CS321 Languages and Compiler Design I

Fall 2013

Week 8: Types

Andrew Tolmach

Portland State University

THE TYPE ZOO

```
int x = 17

Z[1023] := 99;

double e = 2.81828

type emp = {name:  string, age:  int}

class Foo extends Bar { ...  }

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

String s = "abc"

type days = set of day

fold ::  (a -> b -> b) -> [a] -> b -> b

’a btree = LEAF of ’a | NODE of ’a * ’a btree * ’a btree
```

Organizing the Zoo

Mathematical View $\Longleftrightarrow$ Programming Language View $\Longleftrightarrow$ Machine View

Programming Language view: Types classify values and operations

Mathematical View: Types are sets of values

Machine View: Types describe memory layout of values

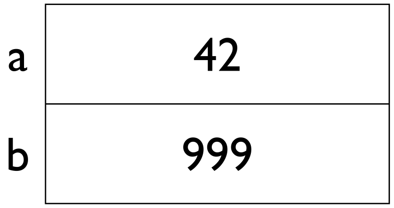
Example: Integers

$\mathbb{Z} \Longleftrightarrow$ `int` $\Longleftrightarrow$

+/-	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1

Example: Records

$$\mathbb{Z} \times \mathbb{R} \iff \{\texttt{a:int,b:int}\} \iff$$



a	42
b	999

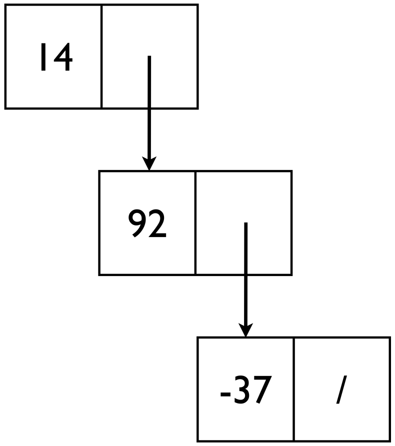
Example: Arrays

$$\mathbb{N} \to \mathbb{Z} \iff \texttt{int[]} \iff$$

0	18
1	22
2	
3	327899
4	-12
.	.
.	.
.	.
98	107
99	2222

Example: Sequences

$\mathbb{Z}^* \iff$ `[Int]` $\iff$



OTHER USEFUL AXES FOR CLASSIFICATION

Primitive Types vs. User-defined Types

- Is the type “built in” to the language?

Atomic Types vs. Type Constructors

- Are values constructed out of smaller parts?

Abstract Types vs. Concrete Types

- Are value representations visible to the programmer?

Mutable Types vs. Immutable Types

- Can values be modified once created?

Reference Types vs. Non-reference Types

- What are the semantics of assignment?

etc.

Machine 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 type-specific **operations** are supported directly by hardware; the operands are in some sense implicitly typed.

Typical machine-level types:

- **Integers** of various sizes, signedness, etc. with standard arithmetic operations.
- **Bit vectors** of various sizes, with bit-level logic operations (`and`, `or`, etc.)
- **Floating point** numbers of various sizes, with standard arithmetic operations.
- **Pointers** to values stored in memory, with arithmetic operations.
- **Instructions**, i.e., code, which can be executed.

There is no abstraction at machine level: programs can inspect individual words and bits of any value.

PRIMITIVE ATOMIC TYPES

Most higher-level languages provide some **built-in atomic** types.

- e.g. in Java: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`

Built-in type names and operators are keywords or part of initial environment

Language may have special syntax for writing literal values (e.g. numbers, character strings)

Atomic types are usually **abstract**: programs cannot inspect internal details of value representation. e.g.,

- usually cannot extract mantissa or exponent from a floating point number (C/C++ is an exception)
- can't tell what convention is used to encode booleans as numbers (C/C++ is an exception)

Atomic types are often closely based on standard machine-level types, giving them an obvious representation and (usually) efficient implementation for operations

NUMBERS: MACHINE VS. MATHEMATICS

Atomic types often correspond to simple mathematical concepts, but the correspondence is not exact. This can be confusing and dangerous, especially for the numeric types!

