash e_3 : T
  \end{align*} \]
  
  \[ E \vdash \text{if} \ (e_1) \ e_2 \ \text{else} \ e_3 : T \]

- What happens when we don’t know statically which branch will be taken?

- Consider the typechecking problem:

  \[ x : \text{bool} \vdash \text{if} \ (x) \ 3 \ \text{else} \ -1 : ? \]

- The true branch has type Pos and the false branch has type Neg.
  - What should be the result type of the whole if?
Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: \( \text{Pos} \subseteq \text{Int} \)
- This subset relation gives rise to a \textit{subtype} relation: \( \text{Pos} <: \text{Int} \)
- Such inclusions give rise to a \textit{subtyping hierarchy}:

```
Subtyping Hierarchy:

Any ←───> Int
      ↑  ↓
     Neg  Zero  Pos
     ↓  ↓
    True  False
```

- Given any two types \( T_1 \) and \( T_2 \), we can calculate their \textit{least upper bound} (LUB) according to the hierarchy.
  - Example: \( \text{LUB(True, False)} = \text{Bool} \), \( \text{LUB(Int, Bool)} = \text{Any} \)
  - Note: might want to add types for “NonZero”, “NonNegative”, and “NonPositive” so that set union on values corresponds to taking LUBs on types.
“If” Typing Rule Revisited

• For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

\[ E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2 \]

\[ E \vdash \text{if} (e_1) \ e_2 \ \text{else} \ e_3 : \text{LUB}(T_1, T_2) \]

• Note that LUB\((T_1, T_2)\) is the most precise type (according to the hierarchy) that is able to describe any value that has either type \(T_1\) or type \(T_2\).

• In math notation, LUB\((T_1, T_2)\) is sometimes written \(T_1 \vee T_2\).

• LUB is also called the \textit{join} operation.
Subtyping Hierarchy

• A subtyping hierarchy:

\[
\begin{align*}
\text{Any} & \quad \text{Int} & \quad \text{Bool} \\
\text{Neg} & \quad \text{Zero} & \quad \text{Pos} & \quad \text{True} & \quad \text{False}
\end{align*}
\]

• The subtyping relation is a partial order:
  – Reflexive: \( T <: T \) for any type \( T \)
  – Transitive: \( T_1 <: T_2 \) and \( T_2 <: T_3 \) then \( T_1 <: T_3 \)
  – Antisymmetric: If \( T_1 <: T_2 \) and \( T_2 <: T_1 \) then \( T_1 = T_2 \)
Soundness of Subtyping Relations

• We don’t have to treat every subset of the integers as a type.
  – e.g., we left out the type NonNeg

• A subtyping relation $T_1 <: T_2$ is sound if it approximates the underlying semantic subset relation.

• Formally: write $\llbracket T \rrbracket$ for the subset of (closed) values of type $T$
  – i.e. $\llbracket T \rrbracket = \{ v | \vdash v : T \}$
  – e.g. $\llbracket \text{Zero} \rrbracket = \{0\}$, $\llbracket \text{Pos} \rrbracket = \{1, 2, 3, \ldots\}$

• If $T_1 <: T_2$ implies $\llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket$, then $T_1 <: T_2$ is sound.
  – e.g. Pos $<$: Int is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  – e.g. Int $<$: Pos is not sound, since it is not the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$
Soundness of LUBs

- Whenever you have a sound subtyping relation, it follows that:
  \[
  \llbracket \text{LUB}(T_1, T_2) \rrbracket \supseteq \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket
  \]
  - Note that the LUB is an over approximation of the “semantic union”
  - Example: \[
  \llbracket \text{LUB}(\text{Zero}, \text{Pos}) \rrbracket = \llbracket \text{Int} \rrbracket = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq
    \{0,1,2,3,\ldots\} = \llbracket \text{Zero} \rrbracket \cup \llbracket \text{Pos} \rrbracket
  \]

- Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule).
- It just so happens that LUBs on types <: Int correspond to +

\[
E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad T_1 <: \text{Int} \quad T_2 <: \text{Int}
\]

\[
E \vdash e_1 + e_2 : T_1 \lor T_2
\]
• When we add subtyping judgments of the form \( T <: S \) we can uniformly integrate it into the type system generically:

\[
E \vdash e : T \quad T <: S
\]

\[
E \vdash e : S
\]

• Subsumption allows any value of type \( T \) to be treated as an \( S \) whenever \( T <: S \).

• Adding this rule makes the search for typing derivations more difficult – this rule can be applied anywhere, since \( T <: T \).
  – But careful engineering of the typing system can incorporate the subsumption rule into a deterministic algorithm. (See, e.g., the OAT type system)
Downcasting

• What happens if we have an Int but need something of type Pos?
  – At compile time, we don’t know whether the Int is greater than zero.
  – At run time, we do.

