Values and Types

Values are the entities or objects manipulated by programs. We divide the universe of values according to types; a type is:

- a set of values; and
- a set of operations defined on those values; and/or
- a set of valid contexts.

Examples:

Integers with the usual arithmetic operations.
Booleans with operators and, or, not and valid as arguments to conditional operations.
Arrays with operations like fetch and store.
Sets with operations like membership testing, union, intersection, etc.

Characterizing Types and Values

- Set of values.
- Defined operations.
- Permitted contexts where values can be used.

(In particular, values that can be anonymously constructed, used in expressions, passed to and from procedures, and assigned into variables are called first-class values.)

- How values are represented and operations are implemented.
- How literal values are described.

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.
- **Booleans** with boolean and conditional operations. (Usually just a special view of integers.)
- **Floating point** numbers of various sizes, with standard arithmetic operations.
- **Characters** with i/o 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, ASCII characters).
Primitive (Atomic, Basic) Values and Types

Primitive values cannot be further broken down by user-defined code; they can be managed only via operators built into the language.

Typical primitive types include integers, floats, characters, booleans, enumerations, etc.

Usually closely allied to hardware types.

Example: enumerations.

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 counter-example: Numerics in Lisp.

Composite Values

Composite values are constructed from more primitive values, which can usually later be selected back from the composite, and perhaps selectively updated.

Example: Records (Ada syntax)

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

```ada
E: EMP := (NAME => "ANDREW", AGE => 99);
if E.NAME = "FRED" ...
E.AGE := 88;
```

Other common composite types include unions and arrays.

In statically typed languages, it is generally necessary to declare new composite types (e.g., EMP) before defining composite values (e.g., E). The type definition indicates how the type is constructed from more primitive types, using one of a few predefined type constructors (e.g., record).

Static and Dynamic Typing

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

Static typing is most common.

- Types are associated with identifiers (esp. variables, parameters, functions).
- Can be statically checked, if language and compiler allow.
- Compiler can optimize representations of values used at runtime.

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

- Types are attached to values (usually explicitly).
- The type associated with identifiers can vary.
- Correctness of operations can’t generally be checked until runtime.
- Optimized representation hard.

Flexibility of Dynamic Typing

Static typing offers the great advantage of catching errors early, and generally supports more efficient execution.

Why ever settle for dynamic typing?

- Simplicity. For short or simple programs, it’s nice to avoid the need for declaring the types of identifiers.
- Flexibility. Dynamic typing allows container types, like lists or arrays, to contain mixtures of values of arbitrary types.

Note: Some statically-typed languages (like Standard ML) offer alternative ways to approach these goals, via type inference and polymorphic typing.
Example

Consider a program for reading and randomly permuting an array. If we want it to work only on integers, static typing is great:

```pascal
VAR a: ARRAY 100 of INT;
PROCEDURE read(x :ARRAY 100 of INT) ...;
PROCEDURE permute(y: ARRAY 100 of INT) ...;
BEGIN read(a); permute (a); END
```

Program can be statically type-checked, a can be stored in a compact format, and both `read` and `permute` knows what this format is.

But suppose we’d like the same program to work on arrays containing both integers and floats. In a dynamically typed language we might write:

```pascal
VAR a: ARRAY 100;
PROCEDURE read(x: ARRAY 100) ...;
PROCEDURE permute(y: ARRAY 100) ...;
BEGIN read(a); permute(a) END
```

Now the elements of `a` can be of different types, so long as `read` and `permute` know how to check for and operate correctly on values of these types. Generally, each array element will need to be tagged with the correct type.

Static Type Checking

Basic task: compare the type information on identifier declarations with the way the identifiers are used.

Can do with an attribute grammar manipulating a static environment containing types.

Let \[(id,typ) : E \rightarrow E_1 E_2\]

\[
E_1.env := E.env;\]

\[
\text{require}(E_1.txt == E.txt);\]

\[
E_2.env := \text{extend}(E.env,E.id,E.txt);\]

\[
E.txt = E_2.txt
\]

Add : \[E \rightarrow E_1 E_2\]

\[
E_1.env := E.env; E_2.env := E.env;\]

\[
\text{require}(E_1.txt = \text{int});\]

\[
\text{require}(E_2.txt = \text{int});\]

\[
E.txt = \text{int};\]

Eq : \[E \rightarrow E_1 E_2\]

\[
E_1.env := E.env; E_3.env := E.env;\]

\[
\text{require}(E_1.txt = E_3.txt);\]

\[
E.txt = \text{bool};\]

...

Id : \[(id) : E \rightarrow \epsilon\]

\[
E.txt := \text{lookup}(E.env,E.id)
\]

This is very similar to ordinary evaluation, except that each expression is typed only once (so no iteration or recursion; typechecking always terminates!)