

Lecture 14 — Type-checking MiniJava

- Dynamically-typed languages
- Type-checking
- SOS rules for MiniJava
- Type-correctness theorem
- Type-checking and efficiency
- MP 7

What you've achieved

- Implemented powerful language with inheritance.
- Can write functional programs, using List class, in style of OCaml and Python.
- Remaining stuff is mostly easy
 - While and switch
 - Floating-point numbers
 - Arrays
 - Better notation for lists
 - Random access to heap

Dynamically-typed languages

- Popular dynamically-typed languages:
- Popular statically-typed languages:
- Which is better?

Heterogeneous lists in dynamically-typed languages

- Example of convenience of dynamically-typed languages: heterogeneous lists can be used in cases where Java/C++ would require class definitions:
 - Structs: just put various values in the list
 - Trees: represent using nested lists
- If random access is important, dynamically-typed languages can also have heterogeneous arrays.
- (Dynamically-typed MiniJava has same power, but just lacks the convenient syntax.)

Type-checking

- *Type-checking*: Determine the type of every expression (including all subexpressions), and see whether all operations are used correctly
- In Java: start with declarations of fields, variables, and methods, then traverse AST and check each expression. (In OCaml, things work differently; we'll discuss that another time.)
- Two reasons:
 - Find bugs
 - Efficiency

Type-checking Java, informally

- $e_1 + e_2$

- $x = e$

- $e.f(e_1, \dots, e_n)$

Type-checking rules for MJ

- Will 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 .

- For now, ignore subclassing (i.e. inheritance)...

Type-checking rules for programs and classes

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

$\pi \vdash \text{class } c \{ fld_1 \dots fld_k \mu_1 \dots \mu_m \}$

Type-checking rules for methods

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

Type-checking rules for statements

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

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

Type-checking rules for expressions

$$\pi, \Gamma \vdash x$$
$$\pi, \Gamma \vdash \text{not } e$$
$$\pi, \Gamma \vdash e_1 * e_2 : \text{int}$$
$$\pi, \Gamma \vdash e_1 + e_2 : \text{int}$$

Type-checking rules for expressions

$\pi, \Gamma \vdash e_1 + e_2 : \text{string}$

$\pi, \Gamma \vdash e_1 + e_2 : \text{string}$

$\pi, \Gamma \vdash e_0.f(e_1, \dots, e_n)$

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

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

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

Type-checking rules, with subclasses (cont.)

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

$$\pi \vdash \text{class } c \text{ extends } s \{ fld_1 \dots fld_k \mu_1 \dots \mu_m \}$$

Type-correctness theorem

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

Proof As a statement about the relationship between type-checking and execution, this is proven by using SOS rules for evaluation and for types.

Lemma Suppose σ and Γ are *consistent*: for all x , $\sigma(x)$ is a value of type τ iff $\Gamma(x) = \tau$. Then for any expression e ,
 $\pi, \Gamma \vdash e : \tau \implies$ if $\sigma, e \Downarrow v$, then v is a value of type τ

Proof By induction on the structure of the proof of $\pi, \Gamma \vdash e : \tau$

- This implies that type tags need not be checked during execution, and can be completely removed from values.

Symbol table construction

- Classically, a table is constructed giving the types of all variables, plus other information.
- We will instead construct a new AST in which nodes are decorated with types and other information
- “Other information”
 - Locations of variables on the stack, and fields in an object
 - Sizes of variables determined from their types. E.g. if local variables are $x_1 : \tau_1, \dots, x_n : \tau_n$, locations on stack are, resp., 0, $\text{size}(\tau_1)$, $\text{size}(\tau_1) + \text{size}(\tau_2)$, etc. Same for method arguments, and for fields.
 - In MiniJava, all variables — that is, all stackvalues — have the same size, so locations are just 0, 1, etc.

The cost of interpreting MiniJava

- *Cost of interpretation:* Interpreter repeatedly looks at, and interprets, the same nodes in the AST over and over. (How much does it cost, for example, to execute `x=0`?)
- *Cost of dynamic typing:*
 - *Choosing what operation to perform dynamically.* E.g. `x+y` could be integer addition or string concatenation; must look at type tags of `x` and `y`. (What would this be in C?)
 - *Boxing and unboxing:* Every value is in a “box” with its type tag. Operations must *unbox* operands — extract tag and value — and *box* result — store type tag and value.
- Can eliminate cost of dynamic typing by type-checking, cost of interpretation by compiling.

MP 7: Compiling statically-typed MJ

- MP 7: Compile statically-typed MJ for an abstract machine
- Type-checker will be provided. It does the following:
 - Type-check and reject unsafe programs
 - Construct new *AST* in which every expression has associated *type and location*.
 - New *AST* operations used to eliminate run-time type-checking and lookup:
 - Where conversion from `int` to `string` is indicated by type-checking, insert `cvtIntToString` operation into *AST*.

MP 7: Compiling statically-typed MJ (cont.)

- When $+$ operation is string concatenation (according to type-checker), change it to `Strcat`. Plus is used only for integer addition.
 - `Var x` becomes either `VarT x` or `FieldT x`. Furthermore, `FieldT x` becomes `FieldRef i`, where i is the offset of x within an object.
 - `NewId C` becomes `NewIdAlloc(C, n)`, where n is the size of C objects.
 - Class constructor has field giving size of objects; Method constructor has field giving size of stack frame.
- Result is that actual names of variables and fields, *and* types, become irrelevant to execution.