CSCE 314
Programming Languages

Name Scope and Type System

Dr. Hyunyoung Lee

Names

- Names refer to different kinds of entities in programs, such as variables, functions, classes, templates, modules, ...
- Names can be reserved or user-defined
- Names can be bound statically or dynamically
- Name bindings have a scope: the program area where they are visible

Variables

- Essentially, variables are bindings of a name to a memory address.
- They also have a type, value, and lifetime
- Bindings can be
  - dynamic (occur at run time), or
  - static (occur prior to run time)
- What are the scopes of names here, when are variables bound to types and values, and what are their lifetimes?

```
const int d = 400;
void f() { double d = 100;
           { double d = 200; std::cout << d;}
           std::cout << d;
         }
double g() { return d+1;}
```

Scope

- Scope is a property of a name binding
- The scope of a name binding are the parts of a program (collection of statements, declarations, or expressions) that can access that binding
- Static/lexical scoping
  - Binding's scope is determined by the lexical structure of the program (and is thus known statically)
  - The norm in most of today's languages
  - Efficient lookup: memory location of each variable known at compile-time
  - Scopes can be nested - inner bindings hide the outer ones

Lexical Scoping

```
namespace std { ... }

namespace N {
  void f(int x) {};
  class B {
    void f (bool b) {
      if (b)
      {
        bool b = false; // confusing but OK
        std::cout << b;
      }
    }
  };
}
```

Dynamic Scoping

- Some versions of LISP have dynamic scoping
- Variable's binding is taken from the most recent declaration encountered in the execution path of the program
- Macro expansion of the C preprocessor gives another example of dynamic scoping
- Makes reasoning difficult. For example,

```
#define ADD_A(x) x + a

void add_one(int *x) {
  const int a = 1;
  x = ADD_A(x);
}
```

```
void add_two(int *x) {
  const int a = 2;
  x = ADD_A(x);
}
```

l- and r-values

Depending on the context, a variable can denote the address (l-value), or the value (r-value)

```
int x;
x = x + 1;
```

Some languages distinguish between the syntax denoting the value and the address, e.g., in ML

```
x := !x + 1
```

From type checking perspective, l- or r-valueness is part of the type of an expression

Lifetime

- Time when a variable has memory allocated for it
- Scope and lifetime of a variable often go hand in hand
- A variable can be hidden, but still alive

```
void f (bool b) {
  if (b) {
    bool b = false; // hides the parameter b
    std::cout << b;
  }
}
```

- A variable can be in scope, but not alive

```
A* a = new A();
A& aref = *a;
delete a;
std::cout << aref; // aref is not alive, but in scope
```

Types and Type Systems

- Types are collections of values (with operations that can apply to them)
- At the machine level, values are just sequences of bits
- Is this 0100 0000 0101 1000 0000 0000 0000 0000
  - floating point number 3.375?
  - integer 1079508992?
  - two short integers 16472 and 0?
  - four ASCII characters @ X NUL NUL?
- Programming at machine-level (assembly) requires that programmer keeps track of what are the types of each piece of data
- Type errors (attempting an operation on a data type for which the operation is not defined) hard to avoid
- Goal of type systems is to enable detection of type errors – reject meaningless programs

Languages with some type system, but unsound

- C, C++, Eiffel
- Reject most meaningless programs:

```
int i = 1; char* p = i;
```

- but allow some:

```
union {
  char* p;
  int i;
} my_union;
void foo() {
  my_union.i = 1;
  char* p = my_union.p;
   . . .
}
```

- and deem the behavior undefined – just let the machine run and do whatever

Sound Type System: Java, Haskell

- Reject some meaningless programs at compile-time:

```
Int i = "Erroneous";
```

- Add checks at run-time so that no program behavior is undefined

```
interface Stack
{ void push(Object elem);
  Object pop();
}
class MyStack { . . . }

Stack s = new MyStack();
s.push(1);
s.push("whoAreYou…");
Int i = (Int) s.pop(); // throws an exception
```

