Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

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, *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
  - *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.
- **Statically typed** languages: (most) type checking is done during compilation
- **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:

```
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 **mod** 2 = 0" will be Boolean.

- Call:

  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:

```
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

```
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!

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

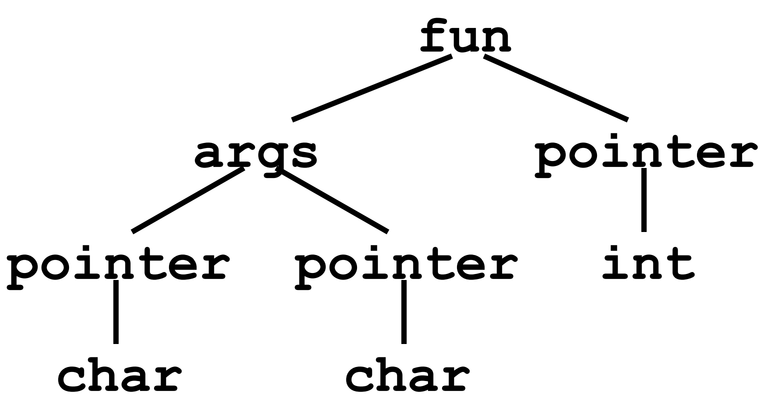
Type ::= **int** | **bool** | … | X | Tname |pointer-to(Type) |
array(*num*, Type) | record(Fields) | class(…) |
Type → Type | Type x Type

  - *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

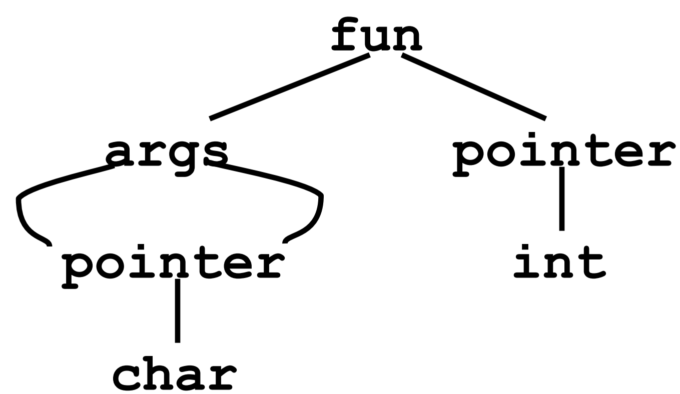
Graph Representations for Type Expressions

- Internal compiler representation, built during parsing
- Example:

```
int *f(char*,char*)
```



Tree forms

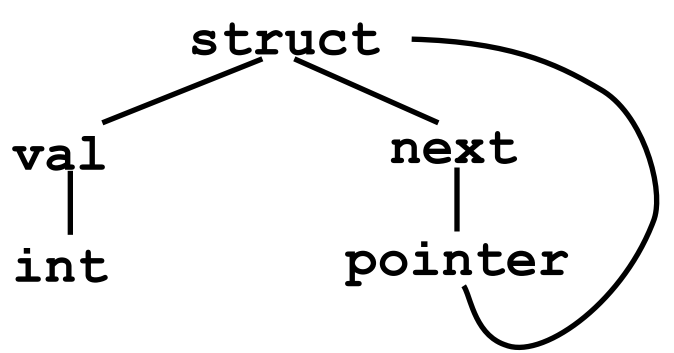


DAGs

Cyclic Graph Representations

Source program

```
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 ***structurally equivalent*** if
    1. They are the same basic types, or
    2. They have the form **TC(T1,…, Tn)** and **TC(S1, …, Sn)**, where **TC** is a type constructor and **Ti** is structurally equivalent to **Si** for all 1 <= i <= n, or
    3. One is a type name that denotes the other.
  - Two types are ***name equivalent*** if they satisfy 1. and 2.

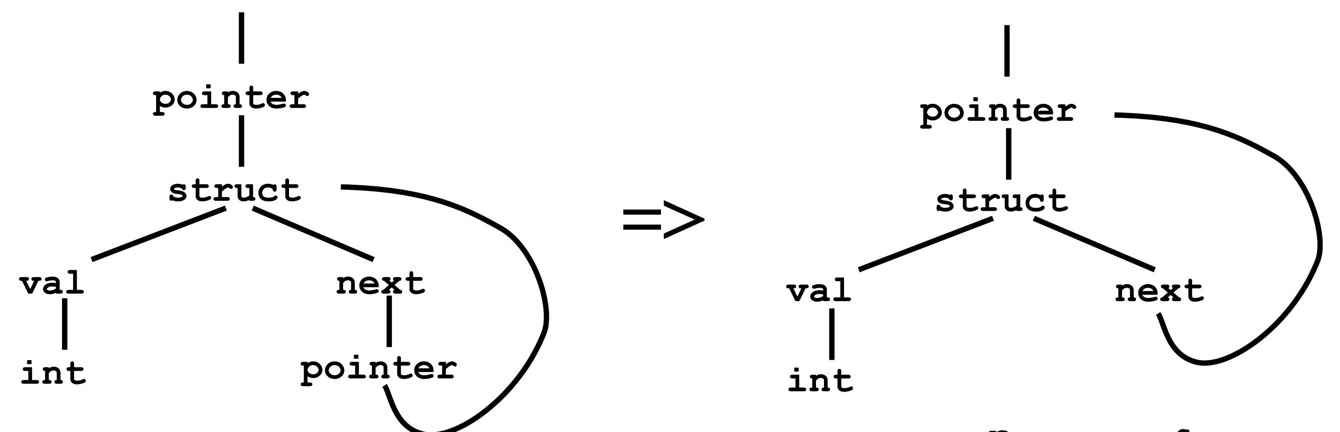
On Structural Equivalence

- **Structural equivalence**: unravel all type constructors obtaining type expressions containing only primitive types, then check if they are equivalent
- Used in C/C++, C#