Example: in most languages, the primitive integer type is stored in a fixed-size hardware word, but the set of mathemetical integers $\mathbb{Z}$ is unbounded

- e.g., 32 bit machine word can only hold values in the range $[-2^{31}, 2^{31} - 1]$

- and word size may vary on different machines, hurting portability

- typically, arithmetic operations will silently overflow

Example: hardware floating point values are rational number of restricted range; they can only approximate arbitrary real numbers

- each numeric operation contributes error to final result

- even after standardization to IEEE754, different hardware (e.g. x86 vs ARM) can still produce different results (the standard is too flexible)

CONSTRUCTED TYPES

Many languages also provide a set of mechanisms for **constructing** new, user-defined types from existing types.

- e.g. Java has array and class definition mechanisms

Each type constructor comes with corresponding mechanisms for:

- constructing values
- inspecting values
- (for mutable types) modifying values

For example, in Java:

- arrays are constructed using `new` and an optional list of initializers; their elements are inspected and modified using subscript notation (e.g. `a[i]`).
- class instances (objects) are constructed using `new` and defined constructor methods; object fields can be accessed using dot notation (e.g. `p.x`).

Abstraction for Constructed Types

Are constructed types **abstract**?

• User-defined types can't be completely abstract, because we must have access to their components in order to write operations over the type.

• But many languages have mechanisms for limiting component access in order to enforce some level of abstraction, e.g. Java's `private` fields

The line between built-in and user-defined types is not always clear-cut.

• e.g, in Java, the `String` type is really just an ordinary class that happens to be provided in the standard library, except that there is special syntax for string literals and concatenation.

• Built-in constructed type values usually need to be kept abstract in order to guarantee that they behave as specified.

We will examine data abstraction mechanisms in detail next week.

MATHEMATICAL VIEW OF TYPE CONSTRUCTORS

It can be enlightening to view type constructors as operators on the underlying **sets** represented by the component types.

Early on in the history of programming languages, it became clear that a small number of type operators suffices to describe most useful data structures:

- Cartesian product ($S_1 \times S_2$)
- Disjoint union ($S_1 + S_2$)
- Mapping (by explicit enumeration or by formula) ($S_1 \rightarrow S_2$)
- Set ($\mathcal{P}^S$)
- Sequence (S^*)
- Recursive definition ($\mu s.T(s)$)

Representation of Constructed Types

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.

Historically, most languages have provided just a few constructors, usually with the property that constructed values can be represented and accessed **efficiently** on conventional hardware.

For conventional languages, this is the short list:

- **Records**
- **Unions**
- **Arrays**

Many languages also support manipulation of **pointers** to values of these types, in order to allow operating on data “by reference” and to support recursive structures.

Records = Cartesian Products

Records, tuples, “structures”, etc. Nearly every language has them.

“Take a bunch of existing types and choose one value from each.”

Examples (Ada Syntax)

```
type EMP is
  record
    NAME : STRING;
    AGE : INTEGER;
  end record;

E: EMP := (NAME => "ANDREW", AGE => 99);
```

(ML syntax):

```
type emp = string * int  (unlabeled fields)
val e : emp = ("ANDREW",99);

type emp =
   {name: string, age: int}  (labeled fields)
val e : emp = {name="ANDREW",age=99};
```

Records (continued)

Standard operations: construction, selection, selective update.

Representation: Fields occupy successive memory addresses (perhaps with some padding to maintain hardware-required alignments), so total size is (roughly) sum of field sizes.

Each field lives at a known static offset from the beginning of the record, allowing very fast access using pointer arithmetic.

Because records may be large, they are often manipulated by reference, i.e., represented by a pointer. The fields within a record may also be represented this way.

Disjoint Unions

Variant records, discriminated records, unions, etc.

“Take a bunch of existing types and choose one value from one type.”

Introduced in Pascal:

```
type RESULT = record
                case found : Boolean of
                  true: (value:integer);
                  false: (error:STRING)
                end;

function search (...) : RESULT;
...
```

Generally behave like records, with **tag** as an additional field.

A variant value is represented by the tag following by the representation of the particular variant. Its size is thus bounded by the size of the largest possible variant plus the tag size.

