Lecture 10 — Type-checking MiniJava

- Once a program has been parsed and transformed to an AST, some other checks must be made before translation to executable form. The main check is *type-checking*. We will discuss type-checking, and introduce a notation for specifying type rules that we will use for several other purposes this semester. Note that type-checking is done only in statically-typed languages.
  - Static typing vs. dynamic typing
  - Proof systems to specify type rules
  - MiniJava type rules (with and without inheritance)

"Context-sensitive syntax"

- **A program can be malformed even though it conforms to the context-free grammar of the language, e.g.**
  - `switch (...) { case 1: ... case 1: ...`
  - `int x; float x; ...`
- **Before attempting to generate executable code, need to make sure program is well-formed.**
- **These types of checks can be performed by the usual recursive traversal of the AST.**

Statically- vs. dynamically-typed languages

- **One type of well-formedness check is *type-checking*, e.g.:**
  - `(new C())+10`
  - `(new C()).x`, **where class** `C` **does not have a field** `x`
- ***Statically-typed languages* check types *before* attempting to generate code or execute the program.**
- ***Dynamically-typed languages* do not check types before execution, so type errors only show up *during* execution.**
- **How can we tell whether a language is statically or dynamically typed?**

Statically- vs. dynamically-typed languages (cont.)

- C
- C++
- Java
- Python
- Javascript
- OCaml
- Scala
- LISP

Type-checking

- In addition to checking type correctness, a type-checker will record the type of every name (variable, method, etc.), so that the correct code can be generated. E.g.
  - `x+y` generates an integer add instruction if `x` and `y` are integers; a floating-point instruction if they are floats; etc.
- Type checkers can record types in two ways:
  - Construct a *symbol table* indexed by *class*, by *class* $\times$ *name* (for fields and methods), and by *class* $\times$ *method* $\times$ *name* (local variables).
  - Change the abstract syntax so that every use of a variable includes its type; iterate over the AST filling in those types at every use.

Proof systems

- **Proof systems are a convenient way to present the type rules of a language. In effect, they are just a nice notation for expressing recursive functions on ASTs.**

- **A *proof system* is:**

  - **A set of *judgments* — assertions (true-false statements), usually presented in some concise mathematical notation.**
  - **A set of *axioms* — judgments that are known to be true.**
  - **A set of *rules of inference* of the form:** $\dfrac{J_1 \quad \vdots \quad J_k}{J}$

    asserting that J is true if $J_1, \ldots, J_k$ are true.

Proof systems (cont.)

- **Axioms are usually given as *patterns* that represent an infinite set of axioms; similarly for rule of inference:**

  n **: int** **(read: "integer constant n has type int")**

  is shorthand for the axioms "0 : int", "1 : int", etc.

  e_1**+**e_2 **: int** **("expression e_1+e_2 has type int")**
  e_1 **: int** **("if expression e_1 has type int")**
  e_2 **: int** **("and expression e_2 has type int")**

  is shorthand for

  x+y : int **3+f(a) : int** **etc.**
  x : int **3 : int**
  y : int **f(a) : int**

Proof systems (cont.)

- **Given a proof system, a *proof* (of judgment J) is a tree (rooted at J) where each node is labelled with a judgment. Leaf nodes are labelled with axioms. An internal node labelled with judgment J_0 just have children labelled $J_1, \ldots, J_k$, where the proof system has a rule of inference inferring J_0 from $J_1, \ldots, J_k$.**

- **A proof tree proves its root judgment because the leaf nodes are assumed true, and every internal node is proven by its children (according to a rule of inference).**

Proof systems for type-checking

- **A proof system for type-checking will give an axiom or rule of inference showing how to type-check expressions or statements formed from each abstract syntax operator.**

- **It may have more than one rule for a single abstract syntax constructor, e.g.**

$$\frac{}{e_1 + e_2 \text{ : int}} \qquad \frac{e_1 \text{ : int} \quad e_2 \text{ : int}}{e_1 + e_2 \text{ : int}}$$

$$\frac{}{e_1 + e_2 \text{ : float}} \qquad \frac{e_1 \text{ : float} \quad e_2 \text{ : float}}{e_1 + e_2 \text{ : float}}$$

Proof systems for type-checking (cont.)

