

# CS558 Programming Languages

Winter 2008

Lecture 11

We divide the universe of values according to **types**; a **type** is:

- a **set** of values; and
- a collection of **operations** defined on those values.

In practice, important to know how values are **represented** and how operations are **implemented** on real hardware.

Examples:

**Integers** (represented by machine integers) with the usual arithmetic operations (implemented by corresponding hardware instructions).

**Booleans** (represented by machine bits or bytes) with operators `and`, `or`, `not` (implemented by hardware instructions or code sequences).

**Arrays** (represented by contiguous blocks of machine addresses) with operations like `fetch` and `update` (implemented by address arithmetic and indirect addressing).

**Strings** (represented how?) with operations like `concatenation`, `substring extraction`, etc. (implemented how?)

## HARDWARE TYPES

Machine language doesn't distinguish types; all values are just bit patterns until **used**. As such they can be loaded, stored, moved, etc.

But certain **operations** are supported directly by hardware; the operands are thus implicitly typed.

Typical hardware types:

- **Integers** of various sizes, signedness, etc. with standard arithmetic operations.
- **Floating point** numbers of various sizes, with standard arithmetic ops.
- **Booleans** with conditional branch operations.
- **Pointers** to values stored in memory.
- **Instructions**, i.e., code, which can be executed.
- Many others are possible, e.g., binary coded decimal.

Details of behavior (e.g., numeric range) are **machine-dependent**, though often subject to **standards** (e.g., IEEE floating point, Unicode characters, etc.).

## LANGUAGE PRIMITIVE TYPES

**Primitive types** of a language are those whose values cannot be further broken down by user-defined code; they can be managed only via operators built into the language.

Usually includes hardware types plus others that can easily mapped to a hardware type.

Example: **enumeration** types are usually mapped to integers.

Numeric types only **approximate** behavior of true numbers. Also, they often inherit machine-dependent aspects of machine types, causing serious **portability** problems.

Example: Integer arithmetic in most languages.

Partial counterexample: Numerics in LISP.

## COMPOSITE TYPES

**Composite types** are built from existing types using **type constructors**.

Typical composite types include **records**, **unions**, **arrays**, **functions**, etc.

**Abstractly**, such type constructors can be seen as mathematical operators on underlying **sets** of simpler values. A small number of set operators suffices to describe most useful type constructors:

**Cartesian product** ( $S_1 \times S_2$ )

- records, tuples, C structs

**Sum or (disjoint) union** ( $S_1 \oplus S_2$ )

- enumerations, Pascal variant records, C unions

**Mapping** ( $S_1 \rightarrow S_2$ )

- arrays, association lists, functions

In addition, we often need a way to represent **recursive structures** such as lists and trees.

## COMPOSITE TYPE REPRESENTATION

**Concretely**, each language defines the internal **representation** of values of the composite type, based on the type constructor and the types used in the construction.

Example: The fields of a record might occupy successive memory addresses (perhaps with some alignment restrictions). The total size of the record is (roughly) the sum of the field sizes.

Often a range of representations are possible, from highly packed to highly indirected. There's often a tradeoff between space and access time.

Example: Arrays of booleans can be efficiently packed using one bit per element, but this makes it more complicated to read or set an element.

## STATIC TYPECHECKING

HLL's differ from machine language in that explicit types appear and type violations are ordinarily caught at some point.

**Static typechecking** is most common.

- Types are associated with identifiers (esp. variables, parameters, functions).
- Every use of an identifier can be checked for type-correctness at compile time.
- "Well-typed programs don't go wrong."
- Compiler can optimize representations of values used at runtime.

## DYNAMIC TYPECHECKING

**Dynamic typechecking** occurs in Lisp, Scheme, Smalltalk, many scripting languages, etc.

- Types are attached to values (usually as explicit tags).
- The type associated with identifiers can vary.
- Correctness of operations can't (in general) be checked until runtime.
- Type violations become checked runtime errors.
- Optimized representation hard.

The main goal of a type system is to characterize programs that won't "go wrong" at runtime.

Informally, we want to avoid programs that confuse types, e.g., by trying to add booleans to integers, or take the square root of a string.

Formally, we can give a set of **typing rules** (sometimes called as **static semantics**) from which we can derive **typing judgments** about program fragments. (This should sound familiar!)

