Lecture 17

CIS 341: COMPILERS
Announcements

• Reminder: HW4 Compiling OAT v.1
• DUE: Thursday, March 26\textsuperscript{th}
• \textit{IF YOU HAVEN’T STARTED, YOU ARE DOOMED}
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 *)
```
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 subtype relation: $\text{Pos} <: \text{Int}$
• Such inclusions give rise to a subtyping hierarchy:

```
      Any
     /   \
   Int   Bool
  / \     /
Neg Zero Pos True False
```

• Given any two types $T_1$ and $T_2$, we can calculate their least upper bound (LUB) according to the hierarchy.
  – Example: $\text{LUB}(\text{True}, \text{False}) = \text{Bool}$, $\text{LUB}(\text{Int}, \text{Bool}) = \text{Any}$
  – 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(T₁, T₂) is the most precise type (according to the hierarchy) that is able to describe any value that has either type T₁ or type T₂.

• In math notation, LUB(T₁, T₂) is sometimes written T₁ ∨ T₂

• LUB is also called the join operation.
• A *subtyping hierarchy*:

```
Any

Int

Neg     Zero     Pos

Neg     Zero     Pos

Any

Bool

True     False
```

• 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 $⟦T⟧$ for the subset of (closed) values of type $T$
  - i.e. $⟦T⟧ = \{v \mid \vdash v : T\}$
  - e.g. $⟦\text{Zero}⟧ = \{0\}$, $⟦\text{Pos}⟧ = \{1, 2, 3, \ldots\}$

- If $T_1 <: T_2$ implies $⟦T_1⟧ \subseteq ⟦T_2⟧$, 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:
  \[ [\text{LUB}(T_1, T_2)] \supseteq [T_1] \cup [T_2] \]
  - Note that the LUB is an over approximation of the “semantic union”
  - Example: \[ [\text{LUB(Zero, Pos)] = [\text{Int]} = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq \{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\text{Zero}] \cup [\text{Pos}] \]

- 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 +

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

• When we add subtyping judgments of the form \( T <: S \) we can uniformly integrate it into the type system:

<table>
<thead>
<tr>
<th>SUBSUMPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \vdash e : T \quad T &lt;: S )</td>
</tr>
</tbody>
</table>

\[ E \vdash e : S \]

• 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.
  – Basic Idea: Use subsumption only where absolutely necessary
    • When checking an argument expression against a function type
    • When checking an expression against a declaration type
Checked 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 can find out.

• Add a “checked downcast”
  \[ E \vdash e_1 : \text{Int} \quad E, x : \text{Pos} \vdash e_2 : T_2 \quad E \vdash e_3 : T_3 \]

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

• 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: \( \text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int} \)
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 $T_1 \to T_2 <: S_1 \to S_2$?
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: \( \{ \text{lab}_1: T_1; \text{lab}_2: T_2; \ldots; \text{lab}_n: T_n \} \)
  - Each \( \text{lab}_i \) is a label drawn from a set of identifiers.

**RECORD**

\[
E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad \ldots \quad E \vdash e_n : T_n
\]

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

**PROJECTION**

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

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

• Depth subtyping:
  – Corresponding fields may be subtypes

\[
T_1 <: U_1 \\ T_2 <: U_2 \\ ... \\ T_n <: U_n
\]

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

• Width subtyping:
  – Subtype record may have \textit{more} fields:

\[
m \leq n
\]

\[
\{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_n:T_n\} <: \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_m:T_m\}
\]
Immutable Record Subtyping (cont’d)

- Width subtyping assumes an implementation in which order of fields in a record matters:
  \[
  \{x:\text{int}; y:\text{int}\} \neq \{y:\text{int}; x:\text{int}\}
  \]
- But:  \[
  \{x:\text{int}; y:\text{int}; z:\text{int}\} <: \{x:\text{int}; y:\text{int}\}
  \]
  - Implementation: a record is a struct, subtypes just add fields at the end of the struct.

- Alternative: allow permutation of record fields:
  \[
  \{x:\text{int}; y:\text{int}\} = \{y:\text{int}; x:\text{int}\}
  \]
  - 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:\text{int}; z:\text{int}; y:\text{int}\} = \{x:\text{int}; y:\text{int}; z:\text{int}\} <:/ \{y:\text{int}; z:\text{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:

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

\[
q: \{y:\text{int}; \ z:\text{int}\} = p
\]
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:  \( \text{Int} \rightarrow \text{NonZero} \rightarrow \text{Int} \)
  
  – Recall that \( \text{NonZero} <: \text{Int} \)

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

• 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:
  \[ T_1 \text{ ref } <: T_2 \text{ ref } \text{ implies } T_1 = T_2 \]

• 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:

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

– 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: \text{T }\rightarrow \text{unit} <: \text{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: *)