Dynamic (but Sound) Type System

- Scheme, Javascript
- Reject no syntactically correct programs at compile-time, types are enforced at run-time:

```
(car (cons 1 2)) ; ok
(car 5)          ; error at run-time
```

- Straightforward to define the set of safe programs and to detect unsafe ones

Type Systems

Common errors -- examples of operations that are outlawed by type systems:

- Add an integer to a function
- Assign to a constant
- Call a non-existing function
- Access a private field

Type systems can help:

- in early error detection
- in code maintenance
- in enforcing abstractions
- in documentation
- in efficiency

Type Systems Terminology

Static vs. dynamic typing

- Whether type checking is done at compile time or at run time

Strong vs. weak typing

- Sometimes means no type errors at run time vs. possibly type errors at run time (type safety)
- Sometimes means no coersions vs. coersions (implicit type conversion)
- Sometimes even means static vs. dynamic

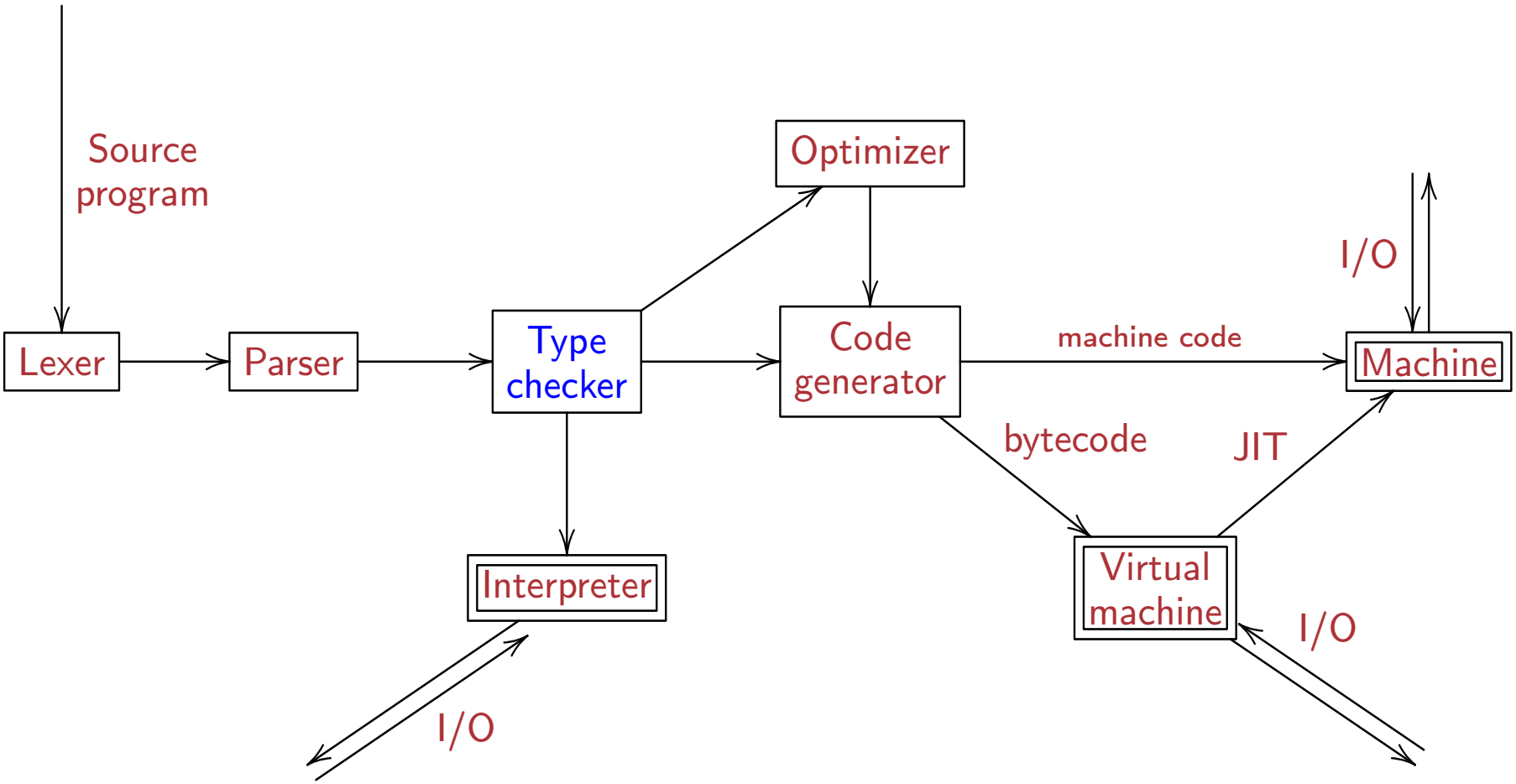
Type Systems Terminology (Cont.)

