HW 3 due next Tuesday

Rest of the semester:
- concurrency
- types
- OO
- concurrency
- other stuff
What’s a type?

- A *predicate* on what values might occur at run time

- “int x;” in Java means \( x \in [-2^{31}, 2^{31}) \)
Can a type be an arbitrary predicate?
- e.g., $x$ : an integer equal to gcd($a, b$)

In principle, yes, but:
- might be undecidable or at least intractable (theorem proving)
- but automatic theorem provers are getting better and better

Types should be *efficiently decidable*
Languages have *type systems*

Type system describes:
- what types can be expressed
- what types expressions have
Type safety

Type errors:
- Improper, type-inconsistent operations during program execution

**Type safety** = absence of run-time type errors

- Operations unsupported by a value will *never* occur *at run time*
  - cannot multiply a string and an int
  - cannot call a function that is not defined
  - cannot access a field that does not exist
  - cannot access access outside the bounds of an array
Ensuring type safety

Bind (assign) types, then check types

- Type binding
  - define types for constructs in program (e.g., variables, functions)
  - can be either explicit (int x) or implicit (x = 1)
  - type consistency (safety) = correctness with respect to type bindings

- Type checking
  - static (compile-time) checks to enforce type safety of the program
Strong vs. weak typing

Strong and weak typing refer to how much type consistency is enforced

Strongly typed languages
- guarantees that accepted programs are type-safe

Weakly typed languages
- allow programs that contain type errors
Static vs. dynamic typing

Languages can be either statically or dynamically typed

Statically typed
- e.g., C, Java, Scala, ML
- Types defined and checked at compile time
- Do not change during execution of the program
- Compiler can compute a static type for every expression

Dynamically typed
- e.g., Scheme, JavaScript, Smalltalk, Ruby
- Types defined and checked at run time

Static vs. dynamic is orthogonal to safe vs. unsafe
## Type systems

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td>Java, C#, Scala</td>
<td>Smalltalk, JavaScript, Scheme</td>
</tr>
<tr>
<td>weak</td>
<td>C, C++</td>
<td>Perl, Forth, assembly code</td>
</tr>
</tbody>
</table>

Tuesday, March 23, 2010
Soundness

Sound type systems

- can statically ensure program is type-safe

Soundness implies strong typing

Static type safety requires a conservative approximation of the values that might occur during all possible executions

- May reject type-safe programs
- Need to be expressive, reject as few type-safe programs as possible
Summary

Static vs. dynamic typing
- when to define/check types?

Strong vs. weak typing
- how many type errors?

Sound type systems
- statically ensure no type errors will occur at run time
  (and possibly reject some programs that have no type errors)
Type systems

Static vs. dynamic is a continuum:

- Most statically typed languages have some dynamic type checks
- Infeasible to check everything statically
- Often, a tradeoff between static typing and expressiveness

Type safety is not a continuum:

- C, C++ are unsafe
- Java, C# are safe
- but, can say C++ is safer than C (barely)
Dynamic checks

Even statically typed languages have some dynamic checks

- Java:
  - array index out of bounds
  - null pointer dereference
  - load-time inter-module checking

Sometimes these can be eliminated by static analysis

- harder than type-checking (undecidable)
- theorem proving
- can’t always eliminate these checks
Why static typing?

Compiler can reason more effectively
- Allows more efficient code
  - don’t have to check for unsupported operations
- Allows error detection by compiler

But:
- requires at least some type declarations (annotations)
- although many types can be *inferred* (Scala, ML, Haskell)

Documents interfaces for programmers
Type expressions

Types described in the program by *type expressions*
- int, String, Array[Int], InputStream[], Vector<Integer>, ...

Ground types (example: Java)
- Primitive types
  - int, boolean
- Class types
  - String, Object, FileInputStream
- Interface types
  - Serializable, Cloneable

Also, *type constructors* (functions from types to types)
- arrays: int[], String[]
Type aliases

Some languages allow type aliases
- aka type definitions, equates

Example:
- C: typedef int int_array[];
- M3: TYPE int_array = ARRAY OF INT;

- int_array is a type expression denoting the same type as int[]

Different type expressions may denote the same type
Arrays

Different languages have various kinds of array types

Without bounds:
- C, Java: T[]
- M3: ARRAY OF T
- Scala: Array[T]

With size:
- C: T[N]
- M3: ARRAY[N] of T

With bounds:
- M3: ARRAY[1..10] of T
Multidimensional arrays

C, Java only 1-d arrays
- int[][][] is an array of arrays

Fortran
- A(i,j,k)

X10
- arrays are defined over sets of n-d points (regions)
- a: Array[Int](1..10 * 1..100)
- p = [3,14]
- a(p)
Records/structs

More complex type constructor
- \{x_1: T_1, ..., x_n: T_n\}

C: struct { int a; float b; }

Pascal: record a: integer; b: real; end

Objects: generalize notion of records

Support access operations via field names
- x.a, x.b
Tuples

Scala:
- val p: (String, Int) = ("abc", 123)
- p._1  // type String, “abc”
- p._2  // type Int, 123

Can be viewed as a record
Pointer types

Values that are addresses of variables of other types

C:
- int *x;

Pascal:
- x: ^integer;

Java:
- object references
Function types

\[(T_1, T_2, ..., T_n) \Rightarrow T\]
- Type of function values that can be invoked with arguments of types \(T_i\) and that returns type \(T\)

First-class functions:
- \((x: \text{Int}) \Rightarrow x+1\)  // type Int => Int

Methods:
- class C(x:Int) { def m(y: Int) = x+y }
  (new C(3)).m _  // type Int => Int
Type aliases

aka type definitions, equates

C: typedef int int_array[];
M3: TYPE int_arrat = ARRAY OF INTEGER;
Scala: type int_array = Array[Int];
Java doesn’t have type aliases

Aliases are not type constructors!
  - int_array is the same type as int[], not a different type

Different type expressions may denote the same type
Class types

Next time!
Static semantics

How are type systems specified?

