Lecture 16

COMPILER DESIGN
• **HW4**: OAT v. 1.0
  - Parsing & basic code generation
Subtyping and Upper Bounds

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

\[
\begin{array}{c}
\text{Any} \\
\text{Int} \quad \text{Bool} \\
\text{Neg} \quad \text{Zero} \quad \text{Pos} \\
\text{True} \quad \text{False}
\end{array}
\]

- For types \( T_1, T_2 \), define their least upper bound (LUB) wrt the hierarchy
  - Example: \( \text{LUB}(\text{True}, \text{False}) = \text{Bool} \), \( \text{LUB}(\text{Int}, \text{Bool}) = \text{Any} \)
  - Note: may want to add types for “NonZero”, “NonNegative”, “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 LUB\((T_1, T_2)\) is the most precise type (wrt 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 \(T_1 \lor T_2\)
• LUB is also called the \textit{join} operation
• A subtyping hierarchy

- 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 \([T]\) for the subset of (closed) values of type \( T \)
  – i.e. \([T]\) = \( \{v \mid \vdash v : T\} \)
  – e.g. \([\text{Zero}]\) = \{0\}, \([\text{Pos}]\) = \{1, 2, 3, …\}

• If \( T_1 <: T_2 \) implies \([T_1]\) \( \subseteq\) \([T_2]\), then \( T_1 <: T_2 \) is sound
  – Ex 1: \( \text{Pos} <: \text{Int} \) is sound, since \{1, 2, 3, …\} \( \subseteq \) \{…,-3,-2,-1,0,1,2,3,…\}
  – Ex 2: \( \text{Int} <: \text{Pos} \) is not sound, since it is **not** the case that
    \{…,-3,-2,-1,0,1,2,3,…\} \( \subseteq \) \{1,2,3,…\}
Soundness of LUBs

- Whenever we 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 \(<: \text{Int}\) correspond to +

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

- When we add subtyping judgments of the form $T <: S$ we can uniformly integrate it into the type system generically

\[
\text{SUBSUMPTION} \quad E \vdash e : T \quad T <: S \\
______________________________________________
E \vdash e : S
\]

- 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 for example the OAT type system

Zhendong Su  Compiler Design
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”

\[
\begin{align*}
E \vdash e_1 : \text{Int} & \quad E, x : \text{Pos} \vdash e_2 : T_2 & \quad E \vdash e_3 : T_3 \\
\hline
E \vdash \text{ifPos} (x = e_1) e_2 \text{ else } e_3 : T_2 \lor T_3
\end{align*}
\]

• 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 such rules force the programmer to add the appropriate checks
  – We could give integer division the type: \( \text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int} \)
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: $(\text{Pos} \times \text{Neg}) <: (\text{Int} \times \text{Int})$

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

\[(T_1 \times T_2) <: (S_1 \times S_2)\]

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

- One way to see it

- Need to convert an $S_1$ to a $T_1$ and $T_2$ to $S_2$, so the argument type is *contravariant* and the output type is *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

\[
\text{RECORD} \quad E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad \ldots \quad E \vdash e_n : T_n
\]

\[
E \vdash \{\text{lab}_1 = e_1; \text{lab}_2 = e_2; \ldots ; \text{lab}_n = e_n\} : \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots ; \text{lab}_n:T_n\}
\]

\[
\text{PROJECTION} \quad E \vdash e : \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots ; \text{lab}_n:T_n\}
\]

\[
E \vdash e.\text{lab}_i : T_i
\]
Immutable 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

  \[
  \text{WIDTH} \quad m \leq n
  \]

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

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

\{x:int; y:int; z:int\} <: \{x:int; y: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 well
• Width subtyping assumes an implementation where order of fields in a record matters
  \{\text{x:int; y:int}\} \neq \{\text{y:int; x:int}\}

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

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

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

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

```
p = {x=42; y=55; z=66}:{x:int; y:int; z:int}
q : {y:int; z: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 dereference 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, the division operation has type
  \( \text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int} \)
  – Recall that \( \text{NonZero} <: \text{Int} \)
• Should \((\text{NonZero ref}) <: (\text{Int ref})\)  

• Consider this program

```plaintext
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 *)
}
```

Zhendong Su    Compiler Design
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

Assume: $\text{NonZero} <: \text{Int} \Rightarrow \text{ref } \text{Int} <: \text{ref } \text{NonZero}$

```plaintext
Int ref a;
a := 0;
NonZero ref b;
b = a;
return (1 / b);
```
Mutable Structures are Invariant

- 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!

Assume: \( B <: A \Rightarrow B[] <: A[] \)