Type inference

- Whether programmers are required to manually state the types of expressions used in their program or the types can be determined based on how the expr.s are used

- E.g., C requires that every variable be declared with a type; Haskell infers types based on a global analysis

Type Checking in Language Implementation



Type Checking

- Reminder: CF grammars can capture a superset of meaningful programs
- Type checking makes this set smaller (usually to a subset of meaningful programs)
- What kind of safety properties CF grammars cannot express?
  - Variables are always declared prior to their use
  - Variable declarations unique
  - As CF grammars cannot tie a variable to its definition, must parse expressions "untyped," and type-check later
- Type checker ascribes a type to each expression in a program, and checks that each expression and declaration is well-formed

Typing Relation

- By "expression t is of type T", it means that we can see (without having to evaluate t) that when t is evaluated, the result is some value t' of type T
- All of the following mean the same
  - "t is of type T", "t has type T", "type of t is T",
  - "t belongs to type T"
  - Notation: $t : T$ or $t \in T$ or $t :: T$ (in Haskell)

    more commonly, $\Gamma \vdash t : T$

    where Γ is the context, or typing environment

- What are the types of expression x+y below?

```
float f(float x, float y) { return x+y; }
int g(int x, int y) { return x+y; }
```

$x : float, y : float \vdash x+y : float$

$x : int, y : int \vdash x+y : int$

Type Checker As a Function

Type checker is a function that takes a program as its input (as an AST) and returns true or false, or a new AST, where each sub-expression is annotated with a type, function overloads resolved, etc.

Examples of different forms of type checking functions:

```
checkStmt :: Env -> Stmt -> ( Bool, Env )
checkExpr :: Env -> Expr -> Type
```

Defining a Type System with Informal Rules – Example Type Rules

- All referenced variables must be declared
- All declared variables must have unique names
- The + operation must be called with two expressions of type `int`, and the resulting type is `int`

Defining a Type System with Informal Rules – Example Type Check Statement

- Skip is always well-formed
- An assignment is well-formed if
  - its target variable is declared,
  - its source expression is well-formed, and
  - the declared type of the target variable is the same as the type of the source expression
- A conditional is well-formed if its test expression has type bool, and both then and else branches are well-formed statements

Defining a Type System with Informal Rules - Example Type Check Statement (Cont.)

- A while loop is well-formed if its test expression has type bool, and its body is a well-formed statement
- A block is well-formed if all of its statements are well-formed
- A variable declaration is well-formed if the variable has not already been defined in the same scope, and if the type of the initializer expression is the same as the type of the variable

Defining a Type System Using Formal Language

Common way to specify type systems is using natural deduction style rules – "inference rules"

$$\frac{A1 \quad \ldots \quad An}{B}$$

Example:

$$\frac{A \wedge B}{B} \qquad \frac{A \Rightarrow B \quad A}{B}$$

(Do they look/sound familiar?)

Type Rules – Example

A conditional is well-formed if its test expression has type bool, and both then and else branches are well-formed statements

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s1 : \text{ok} \quad \Gamma \vdash s2 : \text{ok}}{\Gamma \vdash \text{if } e \; s1 \; s2 : \text{ok}}$$