Describe the types used in a program

Formal description: static semantics for the programming language
- is to type-checking as grammar is to parsing

Static semantics define types for all legal languages ASTs
We write in ordinary language syntax to mean corresponding AST
Static semantics

Type system describes how to compute types for each expression and how to check compatibility of types

<table>
<thead>
<tr>
<th>if (e) S1 else S2</th>
<th>is e a boolean?</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = z</td>
<td>is z’s type a subtype of y’s?</td>
</tr>
<tr>
<td>x + 4</td>
<td>if x is an int, x+4 is also an int</td>
</tr>
</tbody>
</table>
Type checking

Consider:
- $x < 0 \land b$

What do we need to decide that this is a well-typed expression of type boolean?

- $x$ must be an int
- $0$ must be an int
- $b$ must be a boolean
Type judgment

Formalize as a set of type judgments

\[ A \vdash e : T \]

“in context A, e has type T”

Context A is a set of type assignments \( x : T \) (“\( x \) has type \( T \)”)

Example:
- \( x : \text{int}, b : \text{boolean} \vdash x < 0 \&\& b : \text{boolean} \)
- “if \( x \) is an int and \( b \) is a boolean, then \( x < 0 \&\& b \) is a boolean”
Deriving a judgment

To show:
- $x : \text{int}, b : \text{boolean} \vdash x < 0 \& \& b : \text{boolean}$

Need to show:
- $x : \text{int} \vdash x : \text{int}$
- $x : \text{int} \vdash x < 0 : \text{boolean}$
- $b : \text{boolean} \vdash b : \text{boolean}$
To show:
- $A \vdash e_1 \&\& e_2 : \text{boolean}$

Need to show:
- $A \vdash e_1 : \text{boolean}$
- $A \vdash e_2 : \text{boolean}$
As an inference rule

Premises $\vdash e_1 : \text{boolean}$ $\vdash e_2 : \text{boolean}$

Conclusion $\vdash e_1 \&\& e_2 : \text{boolean}$

Holds for any choice of the syntactic meta-variables $e_1, e_2$
Inference rule:

\[
\begin{array}{ccc}
P_1 & \ldots & P_n \\
\hline
\end{array}
\]

\[Q\]

“if \(P_1\) and \(P_2\) and \ldots and \(P_n\), then \(Q\)”

Each \(P\), \(Q\) is a typing judgment or another proposition

Can omit premises \(P\) if \(Q\) is always true — rule called an \textit{axiom}
Why inference rules?

Compact, precise language for specifying static semantics

Inference rules are to type checking as grammars are to parsing

Type checking is attempt to prove type judgments $A \vdash e : T$ by walking backward through the rules
Constants

A ⊬ n : int

A ⊬ true : boolean
A ⊬ false : boolean
Variables

\[ A \vdash x : A(x) \]
Assignment rule

\[ A \vdash x : T \quad A \vdash e : T \]

\[ \frac{}{A \vdash x = e : T} \]
Deriving a judgment

Let
- \( A = x : \text{int}, b : \text{boolean} \)

Then, we can derive:

\[
\begin{align*}
A \vdash x : \text{int} & & A \vdash 0 : \text{int} \\
\hline
A \vdash x < 0 : \text{boolean} & & A \vdash b : \text{boolean} \\
\hline
A \vdash x < 0 \&\& b : \text{boolean}
\end{align*}
\]

This is a *proof* of the typing judgement. The judgment is *derived* from the axioms.
Binary operations

\[
\begin{align*}
A \vdash e_1 : \text{int} & \quad A \vdash e_2 : \text{int} \\
\hline
A \vdash e_1 + e_2 : \text{int}
\end{align*}
\]
Checking statements

\[ A \vdash e : \text{boolean} \quad A \vdash s_1 : \text{void} \quad A \vdash s_2 : \text{void} \]

\[ \text{if (e) } s_1 \text{ else } s_2 : \text{void} \]

\[ A \vdash e : \text{boolean} \quad A \vdash s : \text{void} \]

\[ \text{while (e) } s : \text{void} \]
Checking arrays

\[
A \vdash e : \text{int}
\]

\[
\underline{A \vdash \text{new } T[e] : T[]}
\]

\[
\begin{align*}
A &\vdash e_1 : T[] \\
A &\vdash e_2 : \text{int}
\end{align*}
\]

\[
\underline{A \vdash e_1[e_2] : T}
\]
Checking field accesses

\[ A \vdash e : \{ x_1 : T_1, \ldots, x_n : T_n \} \]

\[ A \vdash e.x_i : T_i \]
Checking calls

\[
A \vdash e_0 : (T_1, \ldots, T_n) \Rightarrow T \quad A \vdash e_i : T_i \\
\hline
A \vdash e_0(e_1, \ldots, e_n) : T
\]