Principles of Programming Languages

http://www.di.unipi.it/∼andrea/Didattica/PLP-14/
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 21

• Type systems
• Type safety
• Type checking
  – Equivalence, compatibility and coercion
• Primitive and composite types
  – Discrete and scalar types
  – Tuples and records
  – Arrays
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 (checking) 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 given context.
   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 (and sometimes on the context).
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.* segmentation fault when dereferencing 0 in C
  – Continue execution possibly with 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 (strongly typed) 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** (aka “weakly typed”): 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** (aka “strongly typed”): Lisp, Smalltalk, ML, Haskell, Java, JavaScript
  – Dynamically typed: Lisp, Smalltalk, JavaScript
  – Statically typed: ML, Haskell, Java
Type checking

• To prevent type errors, before any operation is performed, its operands must be type-checked to ensure that they comply with the compatibility rules of the type system
  – \textit{mod} operation: check that both operands are integers
  – \textit{and} operation: check that both operands are booleans
  – \textit{indexing operation}: check that the left operand is an array, and that the right operand is a value of the array’s index type.

• \textbf{Statically typed} languages: (most) type checking is done during compilation

• \textbf{Dynamically typed} languages: type checking is done at runtime
Static vs dynamic typing

• In a **statically typed** PL:
  – all variables and expressions have fixed types (either stated by the programmer or inferred by the compiler)
  – most operands are type-checked at compile-time.

• Most PLs are called “statically typed”, including Ada, C, C++, Java, Haskell, ... even if some type-checking is done at run-time (e.g. access to arrays)

• 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:

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

Knowing that n’s type is Integer, the compiler infers that the type of “n \text{mod} 2 = 0” will be Boolean.

• Call:

```ada
p: Integer;
...
if is_even(p+1) ...
```

Knowing that p’s type is Integer, the compiler 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:

```python
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 type checking

• **Static typing** is more efficient
  – No run-time checks
  – Values do not need to be tagged at run-time

• **Static typing** is often considered more secure
  – The compiler guarantees that the object program contains no type errors. With dynamic typing you rely on the implementation.

• **Dynamic typing** is more flexible
  – 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

• **Note:** type safety is independent of dynamic/static
Static typing is conservative

• In JavaScript, we can write a function like

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

Some uses will produce type error, some will not.

• Static typing must be *conservative*

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

Cannot decide at compile time if run-time error will occur!
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
Towards Type Equivalence: Type Expressions

• **Type expressions** are used in declarations and type casts to define or refer to a type

\[
\text{Type ::= int | bool | ... | X | Tname | pointer-to(Type) | array}(num, \text{Type}) | \text{record}(\text{Fields}) | \text{class}(...) | \text{Type } \rightarrow \text{Type} | \text{Type x Type}
\]

– *Primitive types*, such as **int** and **bool**
– *Type constructors*, such as pointer-to, array-of, records and classes, and functions
– *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions
Graph Representations for Type Expressions

- Internal compiler representation, built during parsing
- Example: `int *f(char*,char*)`

Tree forms

DAGs
Cyclic Graph Representations

Source program

```c
struct Node
{   int val;
    struct Node *next;
};
```

Internal compiler representation of the `Node` type: cyclic graph
Equivalence of Type Expressions

• Two different notions: name equivalence and structural equivalence
  – Two types are \textit{structurally equivalent} if
    1. They are the same basic types, or
    2. They have the form \( \text{TC}(T_1, \ldots, T_n) \) and \( \text{TC}(S_1, \ldots, S_n) \), where \( \text{TC} \) is a type constructor and \( T_i \) is structurally equivalent to \( S_i \) for all \( 1 \leq i \leq n \), or
    3. One is a type name that denotes the other.
  – Two types are \textit{name equivalent} if they satisfy 1. and 2.
On Structural Equivalence

• **Structural equivalence**: unravel all type constructors obtaining type expressions containing only primitive types, then check if they are equivalent

• Used in C/C++, C#

```pseudoPascal
-- 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
```
Structural Equivalence of Recursive Type Expressions

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by (maximally) sharing nodes

```c
struct Node
{   int val;
    struct Node *next;
};
struct Node s, *p;
p = &s; // OK
*p = s; // OK
p = s;  // ERROR
```
On Name Equivalence

- Each **type name** is a distinct type, even when the type expressions that the names refer to are the same.
- Types are identical only if names match.
- Used for Abstract Data Types and by OO languages.
- Used by Pascal (inconsistently).