• Add a “checked downcast”

\[ \text{E} \vdash e_1 : \text{Int} \quad \text{E}, x : \text{Pos} \vdash e_2 : T_2 \quad \text{E} \vdash e_3 : T_3 \]

\[ \text{E} \vdash \text{ifPos} (x = e_1) e_2 \text{ else } e_3 : T_2 \lor T_3 \]

• At runtime, ifPos checks whether \( e_1 \) is > 0. If so, branches to \( e_2 \) and otherwise branches to \( e_3 \).

• Inside the expression \( e_2 \), \( x \) is the name for \( e_1 \)’s value, which is known to be strictly positive because of the dynamic check.

• Note that such rules force the programmer to add the appropriate checks
  – We could give integer division the type: \( \text{Int} \to \text{NonZero} \to \text{Int} \)
SUBTYPING OTHER TYPES
Extending Subtyping to Other Types

• What about subtyping for tuples?
  – Intuition: whenever a program expects something of type $S_1 \times S_2$, it is sound to give it a $T_1 \times T_2$.
  – Example: (Pos * Neg) <: (Int * Int)

• What about functions?

• When is $T_1 \rightarrow T_2$ <: $S_1 \rightarrow S_2$?
Subtyping for Function Types

- One way to see it:

\[
\begin{align*}
S_1 &\quad T_1 &\quad T_2 &\quad S_2 \\
\text{Expected function} & & & \text{Actual function} \\
\end{align*}
\]

- Need to convert an \( S_1 \) to a \( T_1 \) and \( T_2 \) to \( S_2 \), so the argument type is \textit{contravariant} and the output type is \textit{covariant}.

\[
S_1 <: T_1 \quad T_2 <: S_2
\]

\[
(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)
\]
Immutable Records

- Record type: \{lab_1:T_1; lab_2:T_2; \ldots; lab_n:T_n\}
  - Each lab_i is a label drawn from a set of identifiers.

\begin{align*}
  \text{RECORD} & \quad E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad \ldots \quad E \vdash e_n : T_n \\
  \quad & \quad E \vdash \{lab_1 = e_1; lab_2 = e_2; \ldots; lab_n = e_n\} : \{lab_1:T_1; lab_2:T_2; \ldots; lab_n:T_n\} \\
  \text{PROJECTION} & \quad E \vdash e : \{lab_1:T_1; lab_2:T_2; \ldots; lab_n:T_n\} \\
  \quad & \quad E \vdash e.lab_i : T_i
\end{align*}
Immutability Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

  \[
  T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n
  \]

  \[
  \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_n:T_n\} <: \{\text{lab}_1:U_1; \text{lab}_2:U_2; \ldots; \text{lab}_n:U_n\}
  \]

- Width subtyping:
  - Subtype record may have more fields:

  \[
  m \leq n
  \]

  \[
  \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_n:T_n\} <: \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_m:T_m\}
  \]
Depth & Width Subtyping vs. Layout

- Width subtyping (without depth) is compatible with "inlined" record representation as with C structs:

\[
\{x: \text{int}; y: \text{int}; z: \text{int}\} <: \{x: \text{int}; y: \text{int}\}
\]

[Width Subtyping]

- The layout and underlying field indices for 'x' and 'y' are identical.
- The 'z' field is just ignored

- Depth subtyping (without width) is similarly compatible, assuming that the space used by A is the same as the space used by B whenever A <: B
- But... they don't mix without
Immutable Record Subtyping (cont’d)

• Width subtyping assumes an implementation in which order of fields in a record matters:
  \{x:int; y:int\} \neq \{y:int; x:int\}

• But: \{x:int; y:int; z:int\} <: \{x:int; y:int\}
  – Implementation: a record is a struct, subtypes just add fields at the end of the struct.

• Alternative: allow permutation of record fields:
  \{x:int; y:int\} = \{y:int; x:int\}
  – Implementation: compiler sorts the fields before code generation.
  – Need to know all of the fields to generate the code

• Permutation is not directly compatible with width subtyping:
  \{x:int; z:int; y:int\} = \{x:int; y:int; z:int\} \n\:<\: \{y:int; z:int\}
If you want both:

- If you want permutability & dropping, you need to either copy (to rearrange the fields) or use a dictionary like this:

\[
p = \{x=42; y=55; z=66\}:\{x: \text{int}; y: \text{int}; z: \text{int}\}
\]

\[
q : \{y: \text{int}; z: \text{int}\} = p
\]
MUTABILITY & SUBTYPING
• What is the type of `null`?
• Consider:
  ```java
  int[] a = null; // OK?
  int x   = null; // not OK?
  string s = null; // OK?
  ```

• Null has any *reference type*
  – Null is generic

• What about type safety?
  – Requires defined behavior when dereferencing `null`
    e.g. Java's `NullPointerException`
  – Requires a safety check for every dereferencence operation
    (typically implemented using low-level hardware "trap" mechanisms.)
Subtyping and References

• What is the proper subtyping relationship for references and arrays?