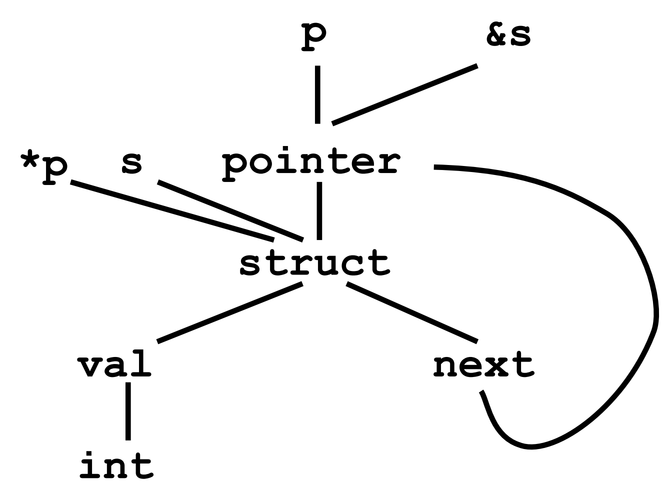
```
-- pseudo Pascal
type Student = record
     name, address : string
     age : integer

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




```
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)

```
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

```
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
| **id :** T
$T \rightarrow$ **boolean**
| **char**
| **integer**
| **array [num] of** T
| **^** T
$S \rightarrow$ **id :=** E
| **if** E **then** S
| **while** E **do** S
| S **;** S

Pointer to *T*

$E \rightarrow$ **true**
| **false**
| **literal**
| **num**
| **id**
| E **and** E
| E **+** E
| E **[** E **]**
| E **^**

Pascal-like pointer dereference operator

Synthesized attributes
T.type : type expression
E.type : type of expression or *type_error*
S.type : *void* if statement is well-typed, *type_error* otherwise

Declarations

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

Parametric types:
type constructor

Checking Statements

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

- Note: the type of **id** is determined by scope's environment: **id**.type = *lookup*(**id**.entry)

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

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

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

Checking Expressions

$E \rightarrow$ **true** { E.type = *boolean* }

$E \rightarrow$ **false** { E.type = *boolean* }

$E \rightarrow$ **literal** { E.type = *char* }

$E \rightarrow$ **num** { E.type = *integer* }

$E \rightarrow$ **id** { E.type = *lookup*(**id**.entry) }

$E \rightarrow E_1$ **+** E_2 { E.type := (**if** E_1.type = *integer* **and** E_2.type = *integer*
then *integer* **else** *type_error*) }

$E \rightarrow E_1$ **and** E_2 { E.type := (**if** E_1.type = *boolean* **and** E_2.type = *boolean*
then *boolean* **else** *type_error*) }

$E \rightarrow E_1$ **[** E_2 **]** { E.type := (**if** E_1.type = *array*(*s*, ***t***) **and** E_2.type = *integer*
then ***t*** **else** *type_error*) }

- Parameter ***t*** is set with the unification of E_1.type = *array*(*s*, ***t***)

$E \rightarrow E_1$ **^** { E.type := (**if** E_1.type = *pointer*(***t***) **then** ***t***
else *type_error*) }

- Parameter ***t*** is set with the unification of E_1.type = *pointer*(***t***)

Type Conversion and Coercion

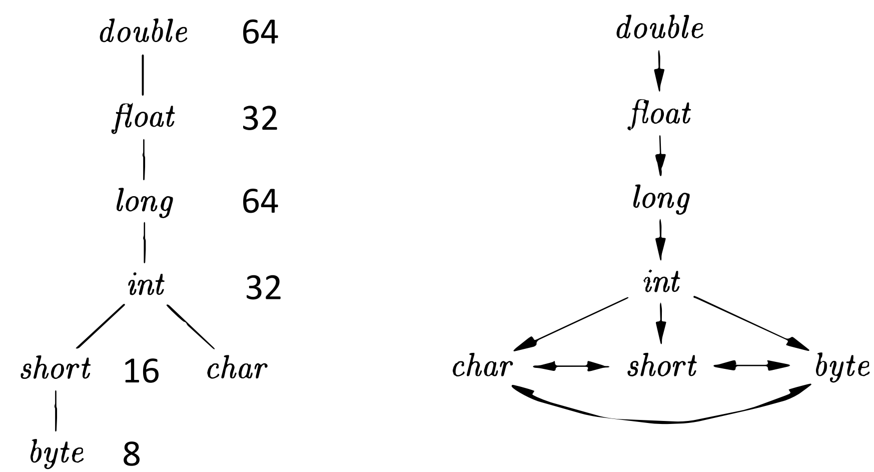
- **Type conversion** is explicit, for example using type casts
- **Type coercion** is implicitly performed by the compiler to generate code that converts types of values at runtime (typically to *narrow* or *widen* a type)
- Both require a *type system* to check and infer types from (sub)expressions

On Coercion

- Coercion may change the representation of the value or not
  - Integer → 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
- 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



(a) Widening conversions (b) Narrowing conversions

Handling coercion during translation

Translation of sum without type coercion:

$E \rightarrow E_1 + E_2$ { E.place := *newtemp*();
gen(E.place ':=' E_1.place '+' E_2.place) }

With type coercion:

$E \rightarrow E_1 + E_2$ { E. type = **max**(E_1.type,E_2.type);
a_1 = **widen**(E_1.addr, E_1.type, E.type) ;
a_2 = **widen**(E_2.addr, E_2.type, E.type);
E.addr = new Temp();
gen(E.addr '=' a_1 '+' a_2); }

where:

- **max(T_1,T_2)** returns the least upper bound of T_1 and T_2 in the widening hierarchy
- **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’ , …}

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

  Integer = {…, –2, –1, 0, +1, +2, …}

  PL- or implementation-defined set of whole numbers

  Float = {…, –1.0, …, 0.0, +1.0, …}

  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

- $S \times T$ denotes the Cartesian product of S and T:

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

- We can generalise to **tuples**:

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

- 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:

```
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:

record construction

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

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

Arrays as mappings

- An array of type $S \rightarrow T$ is a *finite* mapping.
- S is typically a finite range of consecutive values $\{l, l+1, \ldots, u\}$, called the array's **index range**.
- Basic operations on arrays:
  - construction of an array from its components
  - indexing – using a computed index value to select a component

- 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 (eg. *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:

```
float v1[] = {2.0, 3.0, 5.0, 7.0};
float v2[10];
```

index range is {0, …, 3}

index range is {0, …, 9}

- Function:

```
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("]");
}

…
print_vector(v1, 4);  print_vector(v2, 10);
```

A C array doesn't know its own length!

Example: Ada dynamic arrays

- Array type and variable declarations:

```
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:

```
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:

```
float[] v1 = {1.0, 0.5, 5.0, 3.5};
float[] v2 = {0.0, 0.0, 0.0};
…
v1 = v2;
```

index range is {0, …, 3}

index range is {0, …, 2}

v1’s index range is now {0, …, 2}

- Method:

```
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);
```

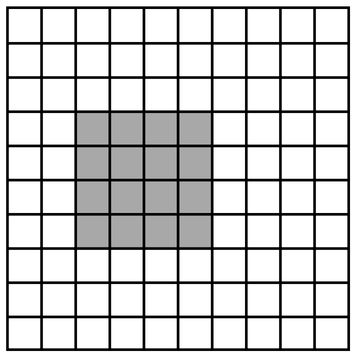
Enhanced for:

```
for  (float f : v)
System.out.print(" " + f)
```

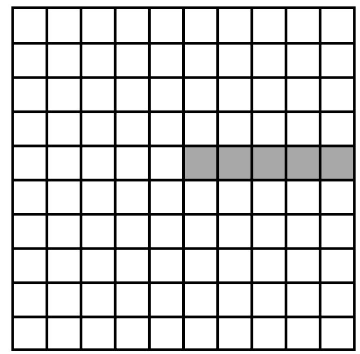
Array-level operations

- Assigment
  - Value or Reference Model
- Comparison for equality or lexicographic ordering (Ada)
- Arithmetic (pointwise) + specific *intrinsic* (built-in) operations in Fortran 90 (and APL)
  - Searching, transposition, reshaping…
- ***Slice*** or ***section***
  - Returns a sub-array by selecting sub-ranges of dimensions

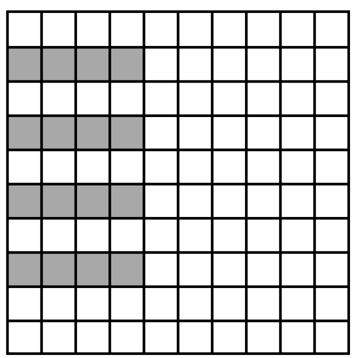
Slicing in Fortran 90



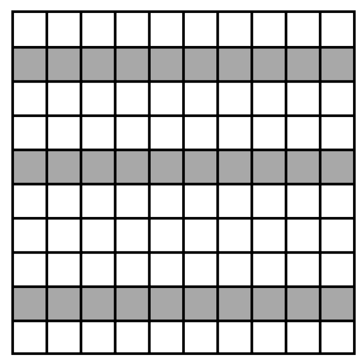
`matrix(3:6, 4:7)`



`matrix(6:, 5)`



`matrix(:4, 2:8:2)`



`matrix(:, (/2, 5, 9/))`

Array allocation

- ***static array, global lifetime* —** If a static array can exist throughout the execution of the program, then the compiler can allocate space for it in *static global memory*
- ***static array, local lifetime* —** If a static array should not exist throughout the execution of the program, then space can be allocated *in the subroutine's stack frame* at run time.
- ***dynamic array, local lifetime* —** If the index range is known at runtime, the array can still be allocated *in the stack*, but in a variable size area
- ***fully dynamic* —** If the index range can be modified at runtime it has to be allocated *in the heap*

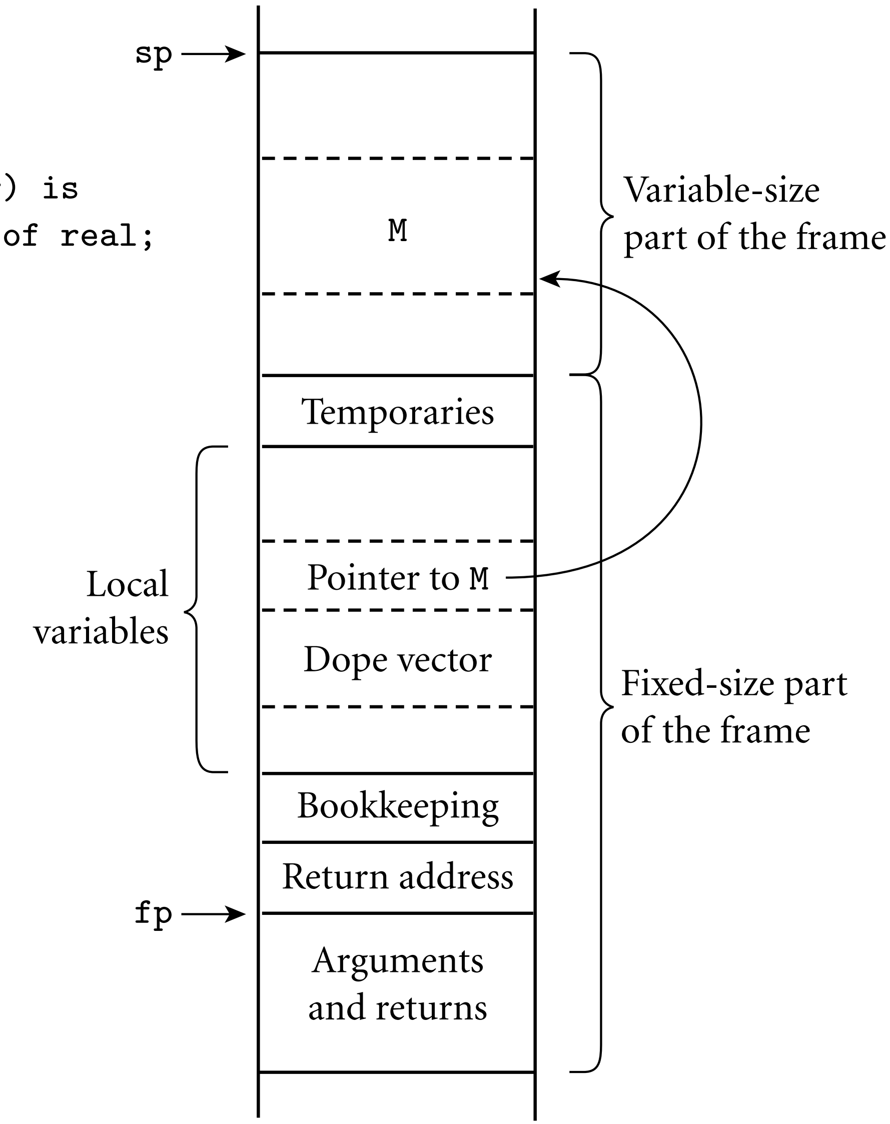
Dope vector: run-time data structure that keeps information about lower (and upper) limits of arrays ranges

  – Needed for checking bounds and computing addresses of elements

Allocation of dynamic arrays on stack