```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 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”
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
MODULARITY & ABSTRACTION
Modular Programming

• Programs are typically composed of many modules.
  – Separate compilation – scalable to millions of lines
  – Code reuse – libraries, sharing
  – Namespace management
  – Encapsulation – hiding complexity
  – Abstraction & abstract data-types
  – Security

• What is a module?
  – A collection of named, related values and types
  – Definitions (partially) hidden from the outside

• Examples: Java classes & packages, C++ classes, Modula-3 modules, SML/Ocaml structures & functors, CLU clusters, C source files, …
Separate Compilation

• Program is made of several *compilation units*
  – Independent inputs to the compiler

• Avoids needing to recompile the whole program for every change
• Code is more reusable (libraries)
• Examples:
  – C: .c files / Java: .java files / OCaml: .ml files

• For building a whole program out of compilation units:
• Need to know how to reference values in other units
  – Solution: namespaces + linking
• Need to know datatype sizes (for code generation) or types (for type safety)
  – Solution: interfaces (C: .h files / Java: .class files / OCaml: .mli files)
Namespaces

• In C and FORTRAN: all global identifiers are visible everywhere
• Problem:
  – Can’t have two global variables or functions with the same name
  – (Also, linker doesn’t type check)
• Solutions:
  – C++, Java qualified identifiers: C.x or P₁.P₂.P₃.C.x (where C is a class name)
  – Modula-3, OCaml: qualified identifiers + renaming
  – Java, Modula-3, OCaml: link-time type checking

• Wrinkle: object code formats typically have a flat name space
  – Need to *mangle* qualified identifiers
  – e.g. C++: `int C::f(int x)` becomes `f__1Ci`
**Linking**

Input:

```c
extern int x;

void main() {
    printf("%d", x);
}
```

compiles to asm:  

```
f1.s
f2.s
```

assembles to obj:

```
f1.o
f2.o
```

linker

- **Problem:** compiler can’t generate code to access variable `x` because its address is unknown.

- **Solution:** Generate placeholder reference to `x` in `f1.s`, generate definition of `x` in `f2.s`, linker patches the files together, replacing placeholders in `f1.s` with actual value from `f2.s`
  
  - Exact mechanism depends on linker/OS object file format
Encapsulation

- It’s often useful to hide some information contained in a module.
- Example:

```java
String[] names; // should be hidden
String[] passwords; // should be hidden

bool check_password(String n, String p) {
    int j = 0;
    while (j < names.length) {
        if (names[j] == n & passwords[j] == p)
            return true;
        j = j + 1;
    }
    return false;
}
```

- Encapsulation can protect a module’s data from tampering
  - Good software engineering practices rely on encapsulation.
Encapsulation Mechanisms

• Fundamentally, need a way to indicate which identifiers should be exported from a module.
• C++/Java: “public” vs. “private” qualifiers:

```cpp
class PWChecker {
    private String[] names;       // should be hidden
    private String[] passwords;   // should be hidden
    public bool check_password(String n, String p) {...} }
```

• ML / Modula-3: separate interfaces (omit hidden identifiers):

```ml
module type PWChecker = sig
    val check_password : String * String -> bool
    (* Note: no declaration for names or password *)
end
```

• C: “static” qualifier

```c
static int check_password(char *n, char *p)
```
• Records (or structs) bundle values together, mapping names to values.
• Modules also bundle values together…
  – Except that modules are computed at load time
  – They are (usually) 2nd class (e.g. modules cannot be passed arguments to functions). (OCaml v. 3.12 has support for first-class modules.)
• But… module interfaces look like record types:

```ocaml
module PWC = struct
  let names : string array = ...
  let passwords : string array = ...
  let check_password (n:string, p:string):bool = ...
  let is_name (n:string):bool = ...
end :

sig
  val check_password : string * string -> bool
  val is_name : string -> bool
end
```
Abstract Data Types

• Key idea: abstract type
  – An identifier representing an unknown type

• Abstract Data Type is
  – A type identifier (possibly parameterized) +
  – Declared operations on that type +
  – Concrete type definition (a representation) +
  – Concrete implementation of the operations

• IntSet interface in OCaml:
  module type IntSet = sig
    type intset      (* Note: no type definition *)
    val empty : intset
    val insert : int -> intset -> intset
    val has : int -> intset -> bool
  end
module IntSet1 : IntSet = struct
  type intset = int list
  let empty = []
  let insert i s = i::s
  let rec has = ...
end

module IntSet2 : IntSet = struct
  type intset = Leaf | Node of intset * int * intset
  let empty = Leaf
  let rec insert i s = ...
  let rec has = ...
end