```pascal
type link = ^node;
var next : link;
  last : link;
  p : ^node;
q, r : ^node;

With name equivalence in Pascal:
p := next       FAIL
last := p       FAIL
q := r          OK
next := last    OK
p := q          FAIL !!!
```
On Name Equivalence

- **Name equivalence**: sometimes “aliases” needed

```pascal
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 checking with attributed grammars
A simple language example

\[ P \rightarrow D ; S \]
\[ D \rightarrow D ; D \]
\[ T \rightarrow \text{boolean} \]
\[ \mid \text{id : } T \]
\[ \mid \text{char} \]
\[ \mid \text{integer} \]
\[ \mid \text{array [ num ] of } T \]
\[ \mid ^{T} \]
\[ S \rightarrow \text{id := } E \]
\[ \mid \text{if } E \text{ then } S \]
\[ \mid \text{while } E \text{ do } S \]
\[ \mid S ; S \]

\[ E \rightarrow \text{true} \]
\[ \mid \text{false} \]
\[ \mid \text{literal} \]
\[ \mid \text{num} \]
\[ \mid \text{id} \]
\[ \mid E \text{ and } E \]
\[ \mid E + E \]
\[ \mid E [ E ] \]
\[ \mid E ^{T} \]

Synthesized attributes
\[ T.\text{type} : \text{type expression} \]
\[ E.\text{type} : \text{type of expression or type_error} \]
\[ S.\text{type} : \text{void if statement is well-typed, type_error otherwise} \]

Pointer to \( T \)
Pascal-like pointer dereference operator
Declarations

\[ D \rightarrow \text{id} : T \]  \{ \text{addtype(id.entry, T.type) } \}
\[ T \rightarrow \text{boolean} \]  \{ T.type := boolean \}
\[ T \rightarrow \text{char} \]  \{ T.type := char \}
\[ T \rightarrow \text{integer} \]  \{ T.type := integer \}
\[ T \rightarrow \text{array [ num ] of } T_1 \]  \{ T.type := array(1..num.val, T_1.type) \}
\[ T \rightarrow \text{^ } T_1 \]  \{ T.type := pointer(T_1) \}

Parametric types:
- type constructor
Checking Statements

$S \rightarrow \text{id} := E \{ S.\text{type} := (\text{if id.type} = E.\text{type} \text{ then void else type_error}) \}$

• Note: the type of \text{id} is determined by scope’s environment:
  \text{id.type} = \text{lookup(id.entry)}

$S \rightarrow \text{if } E \text{ then } S_1 \{ S.\text{type} := (\text{if } E.\text{type} = \text{boolean then } S_1.\text{type else type_error}) \}$

$S \rightarrow \text{while } E \text{ do } S_1 \{ S.\text{type} := (\text{if } E.\text{type} = \text{boolean then } S_1.\text{type else type_error}) \}$

$S \rightarrow S_1 ; S_2 \{ S.\text{type} := (\text{if } S_1.\text{type} = \text{void and } S_2.\text{type} = \text{void then void else type_error}) \}$
Checking Expressions

\[
E \rightarrow \text{true} \quad \{ \; E.\text{type} = \text{boolean} \; \}
\]
\[
E \rightarrow \text{false} \quad \{ \; E.\text{type} = \text{boolean} \; \}
\]
\[
E \rightarrow \text{literal} \quad \{ \; E.\text{type} = \text{char} \; \}
\]
\[
E \rightarrow \text{num} \quad \{ \; E.\text{type} = \text{integer} \; \}
\]
\[
E \rightarrow \text{id} \quad \{ \; E.\text{type} = \text{lookup(id.entry)} \; \}
\]
\[
E \rightarrow E_1 + E_2 \quad \{ \; E.\text{type} := (\text{if } E_1.\text{type} = \text{integer} \text{ and } E_2.\text{type} = \text{integer} \text{ then integer else type_error}) \; \}
\]
\[
E \rightarrow E_1 \text{ and } E_2 \quad \{ \; E.\text{type} := (\text{if } E_1.\text{type} = \text{boolean} \text{ and } E_2.\text{type} = \text{boolean} \text{ then boolean else type_error}) \; \}
\]
\[
E \rightarrow E_1 [ \; E_2 \; ] \quad \{ \; E.\text{type} := (\text{if } E_1.\text{type} = \text{array(s, t)} \text{ and } E_2.\text{type} = \text{integer} \text{ then t else type_error}) \; \}
\]