```
-- Ada:
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



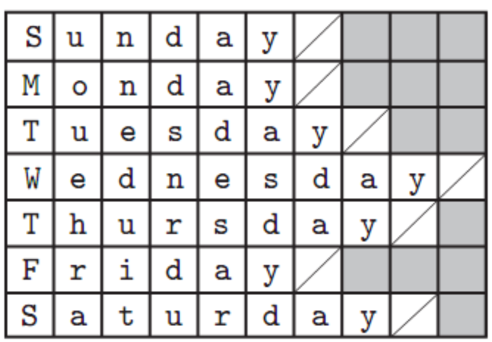
Arrays: memory layout

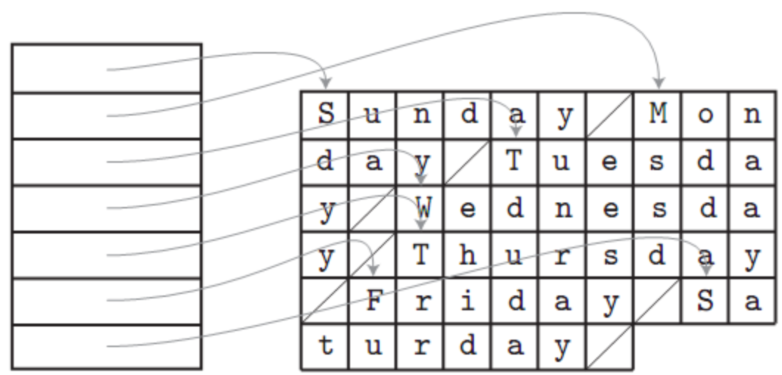
- Contiguous elements
  - column major - only in Fortran
  - row major
    - used by everybody else
- Row pointers
  - an option in C, the rule in Java
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g. an array of strings
  - requires extra space for the pointers