- **Any applicable rule can be used. The proof system needs to be designed so that if two rules are applicable to one expression, they must not produce conflicting results.**

- **E.g.** $e_1 + e_2$ **: string** $\qquad$ $e_1 + e_2$ **: string**
  e_1 **: string** $\qquad$ e_2 **: string**

Type-checking rules for MJ

- **We give several kinds of judgments. Γ is a *type environment*, giving the types of variables.**

$\vdash \pi$	**π is a type-correct program**
$\pi \vdash \kappa$	**κ is a type-correct class in program π**
$\pi, \Gamma \vdash \mu$	**μ a type-correct method in a class whose fields are given in Γ.**
$\pi, \Gamma \vdash S$	**S is a type-correct statement, when Γ gives all variable declarations surrounding S (fields, parameters, locals)**
$\pi, \Gamma \vdash e : \tau$	**e is a type-correct expression of type τ, where Γ gives the types of any variables occurring in e.**

Type-checking rules for expressions

$\pi,\ \Gamma \vdash \texttt{x}$

$\pi,\ \Gamma \vdash \texttt{not}\ e$

$\pi,\ \Gamma \vdash e_1\ \texttt{*}\ e_2 : \mathsf{int}$

$\pi,\ \Gamma \vdash e_1 + e_2 : \mathsf{int}$

Type-checking rules for expressions

$\pi,\ \Gamma \vdash e_1 + e_2 :$ string

$\pi,\ \Gamma \vdash e_1 + e_2 :$ string

$\pi,\ \Gamma \vdash e_0.f(e_1, \ldots, e_n)$ *(ignore subclassing for now)*

Type-checking rules for statements

$\pi,\ \Gamma \vdash \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2$

$\pi,\ \Gamma \vdash x\ \texttt{=}\ e$

Type-checking rules for programs and classes

$\vdash \pi$ where $\pi = cl_1 \ \ldots \ cl_n$

$\pi \vdash \texttt{class}\ c\ \{\ fld_1\ \ldots\ fld_k\ \mu_1\ \ldots\ \mu_m\ \}$

Type-checking rules for methods

$\pi, \Gamma \vdash \tau \;\; f \;\; \texttt{(args)} \;\; \{ \;\; S_1 \;\; \ldots \;\; S_n \;\; \texttt{return} \;\; e \;\; \}$

Subclasses

- **Write $C < B$ if C is B or a descendant of B.**
- **Rules that take subclasses into account: assignment, method call, method definition.**
- **Basic rule of inheritance: Wherever an object of a class C can be safely used as a receiver of a method, an object of a descendant class D can also be used safely:**
  - **If the method refers to a field of C, objects of D inherit that field.**
  - **If the method calls a method f using `this` as the receiver, `this` inherits (or redefines) f.**

Type-checking rules, with subclasses

$\pi, \Gamma \vdash x{=}e$

$\pi, \Gamma \vdash \tau\ \ f\ \ \texttt{(args)}\ \{\ S_1\ \ldots\ S_n\ \texttt{return}\ e\ \}$

Type-checking rules, with subclasses (cont.)

$$\pi, \Gamma \vdash e_0.f(e_1, \ldots, e_n) : \tau$$

$$\pi \vdash \texttt{class}\ c\ \texttt{extends}\ s\ \{\ fld_1\ \ldots\ fld_k\ \mu_1\ \ldots\ \mu_m\ \}$$

Type-correctness theorem

Theorem Suppose $\vdash \pi$. Then when we execute π, there will be no type errors raised.

- To prove, we need to define precisely what "execute π" means. We will start to do that next week.
- Note this implies that there is no need to check types at run time, because any argument to an operator or method is guaranteed to have the correct type. Dynamically-typed languages cannot make that guarantee, so types must be checked at run-time. This is the main reason dynamically-typed languages are inefficient.

Wrap-up

- **Today we discussed:**
  - Using proof systems to specify type rules
  - Type rules for MiniJava
- **We discussed it because:**
  - Type-checking is the next step in a compiler for a statically-typed language like MJ.
- **After the mid-term, we will begin to talk about executing MJ programs. (There will not be an MP on type-checking MJ, because it is complicated without being interesting.)**
- **What to do now:**
  - Mid-term 1 Monday night, 7PM. *See web page for correct room.*
  - Review session for mid-term Sunday night; see web page for time and location.