Variant Insecurities

Pascal variant records are **insecure** because it is possible to manipulate the tag independently from the variant contents.

```
tr.value := 101;    { write an integer }
write tr.error;     { but read a string! }

if (tr.found) then begin     { check for integer }
   tr := tr1;       { overwrite with arbitrary RESULT }
   x := tr.value    { might now contain a string }
```

These problems were fixed in Ada by requiring tag and variant contents to be set simultaneously, and inserting a runtime check on the tag before any read of the variant contents.

We will see a better solution to safe variants in this week's lab.

NON-DISCRIMINATED UNIONS

C unions don't even have a tag mechanism: they are just a way to declare that two fields of potentially different type should share the same storage location.

```
union resunion {
    int value;
    char *error;
};
```

The programmer can manually wrap up the union with a tag:

```
struct result {
  int found;  /* boolean tag */
  union resunion u;
}

struct result search (...);
```

Since the connection between tag and union value is informal, this is completely unsafe.

OBJECTS

Objects in class-based OO languages, e.g. Java, can be viewed as a sort of variant record.

- The object's class identifier is stored at the beginning of the record, and acts like a variant tag, distingushing among different subclasses.

- The remaining record fields correspond to the object's instance variables.

The class tag is used to control dynamic dispatch to the class's methods, and (depending on the language) might be accessible more directly (e.g. Java's `instanceof`).

- The class tag cannot be altered after the object is created, so there is no danger of insecurity.

Arrays

The oldest type constructor, found in Fortran (the first real high-level language); crucial for numerical computations.

Basic implementation idea: a **table** laid out in adjacent memory locations permitting **indexed access** to any element in constant time, using the hardware's ability to compute memory addresses.

Mathematically: A finite **mapping** from an **index set** to **element set**.

Index set is nearly always a set of integers $\texttt{0..}n-1$, or some other discrete set isomorphic to such a set.

Multidimensional arrays (**matrices**) can be built in several ways:

- using an index set of tuples of integers (e.g. in Fortran)
- using an element set of arrays (e.g. in C/C++/Java)

ARRAY SIZE

Is the size of an array part of its type? Some older languages (e.g. Fortran) took this attitude, but most modern languages are more flexible, and allow the size to be set independently for each array value when the array is first created:

- as a local variable, e.g., in Ada:

```
function fred(size:integer);
  var bill: array(0..size) of real;
```

- or on the heap, e.g., in Java:

```
int[] bill = new int[size];
```

Arrays are often large, and hence better manipulated by reference.

The major security issue for arrays is **bounds checking** of index values. In general, it's not possible to check all bounds at compile time (though often possible in particular cases). Runtime checks are always possible; this may be costly, but its usually worth it!

FUNCTIONS AND MAPPINGS

Mathematical mappings can also be represented by an algorithmic **formula**.

A **function** gives a “recipe” for computing a **result** value from an **argument** value.

A program function can describe an infinite mapping.

But differs from mathematical function in that:

- it must be specified by an explicit algorithm
- executing the function may have side-effects on variables.

It can be very handy to manipulate functions as first-class values. But most languages put severe limitations on what can be done with functions.

How does one represent a function as a first-class value? In some languages, can just use a **code pointer**. In others, representation must include values of **free variables**, which can affect runtime performance. More on this later in the course.

SEQUENCES

What about data structures of essentially **unbounded** size, such as **sequences** (or **lists**)?

“Take an arbitrary number of values of some type.”

Such data structures require special treatment: they are typically represented by small segments of data linked by pointers, and dynamic storage allocation (and deallocation) is required.

The basic operations on a sequence include

- **concatenation** (especially concatenating a single element onto the head or tail of an existing sequence); and

- **extraction** of elements (especially the head).

An important example is the (unbounded) **string**, a sequence of chars.

Best representation depends heavily on what nature and frequency of various operations. Hard to give single, uniformly efficient implementation.

DEFINING SEQUENCES

Unless the programming language supports sequences directly, the programmer must define them using a **recursive** definition.

For example, a list of integers is either

- **empty**, or
- has a **head** which is an integer and **tail** which is itself a list of integers.