Arrays’ memory layout in C

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == ’s’;  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == ’s’;  /* in Tuesday */
```





- Address computation varies a lot
- With contiguous allocation part of the computation can be done statically

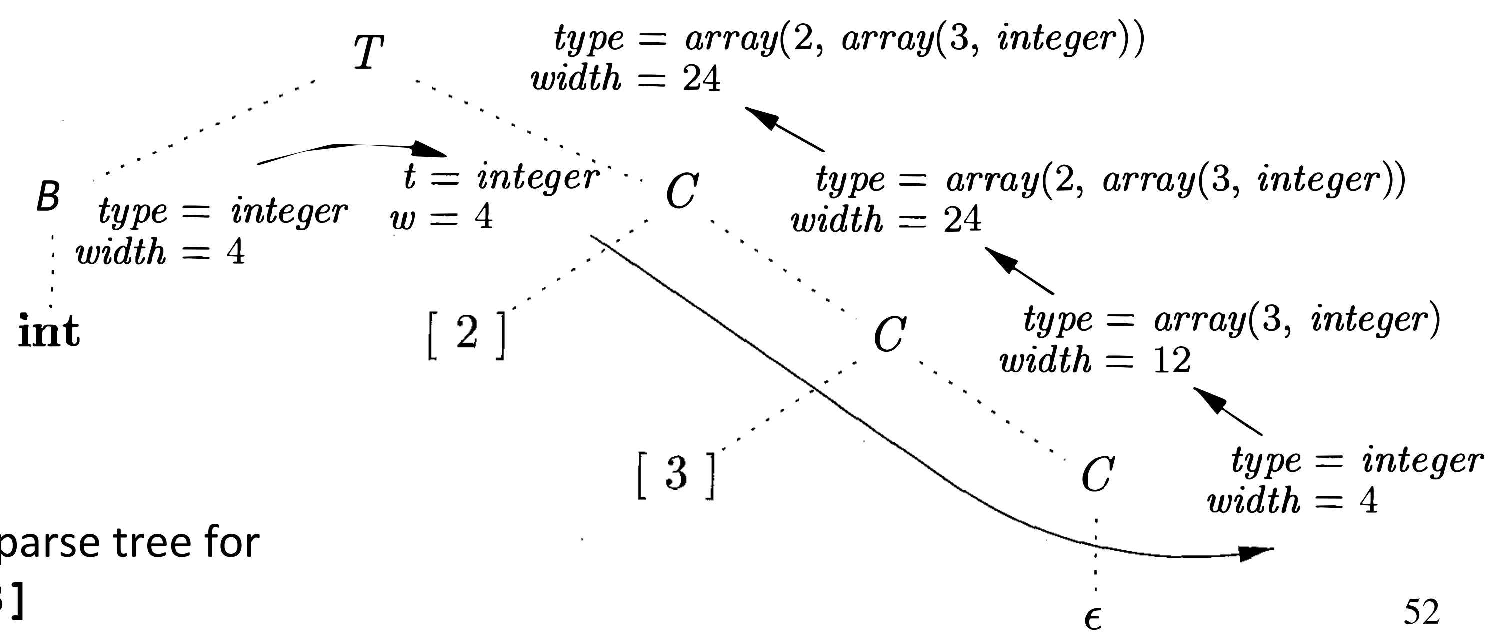
Compiling array declarations and addressing

- Translation scheme for associating with an array declaration a *type expression* and the *width* of its instances
- Computing the address of an array element: one- and multi-dimensional cases
- Generating three address code for addressing array elements

Declaration of Multidimensional Arrays: Syntax Directed Translation Scheme for type/width

Example: `int[2][3]`

$T \rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
	C	$\{\ T.type = C.type;\ T.width = C.width\ \}$
$B \rightarrow$	**int**	$\{\ B.type = \text{'integer'};\ B.width = 4;\ \}$
$B \rightarrow$	**float**	$\{\ B.type = \text{'float'};\ B.width = 8;\ \}$
$C \rightarrow$	ε	$\{\ C.type = t;\ C.width = w;\ \}$
$C \rightarrow$	[**num**] C_1	$\{\ C.type = array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value * C_1.width;\ \}$



Annotated parse tree for
`int[2][3]`

Addressing Array Elements: One-Dimensional Arrays

- Assuming that elements are stored in adjacent cells:

```
A : array [10..20] of integer;
```

low *high* *Type's size*

… `:= A[i]` $= base_{\mathtt{A}} + (i - low) * w$

- If *base, low* and *w* are known at compile time:

$$= i * w + c \quad \textbf{\textit{where}}\ c = base_{\mathtt{A}} - low * w$$

Example with $low = 10$; $w = 4$

…

t1 := c **//c =** $base_{\mathtt{A}}$ - 10 * 4, can be stored in the symbol table

t2 := i * 4

t3 := t1[t2]

… **:= t3**

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;`

$$low_1 = 1, low_2 = 1,$$
$$n_1 = high_1 - low_1 + 1 = 2, \quad n_2 = 3,$$
$$w = 4 \ \text{(element type size)}$$

base$_{\mathtt{A}}$

`A[1][1]`
`A[1][2]`
`A[1][3]`
`A[2][1]`
`A[2][2]`
`A[2][3]`

(as in C) Row-major

base$_{\mathtt{A}}$

`A[1][1]`
`A[2][1]`
`A[1][2]`
`A[2][2]`
`A[1][3]`
`A[2][3]`

Column-major (as in Fortran)

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;` (Row-major)

… **`:= A[i][j]`** $= base_{\mathtt{A}} + ((i - low_1) * n_2 + j - low_2) * w$

$= ((i * n_2) + j) * w + c$

where $c = base_{\mathtt{A}} - ((low_1 * n_2) + low_2) * w$

Example with $low_1 = 1$; $low_2 = 1$; $n_2 = 3$; $w = 4$

