Lecture 16

COMPILER DESIGN
Announcements

• **HW4: OAT v. 1.0**
  – Parsing & basic code generation
  – **Due:** Tuesday, November 12\(^{\text{th}}\) at 23:59
  – Can submit by Thursday, November 14\(^{\text{th}}\) at 23:59 \textit{w/o penalty}

• **HW5: OAT v. 2.0**
  – Records, function pointers, type checking, array-bounds checks, etc.
  – **Due:** Thursday, November 28\(^{\text{th}}\) at 23:59
  – \textit{Will be available after midnight today}
SUBTYPING OTHER TYPES
Extending Subtyping to Other Types

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

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

• What about functions?

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

• One way to see it

Expected function

\[ S_1 \xrightarrow{T_1} T_1 \xrightarrow{T_2} T_2 \xrightarrow{S_2} S_2 \]

Actual function

• 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; ... ; lab_n:T_n\}
  - Each lab_i is a label drawn from a set of identifiers

\[
\begin{align*}
\text{RECORD} & \quad E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad ... \quad E \vdash e_n : T_n \\
\text{PROJECTION} & \quad E \vdash \{\text{lab}_1 = e_1; \text{lab}_2 = e_2; \ldots; \text{lab}_n = e_n\} : \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_n:T_n\} \\
& \quad E \vdash e.\text{lab}_i : T_i
\end{align*}
\]
Immutable Record Subtyping

- **Depth subtyping**
  - Corresponding fields may be subtypes

Depth

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

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

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

Width

\[ m \leq n \]

\[ \{ \text{lab}_1:T_1; \text{lab}_2:T_2; \ldots ; \text{lab}_n:T_n \} <: \{ \text{lab}_1:T_1; \text{lab}_2:T_2; \ldots ; \text{lab}_m:T_m \} \]
• 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
  \{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 we want both …

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

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

\[ q : \{y:\text{int}; z:\text{int}\} = p \]
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 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 NonZero <: Int

• Should \( (\text{NonZero ref}) <: (\text{Int ref}) \) ?

• Consider this program

```c
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:
  $$T_1 \ \text{ref} <: T_2 \ \text{ref} \quad \text{implies} \quad T_1 = T_2$$

- 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:
  \[ T \text{ ref} \cong \{ \text{get: unit }\rightarrow T; \; \text{set: } T \rightarrow \text{unit} \} \]
  - get returns the value hidden in the state
  - set updates the value hidden in the state

- When is \( T \text{ ref} <: S \text{ ref} \)?
- Records are like tuples: subtyping extends pointwise over each component
  \{\text{get: unit }\rightarrow T; \text{set: } T \rightarrow \text{unit}\} <: \{\text{get: unit }\rightarrow S; \text{set: } S \rightarrow \text{unit}\}
  - get components are subtypes: \( \text{unit }\rightarrow T <: \text{unit }\rightarrow S \)
  - set components are subtypes: \( T \rightarrow \text{unit} <: S \rightarrow \text{unit} \)

- 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: *)
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 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
OAT'S TYPE SYSTEM

See oat.pdf 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 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
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?
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:

```java
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.
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.
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
```
Each interface and class gives rise to a dispatch vector layout. Note that inherited methods have identical dispatch indices in the subclass. (Width subtyping)