In mathematical notation, we can write

$$list_{\mathbb{Z}} = empty + (\mathbb{Z} \times list_{\mathbb{Z}})$$

or, more compactly,

$$list_{\mathbb{Z}} = \mu t.(empty + (\mathbb{Z} \times t))$$

where the **fixpoint** operator $\mu s.T(s)$ defines the smallest type s such that $s = T(s)$.

Recursive Types

Recursive definitions can be used to define and operate on more complex types, in which the type being defined appears more than once in the definition.

For example, binary trees with integers at internal nodes and leaves could be defined mathematically as

$$bintree = \mu t.(\mathbb{Z} + (\mathbb{Z} \times t \times t))$$

As you know, these trees can be defined in Java using two subclasses to represent the disjoint union, and specifying the common superclass as the type of the subtree fields.

We will see another way to define recursive types of this kind in this week's lab.

VALUES AND REFERENCES

What does the assignment `x := y` actually do at runtime?

If `x,y` are integers, they are probably stored as 32-bit words, and assignment simply copies the word from one memory location (or machine register) to another.

If `x,y` belong to a constructed type (record, array), they are probably stored as a block of words. Assignment might still be defined as copying...

For example, C defines assignment for `struct`s this way, so the program

```
struct emp { char* name; int age; }
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
printf("%d %d", e1.age, e2.age);
```

prints 18, 91.

Values and References (2)

But copying large numbers of words is expensive. Moreover, it is difficult to compile such copies if the size of the value is not statically known.

This is particularly true for recursive structures, which naturally grow without fixed bound (and are commonly allocated on the **heap**).

So many languages represent and manipulate (at least some) composite types by **reference**. That is, a value of such a type is actually a **pointer** to the data, and assignment is just a **shallow copy** of the pointer.

For example, a similar Java program:

```
class emp {  String name;  int age; }
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
System.out.print(e1.age + " " + e2.age);
```

prints 18, 18.

EXPLICIT POINTERS

Many older languages have **pointer types** to enable programmers to construct recursive data structures, e.g., in C:

```
struct intlist {
  int head;
  struct intlist *tail;
}
Intlist mylist =
      (struct intlist *) malloc(sizeof(struct intlist));
...free(mylist)...
```

Note that programmers must make explicit `malloc` (C++: `new`) and `free` calls to manage heap values, and must explicitly manipulate pointers.

Lots of opportunity for **dangling pointer bugs** (failing to realize that a pointer is no longer valid after freeing) and **memory leaks** (failing to free a pointer when it is no longer needed)!

A language that allows dangling pointers cannot be secure!

(Full type safety story in C/C++ is even worse...)

Automated Heap Management

Many modern languages, such as Java, Python, Haskell, etc., **implicitly** allocate space for composite values (records, disjoint unions, arrays) on the heap.

All such values are represented by references (pointers) into the heap, but programmers can't manipulate the pointers directly. In particular, they can't explicitly deallocate records (or objects) from the heap.

Instead, these languages provide automatic **garbage collection** of unreachable heap values, thus avoiding both **dangling pointer** and **memory leak** bugs.

Garbage collection may add some overhead over manual memory management (though not necessarily). But it allows these languages to be **secure**, which is a huge benefit.

Efficiency vs. Rich Primitive Types

Must language designers be slaves to hardware?

Historically, most mainstream, general-purpose languages have only provided built-in types that can be given simple hardware implementations with **efficient** and **predictable** performance.

But more modern languages are including more complicated primitive types, because they are so useful. Examples:

- "Bignum" representations for arbitrary-precision numbers (Scheme, Python, Haskell, etc.)
- Strings (Java, Python, Perl, etc.)
- "Associative arrays" in which index set can be an arbitrary type rather than just integers (Awk, Perl, JavaScript, Python, etc.)
- Lists (LISP, Scheme, Haskell, Python, etc.)
- Sets (Pascal, SETL, Python, etc.)

TYPE EQUIVALENCE

When do two identifiers have the “same” type, or “compatible” types? We need to know this to do static type checking.

- e.g., if `x` has type t_1 and e has type t_2, when does it make sense to allow the assignment `x :=` e ?

To maintain **type safety** we must insist at a minimum that t_1 and t_2 are **structurally equivalent**.

- Two types are structurally equivalent if they each describe the same set of values.

