To Type or not to Type:

Why is there a Question?

What is Type?

- *Value* and *Objects*...
  - the entities with which we compute
- Types
  - a specification of what can meaningfully be done to, or done by, those Values or Objects
- Examples:

v, w: Integer	⇒ can add v and w
c: Char, v: Integer	⇒ can't add v and c
s: Stack, e: Element	⇒ push e onto s

An older view:

- Types describe data layouts in memory
  - this is essentially the meaning of type in C
- This view of type is deprecated
  - it's important to distinguish between implementation and interface
- In a class-based language
  - The class describes the implementation
  - The type describes the protocol (a.k.a. the interface)

Varieties of Typing

- Most languages compute with typed *values*
  - Tkl, Lisp, Snobol, Csh
  - most machines too (float *vs.* int *vs.* long)
- The distinctions between languages:
  - whether the types of *identifiers* are fixed
  - whether the types of *identifiers* and *expressions* are inferred
  - whether there are checks between the programmer's intent and the executing code
    - compare Lisp to BCPL (or C)

Typing Expressions

Static

- Types are known and checked at compile time.
  - explicitly typed, *e.g.*, Java
  - implicitly typed, *e.g.*, ML, Haskell

Dynamic

- Types are known and checked at runtime

Whether types are checked

Untyped:

- The programmer is on her own!

Typed:

- syntactic elements of language—the variables and expressions—are assigned types by the programmer and by inspecting the code.
- The *type system* is the set of rules that let us do this assignment, or check the programmer's assignment.

Sample Type Rules

Inference

$$\frac{o\!:\!\mathrm{obj}\{\ldots,\ \phi\!:\!\alpha \rightarrow \rho,\ \ldots\},\ a\!:\!\alpha}{o.\phi(a)\!:\ \rho}$$

Checking

$$\frac{\phi\!:\!\alpha \rightarrow \rho,\ \mathrm{r}\!:\!\rho,\ \mathrm{ok}(s_0),\ \mathrm{ok}(s_1),\ \phi = \{s_0;\ s_1;\ \ldots;\ \mathrm{return\ r}\}}{\mathrm{ok}(\phi)}$$

Each language has its own type system

= set of rules for checking and inference

Type Systems

- Type systems exist for languages, logics, inter-operation frameworks (*e.g.*, COM, CORBA)
- "Healthiness condition"
  - When an expression *e* is determined to have type *t* (via the type system, statically) then …
  - when *e* is evaluated (at run time), the resulting value will have type *t*.

    e.g., a+b/c

- The subject-reduction property
  - When an expression is “reduced” (*i.e.*, evaluated), the type of the reduced form conforms to the type of the expression
  - In other words: soundness
- Sample Rule applications

a: int
b: int
c: int
div: int x int → rat
plus: int x int → int

plus(a, b): int
div(plus(a, b), c) : rat

Typed and Untyped Languages

Explicitly typed

- all functions and variables are given types (signatures) by the programmer.
  - *e.g.,* Java:

```
Person p;
Student s;
```

Implicitly typed

- all functions and variables are given types by the compiler. The type (signature) is the most general signature that is
  - expressible in the type language
  - consistent with the code that the programmer wrote
- Examples

cons

~~concat~~ : $int \times int\ list \rightarrow int\ list$
$char \times char\ list \rightarrow char\ list$
$\Lambda\alpha.\ \alpha \times \alpha\ list \rightarrow \alpha\ list$

  - Different type languages have different expressiveness.

Type Inference

- Type inference (or type reconstruction) is the process by which the compiler assigns types to expressions
  - using the type rules for the language.
- All compilers use some inference

  a.append(“Hi“).append(“ “).append(“there”)

  x / (n + 1)
- Some languages do a great deal (ML, Haskell)

Untyped Languages

- Examples: Lisp, Csh, Smalltalk, Self
  - any variable can name data of any type (including methods!)
  - the type of a variable may vary from one program point to another:

```
   ...
s find: x          <——
   ...
x match: y         <——
   ...
f reportOn: x      <——
```

The Rôle of Types

- Types characterize what *can be done* to values or objects
- Used in conjunction with your code (which states what *you want done* to your values and objects) provides ***redundancy***:
  - if what you want done is consistent with what the types say *can* be done, your code is more likely to be doing something sensible.
  - Types are an explicit statement about intent:

list xs;	xs will behave like a list and all actions on xs will be consistent with action on lists.

Types in a Value Oriented Language

- Values are bit patterns.

  0011010001110011

  - an int? a date? a uid? what is it?
- a types defines a set of operations that act as interpreters of the bit patterns.

```
Date d;
nextDay(d);
previousDay(d);

String s, t;
strcat(s,t);
streq(s,t);
```

Types in OO Languages

- We can't *see* bit patterns any more!
- Every object is a package: bit pattern + set of operations.
- Can't see the bit pattern except through the set of operations.
  - The action of *strcat*, *streq*, *substring*, *etc*. are entirely encapsulated in the *String* object.

What can Go Wrong Without Types?

- With values: an incorrect operation can be applied to a bit pattern.

```
Date d; String s;
strcat(d,s);
```

  - the code now treats *d* as a String even though it isn't.
- The Result?

Chaos!

In a precise technical sense!

- The resulting state cannot be determined from the definition of the language!
  - We would have to know the details of how dates are represented
  - This ought to be machine dependent
- the failure of the program may not be apparent until much later.

In a Statically typed, Value-oriented Language

This program could never be run!

- Only “well-typed” programs are legal
  – an application of a function to a value is only well typed if it can never be applied to a value of an incorrect type.
- “Well-typed programs don’t go wrong”
  – in ways that can’t be understood in terms of the language itself

In Dynamically Typed, Value-Oriented Languages

- A run-time type error occurs
  - "attempt to apply operation *strcat* to a *date*"
- A type error usually indicates a conceptual problem in the algorithm
  - it can be corrected at the level of the programming language and rather than at the level of the bits.
  - The type structure of the program reflects the conceptual model of the solution.

In an Object-Oriented Language

- The client asks an object to perform an operation. What kind of error can occur?
  - the requested operation is not one of the supported operations defined by the object.
  - the result is *Message Not Understood*
- This occurs in *both* typed and untyped OO languages.
- This is better than a jumble of bits!

Is it good enough?

- Yes… because we don't "do" an incorrect message.
- No... we may travel a long way from the original conceptual error before we finally get *message not understood*.
  - We have to wait until the message is sent before we get the warning of our error
- Typed OO languages
  - Let us find all potential *message not understood* errors *before* we ever run the program.

Costs of typed languages

- Syntactic noise
- Some programs that will *never* generate an error will be type-incorrect
  - the type language is not expressive enough to handle the type.
- In practice, we need a dynamic type system too!
  - Java casts in and out of Vectors and tests into Arrays
  - Objects that arrive at run-time
  - The “right” type system is an engineering compromise

- More development time (and more information needed) in order to compile programs written in typed languages...
  - incremental development must include type information on all pieces, even those not written yet.
  - While development time increases, the increase in the quality of correct code usually more than compensates for the time and effort.
- Higher-order type systems are being investigated to increase the expressiveness of the object language so that more programs can be well typed.