```
t1  := i * 3
t1  := t1 + j
t2  := c            // c = base_A - (1 * 3 + 1) * 4
t3  := t1 * 4
t4  := t2[t3]       // base t2, offset t3
…   := t4
```

Addressing Array Elements: Grammar

Grammar:

$S \rightarrow$ **id** = E ;
   | L = E ;
$E \rightarrow E$ + E
   | **id**
   | L
$L \rightarrow$ **id** [E]
   | L [E]

Synthesized attributes:

E.addr	name of temp holding value of *E*
L.addr	temporary to compute offset
L.array	pointer to symbol table entry for the array name
L.array.base	base address
L.array.type	type of the array, eg. *array(2, array(3,int))*
L.array.type.elem	type of array elements, eg. *array(3,int)*
L.type	type of the subarray generated by *L*
L.type.width	memory allocated for data of type *L.type*

- Nonterminal *L* generates an array name followed by a sequence of indexes, like
  `a[i][j][k]`
- *L* can appear both as left- and right-value

Addressing array elements: generating three address statements

$S \rightarrow$ **id** = E ; { *gen(top.get(***id***.lexeme)* '=' *E.addr);* } // no array

| L = E ; { *gen(L.array.base* '[' *L.addr* ']' '=' *E.addr);* } // address = base + offset

$E \rightarrow E_1 + E_2$ { *E.addr* = **new** *Temp()*;
gen(E.addr '=' E_1*.addr* '+' E_2*.addr);* } // similarly for *, -, …

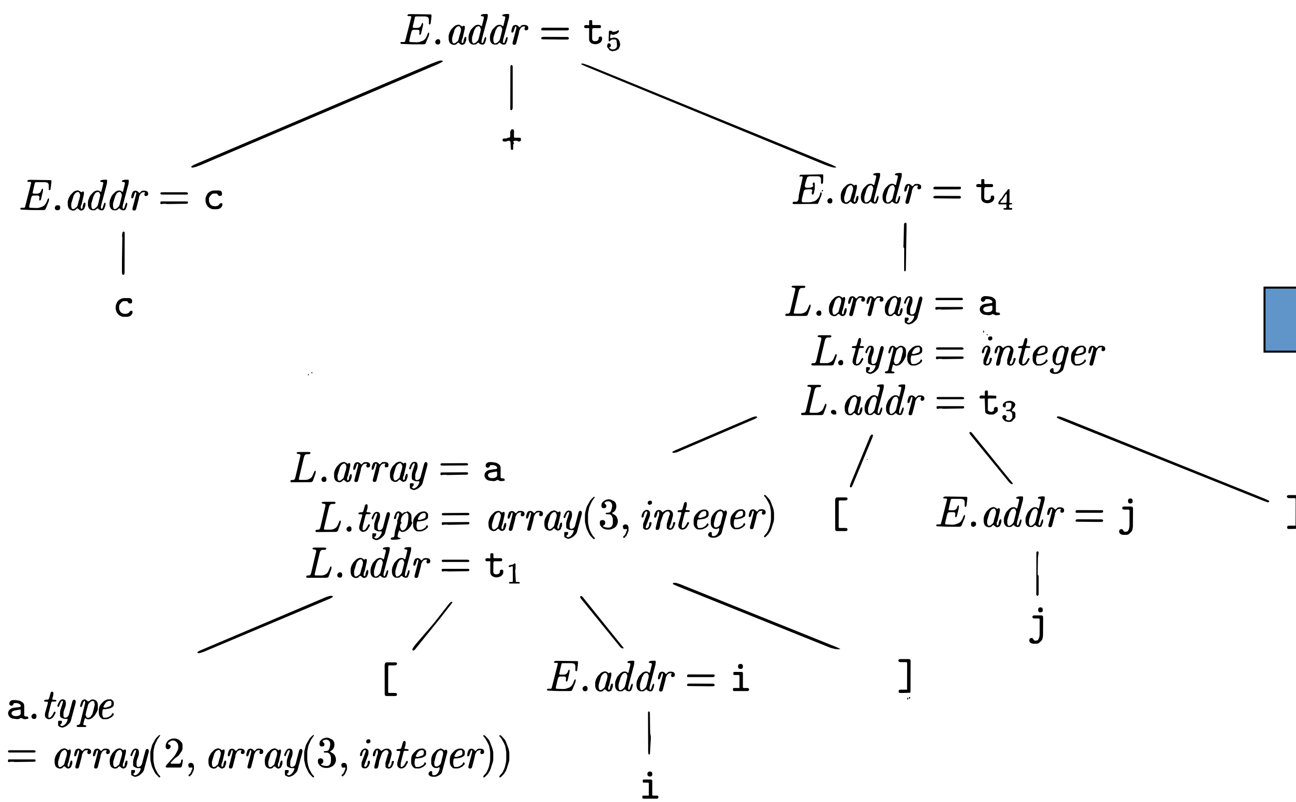
| **id** { *E.addr = top.get(***id***.lexeme);* }

| L { *E.addr* = **new** *Temp();*
gen(E.addr '=' *L.array.base* '[' *L.addr* ']'*);* } // address = base + offset

$L \rightarrow$ **id** [E] { *L.array = top.get(***id***.lexeme);*
L.type = L.array.type.elem;
L.addr = **new** *Temp();*
gen(L.addr '=' *E.addr* '*' *L.type.width);* } // computes the offset

| L_1 [E] { *L.array=*L_1*.array;*
L.type = L_1*.type.elem;*
t = **new** *Temp();*
L.addr= **new** *Temp();*
gen(t '=' *E.addr* '*' *L.type.width);*
gen(L.addr '=' L_1*.addr* '+' *t);* }

Example - generating intermediate code for access to array: `c + a[i][j]`




```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Strings

- A **string** is a sequence of 0 or more characters.
- Usually ad-hoc syntax is supported
- Some PLs (ML, Python) treat strings as *primitive*.
- Haskell treats strings as *lists* of characters. Strings are thus equipped with general list operations (length, head selection, tail selection, concatenation, …).
- Ada treats strings as *arrays* of characters. Strings are thus equipped with general array operations (length, indexing, slicing, concatenation, …).
- Also in C strings are arrays of characters, but handled differently from other arrays
- Java treats strings as *objects*, of class `String`.