- Parameter \( t \) is set with the unification of \( E_1.\text{type} = \text{array}(s, t) \)

\[
E \rightarrow E_1 ^ \wedge \quad \{ \; E.\text{type} := (\text{if } E_1.\text{type} = \text{pointer}(t) \text{ then t else type_error}) \; \}
\]

- Parameter \( t \) is set with the unification of \( E_1.\text{type} = \text{pointer}(t) \)
Type Conversion and Coercion

• **Type conversion** is explicit, for example using type casts

• **Type coercion** is implicitly performed by the compiler to generate code that converts types of values at runtime (typically to *narrow* or *widen* a type)

• Both require a *type system* to check and infer types from (sub)expressions
On Coercion

• Coercion may change the representation of the value or not
  – Integer $\rightarrow$ Real $\quad$ binary representation is changed
    $\{\text{int } x = 5; \text{ double } y = x; \ldots\}$
  – A $\rightarrow$ B subclasses $\quad$ binary representation not changed
    class A extends B{$\ldots$}
    $\{\text{B myBobject = new A(\ldots); }\ldots\}$

• 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
• Popular again in modern scripting languages
Example: Type Coercion and Cast in Java among numerical types

- Coercion (implicit, widening)
  - No loss of information (almost...)
- Cast (explicit, narrowing)
  - Some information can be lost
- Explicit cast is always allowed when coercion is
Handling coercion during translation

Translation of sum without type coercion:
\[ E \rightarrow E_1 + E_2 \quad \{ \quad E\text{.place} := \text{newtemp}(); \]
\[ \text{gen}(E\text{.place} '=>' E_1\text{.place} '+' E_2\text{.place}) \} \]

With type coercion:
\[ E \rightarrow E_1 + E_2 \quad \{ \quad E\text{.type} = \text{max}(E_1\text{.type,} E_2\text{.type}); \]
\[ a_1 = \text{widen}(E_1\text{.addr,} E_1\text{.type,} E\text{.type}); \]
\[ a_2 = \text{widen}(E_2\text{.addr,} E_2\text{.type,} E\text{.type}); \]
\[ E\text{.addr} = \text{new Temp}(); \]
\[ \text{gen}(E\text{.addr} '=>' a_1 '+' a_2); \} \]

where:
- \text{max}(T_1, T_2) returns the least upper bound of \( T_1 \) and \( T_2 \) in the widening hierarchy
- \text{widen(addr, T}_1, T_2) generate the statement that copies the value of type \( T_1 \) in addr to a new temporary, casting it to \( T_2 \)
Pseudocode for widen

Addr widen(Addr a, Type t, Type w) {
    temp = new Temp();
    if (t = w) return a;  // no coercion needed
    elseif (t = integer and w = float) {
        gen(temp '==' '(float)' a);
    }
    elseif (t = integer and w = double) {
        gen(temp '==' '(double)' a);
    }
    elseif ...
    else error;
    return temp; }
}
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
  – subrange

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

35
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 \( i \)th 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
    • 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:**
  ```ada
  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:**
  ```ada
  someday: Date := (jan, 1);
  ...
  put(someday.m+1); put("/"); put(someday.d);
  someday.d := 29; someday.m := feb;
  ```
Example: Haskell tuples

• Declarations:

```haskell
data Month = Jan | Feb | Mar | Apr
| May | Jun | Jul | Aug
| Sep | Oct | Nov | Dec

type Date = (Month, Int)
```

• Set of values:

```haskell
Date = Month × Integer
    = \{Jan, Feb, ..., Dec\} × {..., −1, 0, 1, 2, ...}
```

• Application code:

