Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 23

- Type systems
- Type safety
- Type checking
  - Equivalence, compatibility and coercion
- Primitive and composite types
  - Discrete and scalar types, tuples and records

What is a Data Type?

- A (*data*) *type* is a homogeneous collection of values, effectively presented, equipped with a set of operations which manipulate these values
- Various perspectives:
  - collection of values from a "domain" (the denotational approach)
  - internal structure of a bunch of data, described down to the level of a small set of fundamental types (the structural approach)
  - collection of well-defined operations that can be applied to objects of that type (the abstraction approach)

Advantages of Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Document intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as 3 + true – "Bill"
- Support implementation and optimization
  - Example: short integers require fewer bits
  - Access components of structures by known offset

Type system

A **type system** consists of

1. The set of predefined types of the language.
2. The mechanisms which permit the definition of new types.
3. The mechanisms for the control of types, which include:
    1. **Equivalence rules** which specify when two formally different types correspond to the same type.
    2. **Compatibility rules** specifying when a value of a one type can be used in a context in which a different type would be required.
    3. Rules and techniques for **type inference** which specify how the language assigns a type to a complex expression based on information about its components.
4. The specification as to whether (or which) constraints are **statically** or **dynamically checked**.

Type errors

- A *type error* occurs when a value is used in a way that is inconsistent with its definition
- Type errors are *type system* (thus *language*) *dependent*
- Implementations can react in various ways
  - Hardware interrupt, *e.g. apply fp addition to non-legal bit configuration*
  - OS exception, *e.g. page fault when dereferencing* 0 in C
  - Continue execution with possibily wrong values
- Examples
  - Array out of bounds access
    - C/C++: runtime errors
    - Java: dynamic type error
  - Null pointer dereference
    - C/C++: run-time errors
    - Java: dynamic type error
    - Haskell/ML: pointers are hidden inside datatypes
      - Null pointer dereferences would be incorrect use of these datatypes, therefore static type errors

Type safety

- A language is ***type safe*** when no program can violate the distinctions between types defined in its type system
- In other words, a type system is safe when no program, during its execution, can generate an unsignalled type error
- Also: if code accesses data, it is handled with the type associated with the creation and previous manipulation of that data

Safe and *not safe* languages

- Not safe: C and C++
  - Casts, pointer arithmetic
- Almost safe: Algol family, Pascal, Ada.
  - Dangling pointers.
    - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p.
    - No language with explicit deallocation of memory is fully type-safe.
- Safe or Strongly Typed: Lisp, Smalltalk, ML, Haskell, Java, JavaScript
  - Dynamically typed: Lisp, Smalltalk, JavaScript
  - Statically typed: ML, Haskell, Java

Type checking

- Before any operation is performed, its operands must be **type-checked** to prevent a type error. E.g.:
  - *mod* operation: check that both operands are integers
  - *and* operation: check that both operands are booleans
  - indexing operation: check that the left operand is an array, and that the right operand is a value of the array's index type.

Static vs dynamic typing (1)

- In a **statically typed** PL:
  - all variables and expressions have fixed types (either stated by the programmer or inferred by the compiler)
  - all operands are type-checked at *compile-time*.
- Most PLs are statically typed, including Ada, C, C++, Java, Haskell.

Static vs dynamic typing (2)

- In a **dynamically typed** PL:
  - values have fixed types, but variables and expressions do not
  - operands must be type-checked when they are computed at *run-time*.
- Some PLs and many scripting languages are dynamically typed, including Smalltalk, Lisp, Prolog, Perl, Python.

Example: Ada static typing

- Ada function definition:

```
function is_even (n: Integer)
      return Boolean is
begin
   return (n mod 2 = 0);
end;
```

The compiler doesn't know the value of n. But, knowing that n's type is Integer, it infers that the type of "n **mod** 2 = 0" will be Boolean.

- Call:

```
p: Integer;
…
if is_even(p+1) …
```

The compiler doesn't know the value of p. But, knowing that p's type is Integer, it infers that the type of "p+1" will be Integer.

- Even without knowing the values of variables and parameters, the Ada compiler can guarantee that no type errors will happen at run-time.

Example: Python dynamic typing

- Python function definition:

```
def even (n):
    return (n % 2 == 0)
```

The type of n is unknown. So the "%" (*mod*) operation must be protected by a run-time type check.

- The types of variables and parameters are not declared, and cannot be inferred by the Python compiler. So run-time type checks are needed to detect type errors.

Static vs dynamic typing

- Pros and cons of static and dynamic typing:
  - Static typing is **more efficient**. Dynamic typing requires run-time type checks (which make the program run slower), and forces all values to be tagged (to make the type checks possible). Static typing requires only compile-time type checks, and does not force values to be tagged.
  - Static typing is **more secure**: the compiler can guarantee that the object program contains no type errors. Dynamic typing provides no such security.
  - Dynamic typing is **more flexible**. This is needed by some applications where the types of the data are not known in advance.
    - JavaScript array: elements can have different types
    - Haskell list: all elements must have same type

Static typing is conservative

- In JavaScript, we can write a function like

```
function f(x) { return x < 10 ? x : x(); }
```

Some uses will produce type error, some will not.

- Static typing must be *conservative*

```
if  (possibly-non-terminating-boolean-expression)
   then  f(5);
   else  f(15);
```

Cannot decide at compile time if run-time error will occur!