• Suppose we have NonZero as a type and the division operation has type:  Int -> NonZero -> Int
  – Recall that NonZero <: Int

• Should  (NonZero ref) <: (Int ref)  ?

• Consider this program:

```
Int bad(NonZero ref r) {
  Int ref a = r;  (* OK because (NonZero ref <: Int ref*)
  a := 0;        (* OK because 0 : Zero <: Int *
  return (42 / !r) (* OK because !r has type NonZero *)
}
```
Mutable Structures are Invariant

• Covariant reference types are unsound
  – As demonstrated in the previous example
• Contravariant reference types are also unsound
  – i.e. If $T_1 <: T_2$ then $\text{ref } T_2 <: \text{ref } T_1$ is also unsound
  – Exercise: construct a program that breaks contravariant references.

• Moral: Mutable structures are invariant:
  
  $T_1 \text{ ref } <: T_2 \text{ ref }$ implies $T_1 = T_2$

• Same holds for arrays, OCaml-style mutable records, object fields, etc.
  – Note: Java and C# get this wrong. They allows covariant array subtyping,
    but then compensate by adding a dynamic check on every array update!
Another Way to See It

• We can think of a reference cell as an immutable record (object) with two functions (methods) and some hidden state:
  \[
  \text{T ref} \equiv \{\text{get: unit -> T; set: T -> unit}\}
  \]
  – get returns the value hidden in the state.
  – set updates the value hidden in the state.

• When is T ref <: S ref?
• Records are like tuples: subtyping extends pointwise over each component.
• \{get: unit -> T; set: T -> unit\} <: \{get: unit -> S; set: S -> unit\}
  – get components are subtypes: \(\text{unit -> T} \triangleleft \text{unit -> S}\)
  – set components are subtypes: \(\text{T -> unit} \triangleleft \text{S -> unit}\)
• From get, we must have T <: S (covariant return)
• From set, we must have S <: T (contravariant arg.)
• From T <: S and S <: T we conclude T = S.
STRUCTURAL VS. NOMINAL TYPES
Structural vs. Nominal Typing

• Is type equality / subsumption defined by the structure of the data or the name of the data?
• Example 1: type abbreviations (OCaml) vs. “newtypes” (a la Haskell)

(* OCaml: *)

```ocaml
type cents = int (* cents = int in this scope *)
type age = int

let foo (x:cents) (y:age) = x + y
```

(* Haskell: *)

```haskell
newtype Cents = Cents Integer (* Integer and Cents arr isomorphic, not identical. *)
newtype Age = Age Integer

foo :: Cents -> Age -> Int
foo x y = x + y (* Ill typed! *)
```

• Type abbreviations are treated “structurally”
  Newtypes are treated “by name”
Nominal Subtyping in Java

- In Java, Classes and Interfaces must be named and their relationships explicitly declared:

```java
/* Java: */
interface Foo {
    int foo();
}

class C {
    /* Does not implement the Foo interface */
    int foo() {return 2;}
}

class D implements Foo {
    int foo() {return 341;}
}
```

- Similarly for inheritance: programmers must declare the subclass relation via the “extends” keyword.
  - Typechecker still checks that the classes are structurally compatible
See oat.pdf in HW5

OAT'S TYPE SYSTEM
OAT's Treatment of Types

- Primitive (non-reference) types:
  - int, bool
- Definitely non-null reference types: R
  - (named) mutable structs with width subtyping
  - strings
  - arrays (including length information, per HW4)
- Possibly-null reference types: R?
  - Subtyping: R <: R?
  - Checked downcast syntax if?:

```java
int sum(int[]? arr) {
    var z = 0;
    if?(int[] a = arr) {
        for(var i = 0; i<length(a); i = i + 1;) {
            z = z + a[i];
        }
    }
    return z;
}
```
OAT Features

- Named structure types with mutable fields
  - but using structural, width subtyping

- Typed function pointers

- Polymorphic operations: `length` and `== / !=`
  - need special case handling in the typechecker

- Type-annotated null values: `t null` always has type `t?`

- Definitely-not-null values means we need an "atomic" array initialization syntax
  - for example, null is not allowed as a value of type `int[]`, so to construct a record containing a field of type `int[]`, we need to initialize it
OAT "Returns" Analysis

- Typesafe, statement-oriented imperative languages like OAT (or Java) must ensure that a function (always) returns a value of the appropriate type.
  - Does the returned expression's type match the one declared by the function?
  - Do all paths through the code return appropriately?

- OAT's statement checking judgment
  - takes the expected return type as input: what type should the statement return (or void if none)
  - produces a boolean flag as output: does the statement definitely return?