```haskell
someday = (jan, 1)  // tuple construction
m, d = someday       // component selection
                     // (by pattern matching)
anotherday = (m + 1, d)
```
Mappings

• We write $m : S \rightarrow T$ to state that $m$ is a mapping from set $S$ to set $T$. In other words, $m$ maps every value in $S$ to some value in $T$.

• If $m$ maps value $x$ to value $y$, we write $y = m(x)$. The value $y$ is called the image of $x$ under $m$.

• Some of the mappings in $\{u, v\} \rightarrow \{a, b, c\}$:
  
  $m_1 = \{u \rightarrow a, v \rightarrow c\}$
  $m_2 = \{u \rightarrow c, v \rightarrow c\}$
  $m_3 = \{u \rightarrow c, v \rightarrow b\}$
Arrays (1)

- **Arrays** (found in all imperative and OO PLs) can be understood as mappings.

- If the array’s elements are of type $T$ (*base type*) and its index values are of type $S$, the array’s type is $S \rightarrow T$.

- An array’s **length** is the number of components, $\#S$.

- Basic operations on arrays:
  - **construction** of an array from its components
  - **indexing** – using a *computed* index value to select a component
    - we *can* select the $i$th component
Arrays (2)

- An array of type $S \rightarrow T$ is a finite mapping.
- Here $S$ is nearly always a finite range of consecutive values $\{l, l+1, \ldots, u\}$. This is called the array’s index range.

- In C and Java, the index range must be $\{0, 1, \ldots, n-1\}$.
  In Pascal and Ada, the index range may be any scalar (sub)type other than real/float.

- We can generalise to $n$-dimensional arrays. If an array has index ranges of types $S_1, \ldots, S_n$, the array’s type is $S_1 \times \ldots \times S_n \rightarrow T$. 
When is the index range known?

- A **static array** is an array variable whose index range is fixed by the program code.

- A **dynamic array** is an array variable whose index range is fixed at the time when the array variable is created.
  - In Ada, the definition of an array type must fix the index *type*, but need not fix the index *range*. Only when an array variable is created must its index range be fixed.
  - Arrays as formal parameters of subroutines are often dynamic (e.g. *conformant arrays* in Pascal)

- A **flexible** (or **fully dynamic**) **array** is an array variable whose index range is not fixed at all, but may change whenever a new array value is assigned.
Example: C static arrays

- Array variable declarations:
  
  ```c
  float v1[] = {2.0, 3.0, 5.0, 7.0};
  float v2[10];
  ```

- Function:
  
  ```c
  void print_vector (float v[], int n) {
    // Print the array v[0], ..., v[n-1] in the form "[... ...]."
    int i;
    printf("[%f", v[0]);
    for (i = 1; i < n; i++)
      printf(" %f", v[i]);
    printf("]");
  }
  ```

  ```c
  ...
  print_vector(v1, 4);  print_vector(v2, 10);
  ```
Example: Ada dynamic arrays

- Array type and variable declarations:
  ```ada
  type Vector is
    array (Integer range <>) of Float;
  v1: Vector(1 .. 4) := (1.0, 0.5, 5.0, 3.5);
  v2: Vector(0 .. m) := (0 .. m => 0.0);
  ```

- Procedure:
  ```ada
  procedure print_vector (v: in Vector) is
    -- Print the array v in the form “[... ... ...]”.
  begin
    put('[');  put(v(v'first));
    for i in v'first + 1 .. v'last loop
      put(' ');  put(v(i));
    end loop;
    put(']');
  end;
  ...
  print_vector(v1);  print_vector(v2);
  ```
Example: Java flexible arrays

• Array variable declarations:

  ```java
  float[] v1 = {1.0, 0.5, 5.0, 3.5};  // index range is {0, ..., 3}
  float[] v2 = {0.0, 0.0, 0.0};
  ...
  v1 = v2;  // v1’s index range is now {0, ..., 2}
  ```

• Method:

  ```java
  static void printVector (float[] v) {
    // Print the array v in the form “[... ... ...]”.
    System.out.print("[" + v[0]);
    for (int i = 1; i < v.length; i++)
      System.out.print(" "+ v[i]);
    System.out.print("]");
  }

  ...
  printVector(v1);  printVector(v2);
  ```