- Note: safety is independent of dynamic/static

Type Checking: how does it work

- Checks that each operator is applied to arguments of the right type. It needs:
  - *Type inference*, to infer the type of an expression given the types of the basic constituents
  - *Type compatibility*, to check if a value of type A can be used in a context that expects type B
    - *Coercion rules,* to transform silently a type into a compatible one, if needed
  - *Type equivalence,* to know if two types are considered the same

Type Equivalence

- **Structural equivalence:** unravel all type constructors obtaining type expressions containing only primitive types, then check if they are equivalent
- **Name equivalence**: based on declarations
- Name equivalence more popular today
- But sometimes “aliases” needed

```
-- pseudo Pascal
type Student = record
     name, address : string
     age : integer

type School = record
     name, address : string
     age : integer

x : Student;
y : School;

x:= y;
--ok with structural equivalence
--error with name equivalence
```

```
TYPE stack_element = INTEGER;
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
(* alias *)
        ...
    PROCEDURE push(elem : stack_element);
        ...
    PROCEDURE pop() : stack_element;
...

var st:stack;
st.push(42);      // this should be OK
```

Type compatibility and Coercion

- Type compatibility rules vary a lot
  - Integers as reals OK
  - Subtypes as supertypes OK
  - Reals as integers ???
  - Doubles as floats ???
- When an expression of type **A** is used in a context where a compatible type **B** is expected, an automatic implicit conversion is performed, called **coercion**

Type compatibility and Coercion

- Coercion may change the representation of the value or not
  - Integer → Real *binary representation is changed*

```
{int x = 5; double y = x; …}
```

  - A → B subclasses *binary representation not changed*

```
class A extends B{ … }
{B myBobject = new A(…); … }
```

- Coercion may cause loss of information, in general
  - Not in Java, with the exception of **long** as **float**
- In statically typed languages coercion instructions are inserted during semantic analysis (type checking)
- Popular in Fortran/C/C++, tends to be replaced by overloading and polymorphism

Built-in primitive types

- Typical built-in primitive types:

  Boolean = {*false*, *true*}

  Character = {…, ‘A’, …, ‘Z’, …, ‘0’, …, ‘9’, …}

  Integer = {…, –2, –1, 0, +1, +2, …}

  Float = {…, –1.0, …, 0.0, +1.0, …}

PL- or implementation-defined set of characters (ASCII, ISO-Latin, or Unicode)

PL- or implementation-defined set of whole numbers

PL- or implementation-defined set of real numbers

- *Note:* In some PLs (such as C), booleans and characters are just small integers.
- Names of types vary from one PL to another: not significant.

Terminology

- Discrete types – countable
  - integer, boolean, char
  - enumeration

```
type Color is (red, green, blue);
```

  - subrange

```
type Population is range 0 .. 1e10;
```

- Scalar types - one-dimensional
  - discrete
  - real

Composite types

- Types whose values are *composite*, that is composed of other values (simple or composite):
  - records (unions)
  - Arrays (Strings)
  - algebraic data types
  - sets
  - pointers
  - lists
- Most of them can be understood in terms of a few concepts:
  - Cartesian products (records)
  - mappings (arrays)
  - disjoint unions (algebraic data types, unions, objects)
  - recursive types (lists, trees, etc.)
- Different names in different languages.
- Defined applying *type constructors* to other types (eg *struct, array, record,…*)

An brief overview of composite types

- We review type constructors in Ada, Java and Haskell corresponding to the following mathematical concepts:
  - Cartesian products (records)
  - mappings (arrays)
  - disjoint unions (algebraic data types, unions)
  - recursive types (lists, trees, etc.)

Cartesian products (1)

- In a **Cartesian product**, values of several types are grouped into tuples.
- Let (x, y) be the **pair** whose first component is x and whose second component is y.
- $S \times T$ denotes the Cartesian product of S and T:

  $S \times T = \{ (x, y) \mid x \in S; y \in T \}$

- Cardinality:

  $\#(S \times T) = \#S \times \#T$ ---------- hence the "×" notation

Cartesian products (2)

- We can generalise from pairs to **tuples**. Let $S_1 \times S_2 \times \ldots \times S_n$ stand for the set of all n-tuples such that the ith component is chosen from S_i:

  $$S_1 \times S_2 \times \ldots \times S_n = \{ (x_1, x_2, \ldots, x_n) \mid x_1 \in S_1; x_2 \in S_2; \ldots; x_n \in S_n \}$$

- Basic operations on tuples:
  - **construction** of a tuple from its component values
  - **selection** of an *explicitly-designated* component of a tuple.

so we can select the 1st or 2nd (but not the *i*th) component

- **Records** (Ada), **structures** (C), and **tuples** (Haskell) can all be understood in terms of Cartesian products.

Example: Ada records (1)

- Type declarations:

```
type Month is (jan, feb, mar, apr, may, jun,
     jul, aug, sep, oct, nov, dec);
type Day_Number is range 1 .. 31;
type Date is record
       m: Month;
       d: Day_Number;
     end record;
```

- Application code:

```
someday: Date := (jan, 1);
…
put(someday.m+1); put("/"); put(someday.d);
someday.d := 29; someday.m := feb;
```

record construction

component selection

Example: Haskell tuples

- Declarations:

```
data Month = Jan | Feb | Mar | Apr
     | May | Jun | Jul | Aug
     | Sep | Oct | Nov | Dec
type Date = (Month, Int)
```

- Set of values:

  Date = Month × Integer
  = {*Jan*, *Feb*, …, *Dec*} × {…, –1, 0, 1, 2, …}

- Application code:

```
someday = (jan, 1)
m, d = someday
anotherday = (m + 1, d)
```

tuple construction

component selection (by pattern matching)