Disjoint Unions

- In a **disjoint union**, a value is chosen from one of several different types.
- Let ***S* + *T*** stand for a set of disjoint-union values, each of which consists of a **tag** together with a **variant** chosen from either type *S* or type *T*. The tag indicates the type of the variant:

  $$S + T \;=\; \{\, left\ x \mid x \in S \,\} \cup \{\, right\ y \mid y \in T \,\}$$

  - *left x* is a value with tag *left* and variant *x* chosen from *S*
  - *right x* is a value with tag *right* and variant *y* chosen from *T*.
- We write ***left S* + *right T*** (instead of *S* + *T*) when we want to make the tags explicit.

Disjoint Unions

- Basic operations on disjoint-union values in $S + T$:
  - **construction** of a disjoint-union value from its tag and variant
  - **tag test**, to see whether the variant is from S or T
  - **projection**, to recover the variant in S or in T
- **Algebraic data types** (Haskell), **discriminated records** (Ada), **unions** (C) and **objects** (Java) can be understood as disjoint unions.
- We can generalise to multiple variants: $S_1 + S_2 + \ldots + S_n$.

Variant records (unions)

- Origin: Fortran I *equivalence statement:* variables should share the same memory location
- C's *union* types
- Motivations:
  - Saving space
  - Need of different access to the same memory locations for system programming
  - Alternative configurations of a data type

Fortran I -- equivalence statement

```
integer i
real r
logical b
equivalence (i, r, b)
```

C -- union

```
union {
    int i;
    double d;
    _Bool b;
};
```

Variant records (unions) (2)

- In Ada, Pascal, unions are *discriminated* by a tag, called *discriminant*
- Integrated with records in Pascal/Ada, not in C

ADA – discriminated variant

tag

```
type Form is
     (pointy, circular, rectangular);
type Figure (f: Form := pointy) is record
       x, y: Float;
       case f is
          when pointy      => null;
          when circular    => r: Float;
          when rectangular => w, h: Float;
       end case;
     end record;
```

Using discriminated records in Ada

- Application code:

```
box: Figure :=
    (rectangular, 1.5, 2.0, 3.0, 4.0);
function area (fig: Figure) return Float
is
begin
  case fig.f is
    when pointy =>
      return 0.0;
    when circular =>
      return 3.1416 * fig.r**2;
    when rectangular =>
      return fig.w * fig.h;
  end case;
end;
```

discriminated-record construction

tag test

projection

(Lack of) Safety in variant records

- Only Ada has strict rules for assignment: tag and variant have to be changed *together*
- For *nondiscriminated unions* (Fortran, C) no runtime check: responsibility of the programmer
- In Pascal the tag field can be modified independently of the variant. Even worse: the tag field is optional.
- Unions not included recent OO laguages: replaced by *algebraic data types* or *classes + inheritance*

Haskell/ML algebraic data types

- Type declaration:

```
data Number = Exact Int | Inexact Float
```

Each Number value consists of a tag (**constructor**), together with either an Integer variant (if the tag is *Exact*) or a Float variant (if the tag is *Inexact*).

- Application code:

```
pi = Inexact 3.1416
rounded :: Number -> Integer
rounded num =
   case num of
     Exact i   -> i
     Inexact r -> round r
```

projection (by pattern matching)

Active patterns in F#

- With algebraic data types, the type definition determines uniquely the patterns
- *Active patterns*, can be used to “wrap” a data type, algebraic or not, providing a different perspective for use of pattern matching
- Essentially, active patterns define ad-hoc, unnamed union types

Active pattern definition

```
let (|Even|Odd|) n =
   if n % 2 = 0 then
      Even
   else
      Odd
```

Roughly equivalent to

```
type numKind =
   | Even
   | Odd

let get_choice n =
   if n % 2 = 0 then
      Even
   else
      Odd
```

Using active patterns

```
let testNum n =
   match n with
   | Even -> printfn "%i is even" n
   | Odd -> printfn "%i is odd" n;;§
```

Active Patterns defining Constructors with Parameters

```
/* Active pattern for Sequences */
let (|SeqNode|SeqEmpty|) s =
   if Seq.isEmpty s then SeqEmpty
   else SeqNode ((Seq.head s), Seq.skip 1 s)

/* SeqNode is a constructor with two parameters */

let perfectSquares = seq { for a in 1 .. 10 -> a * a }

let rec printSeq = function
   | SeqEmpty -> printfn "Done."
   | SeqNode(hd, tl) ->
       printf "%A " hd
       printSeq tl;;

> printSeq perfectSquares;;
1 4 9 16 25 36 49 64 81 100 Done.
```

Java objects as unions

- Type declarations:

```
class Point {
  private float x, y;
  …  // methods
}
class Circle extends Point {
  private float r;
  …  // methods
}
class Rectangle extends Point {
  private float w, h;
  …  // methods
}
```

inherits x and y from Point

inherits x and y from Point

Java objects as unions (2)