This signature ascription seals the modules with an abstract type, hiding the representation of intset.
Implementing Abstract Types

• Representation of the abstract type is hidden from code other than the implementation itself
  – CLU, Ada, Modula-3, ML

• Because external code doesn’t know representation, it can’t violate the abstraction boundary
  – e.g. break representation invariants

• Positive: The same interface can be reimplemented multiple ways.
• Positive: Module signatures can bundle together multiple related abstract types.
• Negative: Compiler doesn’t know representation either
  – When compiling external code it must use level of indirection
  – No stack allocation of abstract types
Type Checking A Module

- Module definitions must agree with the interface in the signature
- Inside the module the concrete types are known
  - Extend the context with the definition (or substitute $S_i$ for $I_i$)
- This rule also provides width subtyping

\[
\begin{align*}
E' = E, I_1 = S_1, I_2 = S_2, \ldots I_n = S_n \\
E' \vdash e_1 : T_1 \quad E' \vdash e_2 : T_2 \quad \ldots \quad E' \vdash e_m : T_m \quad E' \vdash e_{m+1} : T_{m+1} \ldots E' \vdash e_k : T_k
\end{align*}
\]

\[
\begin{align*}
E \vdash & \quad \textbf{struct} \\
& \quad \text{type } I_1 = S_1 \\
& \quad \ldots \\
& \quad \text{type } I_n = S_n \\
& \quad \text{let } v_1 : T_1 = e_1 \\
& \quad \ldots \\
& \quad \text{let } v_k : T_k = e_k \\
& \quad \textbf{end} \\
& \quad \textbf{sig} \\
& \quad \text{type } I_1 \\
& \quad \ldots \\
& \quad \text{type } I_n \\
& \quad \text{val } v_1 : T_1 \\
& \quad \ldots \\
& \quad \text{val } v_m : T_m \\
& \quad \textbf{end}
\end{align*}
\]
Classes

• Fields or instance variables:
  – Values may differ from object to object (not shared)
  – Usually mutable
  – Presence inherited from the superclass

• Methods:
  – (Function) values shared among all instances of a class
  – Code inherited from the superclass
  – Immutable (usually)
  – Usually take an implicit argument that refers to the object itself (this or self)

• All components have visibility modifiers
  – public/private/protected (subclass visible)
Objects as Abstract Data Types (ADTs)

- Objects: another way of extending records to ADTs
- Source code for the class defines the concrete types and implementation
- Interface defined either implicitly (via public members) or explicitly via interface ascription

```java
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(); }
}
```

```java
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```
Classes in C++/Java

• Classes have private/public visibility qualifiers that hide part of the object.
• A class is a *partially* abstract type
  – (Note: do not confuse with Java’s ‘abstract’ keyword)

• Interface file declares the representation
  – Method code is (mostly) hidden from the outside

• Positive: This mechanism allows external code to know how much space each object takes while still providing encapsulation
  – Objects can be stack allocated (good for cache coherence/performance)
• Negative: Change to representation can require complete recompilation, even of external code
  – C++ is notoriously slow to compile
• Negative: Each class defines only a *single* type.
IntSet example in C

• intset.h:

```c
struct intset;
extern struct intset *empty;
struct intset *insert(int i, struct intset *s);
int has(int i, struct intset *s);
```

• intset.c:

```c
#include "intset.h"

struct intset {struct intset *left;
    int val; struct intset *right; }

struct intset *empty = NULL;

struct intset *insert(int i, struct intset *s) {...}
int has(int i, struct intset *s) {...}
```
No Abstraction in C

• C provides hiding/encapsulation but no abstraction.

• (Unchecked) Casts allow any client code to violate the representation invariants of the module.
FULL OAT'S TYPE SYSTEM
Class Hierarchy

• The set of class interfaces form a class hierarchy

A class interface for C consists of:

<: D  
C’s superclass
(t1 .. tn) -> C  
C’s constructor signature
x1:t1 ... xn:tn  
C’s field types
m1 : ftyp1 ... mk : ftypk  
C’s method types

• The hierarchy H must be consistent:

if C <: D then D is either “Object” or defined earlier in H
Example from HW5

- Full Oat distinguishes “possibly null” references from “definitely not null” references:

- Types:

\[
\begin{align*}
t & ::= \\
& | \text{bool} \\
& | \text{int} \\
& | \text{null} // \text{the type of the ‘null’ constant} \\
& | \text{ref?} // \text{possibly null reference} \\
& | \text{ref} // \text{definitely not-null reference} \\
\end{align*}
\]

\[
\begin{align*}
\text{ref} & ::= \\
& | C // \text{class type} \\
& | \text{string} \\
& | t[] // \text{array types} \\
\end{align*}
\]