In languages that define type **names**, we may instead require that t_1 and t_2 be **name-equivalent**.

- Two types are name-equivalent if they have identical names.
- Name equivalence implies structural equivalence, but not vice-versa.

In some languages, values can be given more than one type due to **subtyping**, which can also be defined using **structural** or **nominal** criteria. Then we insist that t_2 be a subtype of t_1.

STRUCTURAL EQUIVALENCE

Structural equivalence is defined inductively:

- Primitive types are equivalent iff they are exactly the same type.

- Cartesian product types are equivalent if their corresponding component types are equivalent. (Record field names may or may not matter.)

- Disjoint union types are equivalent if their corresponding component types are equivalent.

- Mapping types (arrays and functions) are the same if their domain and range types are the same.

- Recursive types are a challenge. Are these two types structurally equivalent?

```
type t1 = { a:int, b: POINTER TO t1 };
type t2 = { a:int, b: POINTER TO t2 };
```

Intuitively yes, but it's (a little) tricky for a type-checking algorithm to determine this!

TYPE NAMES

Question of equivalence is more interesting if language has type **names**, which arise for two main reasons:

- As a convenient shorthand to avoid giving the full type each time, e.g.

```
function f(x:int * bool * real) : int * bool * real = ...
type t = int * bool * real
function f(x:t) : t = ...
```

- As a way of improving program correctness by subdividing values into types according to their meaning **within the program**, e.g.

```
type polar = { r:real, a:real }
type rect = { x:real, y:real }
function polar_add(x:polar,y:polar) : polar = ...
function rect_add(x:rect,y:rect) : rect = ...
var a:polar; c:rect;
a := (150.0,30.0) (* ok *)
polar_add(a,a)  (* ok *)
c := a  (* type error *)
rect_add(a,c) (* type error *)
```

Whole idea here is that some structurally equivalent are treated as **inequivalent**.

Name Equivalence

Simplistic idea: Two types are equivalent iff they have the same **name**.

Supports `polar`/`rect` distinction.

But pure name equivalence is very restrictive, e.g.:

```
type ftemp = real
type ctemp = real
var x:ftemp, y:ftemp, z: ctemp;
x := y; (* ok *)
x := 10.0; (* probably ok *)
x := z; (* type error *)
x := 1.8 * z + 32.0; (* probably type error *)
```

Different types now seem **too** distinct; can't even convert from one form of real to another.

NAME EQUIVALENCE (CONTINUED)

Also: what about unnamed type expressions?

```
type t = int * int
procedure f(x: int * int) = ...
procedure g(x: t) = ...
var a:t = (3,4)
g(a); (* ok *)
f(a); (* ok or not ?? *)
```

Because of these problems with pure name equivalence, most languages use **mixed** solutions.

C TYPE EQUIVALENCE

C uses structural equivalence for array and function types, but name equivalence for `struct`, `union`, and `enum` types. For example:

```
char a[100];
void f(char b[]);
f(a); /* ok */

struct polar{float x; float y;};
struct rect{float x; float y;};
struct polar a;
struct rect b;
a = b; /* type error */
```

A type defined by a `typedef` declaration is just an abbreviation for an existing type.

Note that this policy makes it easy to check equivalence of recursive types, which can only be built using `struct`s.

```
struct fred {int x; struct fred *y;} a;
struct bill {int x; struct fred *y;} b;
a = b; /* type error */
```

JAVA TYPE EQUIVALENCE

Java uses nearly strict name equivalence, where names are either:

- One of eight built-in **primitive** types (`int`,`float`,`boolean`, etc.), or
- Declared classes or interfaces (**reference** types).

The only non-trivial type expressions that can appear in a source program are **array** types, which are compared structurally, using name equivalence for the ultimate element type. Java has no mechanism for type abbreviations.

Java types form a **subtyping** hierarchy:

- If class `A` extends class `B`, then `A` is a subtype of `B`.
- If class `A` implements interface `I`, then `A` is a subtype of `I`.
- If numeric type `t` can be coerced to numeric type `u` without loss of precision, then `t` is a subtype of `u`.

If T_1 is a subtype of T_2, then a value of type T_1 can be used wherever a value of T_2 is expected.