```java
B[] b = new B[5];
A[] a = b;
a[0] = new A;
b[0].something in b
```
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:

\[ \text{T ref } \approx \{ \text{get: unit }\rightarrow \text{T; set: T }\rightarrow \text{unit} \} \]

– get returns the value hidden in the state
– set updates the value hidden in the state

• When is \( \text{T ref <: S ref} \)?
• Records, like tuples, subtyping extends pointwise over each component

\[ \{ \text{get: unit }\rightarrow \text{T; set: T }\rightarrow \text{unit} \} <: \{ \text{get: unit }\rightarrow \text{S; set: S }\rightarrow \text{unit} \} \]

– get components are subtypes: \( \text{unit }\rightarrow \text{T }<: \text{unit }\rightarrow \text{S} \)
– set components are subtypes: \( \text{T }\rightarrow \text{unit }<: \text{S }\rightarrow \text{unit} \)

• From get, we must have \( \text{T }<: \text{S} \) (covariant return)
• From set, we must have \( \text{S }<: \text{T} \) (contravariant arg.)
• From \( \text{T }<: \text{S} \) and \( \text{S }<: \text{T} \) we conclude \( \text{T }= \text{S} \)
STRUCTURAL VS. NOMINAL TYPES
Structural vs. Nominal Typing

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

(* OCaml: *)
type cents = int (* cents = int in this scope *)
type age = int

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

(* Haskell: *)
newtype Cents = Cents Integer (* Integer and Cents are 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
  – 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
  – Type-checker still checks the classes are structurally compatible
OAT'S TYPE SYSTEM

Full details later in HW5
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 \textbf{if}?

```
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: \texttt{length}, and \texttt{==} or \texttt{!=}
  – need special case handling in the type-checker

• Type-annotated null values: \texttt{t null} always has type \texttt{t?}

• Definitely-not-null values => "atomic" array initialization syntax

• As an example
  – null is not allowed as a value of type \texttt{int[]}  
  – So to construct a record containing a field of type \texttt{int[]}, need to initialize it
Typesafe, statement-oriented imperative languages like OAT (or Java) must ensure a function (always) returns a value of the appropriate type
  – Does the returned expr'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?
COMPILING CLASSES AND OBJECTS
Code Generation for Objects

• Classes
  – Generate data structure types
    • For objects that are instances of the class and for the class tables
  – Generate the class tables for dynamic dispatch

• Methods
  – Method body code is similar to functions/closures
  – Method calls require dispatch

• Fields
  – Issues are the same as for records
  – Generating access code

• Constructors
  – Object initialization

• Dynamic Types
  – Checked downcasts
  – “instanceof” and similar type dispatch
Multiple Implementations

- The same interface can be implemented by multiple classes

```java
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

class IntSet1 implements IntSet {
    private List<Integer> rep;
    public IntSet1() {
        rep = new LinkedList<Integer>();
    }

    public IntSet1 insert(int i) {
        rep.add(new Integer(i));
        return this;
    }

    public boolean has(int i) {
        return rep.contains(new Integer(i));
    }

    public int size() {return rep.size();}
}

class IntSet2 implements IntSet {
    private Tree rep;
    private int size;
    public IntSet2() {
        rep = new Leaf(); size = 0;
    }

    public IntSet2 insert(int i) {
        Tree nrep = rep.insert(i);
        if (nrep != rep) {
            rep = nrep; size += 1;
        }
        return this;
    }

    public boolean has(int i) {
        return rep.find(i);
    }

    public int size() {return size;}
}
The Dispatch Problem

• Consider a client program that uses the IntSet interface

```
IntSet set = ...;
int x = set.size();
```

• Which code to call?
  – IntSet1.size?
  – IntSet2.size?

• Client code doesn’t know the answer
  – So objects must “know” which code to call
  – Invocation of a method must indirect through the object
Compiling Objects

- Objects contain a pointer to a **dispatch vector** (also called a **virtual table** or **vtable**) with pointers to method code.

- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector.

```
set

IntSet

?.

IntSet1

rep:List

IntSet1.insert

IntSet1.has

IntSet1.size

IntSet2

rep:Tree

size:int

IntSet2.insert

IntSet2.has

IntSet2.size

?.insert

?.has

?.size

Dispatch Vector
```
Method Dispatch (Single Inheritance)

- Idea: every method has its own small integer index
- Index is used to look up the method in the dispatch vector

```java
interface A {
    void foo();
}

interface B extends A {
    void bar(int x);
    void baz();
}

class C implements B {
    void foo() {...}
    void bar(int x) {...}
    void baz() {...}
    void quux() {...}
}

Index
0
1
2
3

Inheritance / Subtyping
C <: B <: A

Zhendong Su    Compiler Design
• Each interface and class gives rise to a dispatch vector layout
• Note that inherited methods have identical dispatch indices in the subclass (Width subtyping)

Dispach Vector

A fields

B fields

C fields

A

B

C

foo

foo

foo

bar

bar

baz

baz

quux