Each judgment has the form

$$TE \vdash e : t$$

Intuitively this says that expression  $e$  has type  $t$ , under the assumption that the type of each free variable in  $e$  is given by the *type environment*  $TE$ .

The key point is that an expression is well-typed **if-and-only-if** we can derive a typing judgment for it.

Consider our usual simple imperative language, and suppose we have just two types, `Int` and `Bool`.

As before, we write  $TE(x)$  for the result of looking up  $x$  in  $TE$ , and  $TE + \{x \mapsto t\}$  for the type environment obtained from  $TE$  by extending it with a new binding from  $x$  to  $t$ .

Here is a suitable set of typing rules:

$$\frac{x \in \text{dom}(TE)}{TE \vdash x : TE(x)} \text{ (Var)}$$

$$\frac{}{TE \vdash i : \text{Int}} \text{ (Int)}$$

$$\frac{TE \vdash e_1 : \text{Int} \quad TE \vdash e_2 : \text{Int}}{TE \vdash (+ e_1 e_2) : \text{Int}} \text{ (Add)}$$

## TYPING RULES (2)

$$\frac{TE \vdash e_1 : \text{Int} \quad TE \vdash e_2 : \text{Int}}{TE \vdash (<= e_1 e_2) : \text{Bool}} \text{ (Leq)}$$

$$\frac{TE \vdash e_1 : t_1 \quad TE + \{x \mapsto t_1\} \vdash e_2 : t_2}{TE \vdash (\text{local } x e_1 e_2) : t_2} \text{ (Local)}$$

$$\frac{TE(x) = t \quad TE \vdash e : t}{TE \vdash (:= x e) : t} \text{ (Assgn)}$$

$$\frac{TE \vdash e_1 : \text{Bool} \quad TE \vdash e_2 : t \quad TE \vdash e_3 : t}{TE \vdash (\text{if } e_1 e_2 e_3) : t} \text{ (If)}$$

$$\frac{TE \vdash e_1 : \text{Bool} \quad TE \vdash e_2 : t}{TE \vdash (\text{while } e_1 e_2) : \text{Int}} \text{ (While)}$$

## FORMALIZING TYPE SAFETY

The typing rules are just (another) formal system in which judgments can be derived. How do we connect this system with our slogan that "well-typed programs don't go wrong" ?

First we need an auxiliary judgment system assigning types to values, written  $\models v : t$ .

For example, we would have  $\models i : \text{Int}$  for every integer  $i$ ,  $\models \text{true} : \text{Bool}$ , and  $\models \text{false} : \text{Bool}$

We also add a special value `error` which does not belong to any type:  $\not\models \text{error} : t$

We extend this notation to environments and stores, and write

$$\models E, S : TE$$

iff  $\text{dom}(E) = \text{dom}(TE)$  and  $\models S(E(x)) : TE(x), \forall x \in \text{dom}(E)$ .

FORMALIZING TYPE SAFETY (2)

Recall our formal **dynamic semantics** for our language, defined using  $\Downarrow$  judgments. In our previous definition, expressions corresponding to runtime errors simply had no applicable rule (they were “stuck.”). Let’s change the system slightly by adding new rules so that all expressions corresponding to runtime errors evaluate to `error` instead.

Now, if everything has been defined correctly, we should be able to prove a **theorem** roughly like this:

If  $TE \vdash e : t$  and  $\models E, S : TE$  and  $\langle e, E, S \rangle \Downarrow \langle v, S' \rangle$  then  $\models v : t$ .

In other words, well-typed programs evaluate to values of the expected type; so in particular, they can’t evaluate to `error`, which belongs to no type.

We can turn the typing rules into a recursive **typechecking algorithm**.

A typechecker is very similar to the **evaluators** we have already built:

- it is parameterized by a type environment;
- it dispatches according to the syntax of the expression being checked (note that there is exactly one rule for each form);
- it calls itself recursively on sub-expressions;
- it returns a type.

There are some differences, though. For example, a typechecker always examines **both** arms of a conditional (not just one). If we consider a language with **functions**, the typechecker processes the body of each function only once, no matter how many times the function is called.

Note that most languages require the types of function parameters and return values to be **declared** explicitly. The typechecker can use this declaration to check that applications of the function are correctly typed, and **separately** checks that the body of the function is correctly typed.