- Methods:

```
class Point {
  …
  public float area()
  { return 0.0; }
}
class Circle extends Point {
  …
  public float area()
  { return 3.1416 * r * r; }
}
class Rectangle extends Point {
  …
  public float area()
  { return w * h; }
}
```

overrides Point’s area() method

overrides Point’s area() method

Java objects as unions (3)

- Application code:

```
Rectangle box =
    new Rectangle(1.5, 2.0, 3.0,4.0);

float a1 = box.area();
Point it = …;
float a2 = it.area();
```

it can refer to a Point, Circle, or Rectangle object

calls the appropriate area() method

Assignments and Expressions

- Fundamental difference between imperative and functional languages
- **Imperative languages**: “computing by means of side effects”
  - Computation is an ordered series of changes to values of variables in memory (state) and statement ordering is influenced by run-time testing values of variables
- Expressions in (pure) **functional language** are *referentially transparent*:
  - All values used and produced depend on the local referencing environment of the expression
  - A function is *idempotent* in a (pure) functional language: it always returns the same value given the same arguments because of the absence of side-effects

L-Values vs. R-Values and Value Model vs. Reference Model

- Consider the assignment of the form: ***a* := *b***
  - ***a*** is an *l-value*, i.e. an expression that should denote a location (an array element **a[2]**, a variable **foo**, a dereferenced pointer ***p** or a more complex expression **(f(a)+3)->b[c]**)
  - ***b*** is an *r-value*: any syntactically valid expression with type compatible to that of ***a***
- Languages that adopt the ***value model*** of variables copy the value of ***b*** into the location of ***a*** (e.g. Ada, Pascal, C, …)
- Languages that adopt the ***reference model*** of variables copy references, resulting in shared data values
  - Clu, Lisp/Scheme, ML, Haskell, Smalltalk adopt the reference model
  - Most imperative programming languages use the value model
  - **Java** is a mix: it uses the value model for built-in types and the reference model for class instances

Assignment in Value Model vs. Reference Model

b := 2;
c := b;
a := b + c

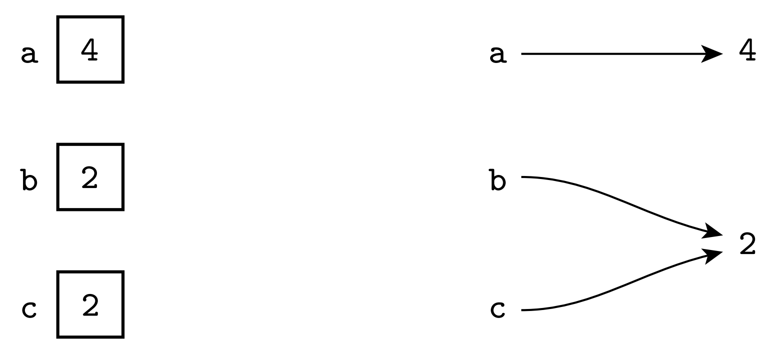


Figure 6.2 The value (left) and reference (right) models of variables. Under the reference model, it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal.

Example: Declaration of variables and assignment

Syntax

Decl ::= **var** Ide = Exp |

Exp ::= …

Com ::= Exp := Exp | …

Semantics: assignment with side effects

C {e1 := e2} r s = *update(x* ***as*** Loc, *v* ***as*** Sval*)* s2

where (x, s1) = *E*{e1} r s

and (v,s2) = *E*{e2} r s1

Evaluates first e1 *then* e2: *store changes are propagated*

Semantics: declaration

D{**var** *x* = *e*} r s = (r[l/x], s[n/l])

where l = *newloc*(s)

and *n* = *E*{*e*} *r s*

Allocates a new location bound to x and containing n

Semantic interpretation functions

D: Decl → Env → Store → (Env x Store)

C: Cmd → Env → Store → Store

E: Exp → Env → Store → Eval *no side eff*

E: Exp → Env → Store → (Eval x Store)

Env = Ide → Dval

Store = Loc → Sval

Dval = … + Loc + …

Eval = … + Loc + Sval + …

Semantics: assignment

C {e1 := e2} r s = *update(x, v)* s

where x = *E*{e1} r s ***as*** Loc

and v = *E*{e2} r s ***as*** Sval

No side-effects, no coercion

Denotational semantics of value model and reference model

- A PL with **value model** has the usual Env and Store semantic domains
  - Env = Ide → Dval (Dval = … + Loc + …)
  - Store = Loc → Sval
  - *Semantic interpretation function* *E*: Exp → Env → Store → (Eval x Store)
- "r-values" are expressions that evaluate to elements of domain Sval (storable values)
- "l-values" are expressions **e** that evaluate to locations: (E{**e**} r s **as Loc**)
- In a PL with **reference model**, conceptually there is no Store, but only
  - Env = Ide → Dval thus *E*: Exp → Env → Eval
- The main binding operator is **let**

  Exp = … | let Ide = Exp in Exp

with semantics

E{ let x = e in e1 } r = *E*{ e1 } r[*E*{e}r / x]

- Note: let x = e in e1 can be seen as syntactic sugar for (λ x. e1) e

References and pointers

- Most **implementations** of PLs have as target architecture a Von Neumann one, where memory is made of cells with addresses
- Thus implementations use the ***value model*** of the target architecture
- Assumption: every data structure is stored in memory cells
- We "define":
  - A **reference to X** is the address of the (base) cell where X is stored
  - A **pointer to X** is a location containing the address of X
- Value-model-based implementation can mimic the ***reference model*** using *pointers* and standard assignment
  - Each variable is associated with a location
  - To let variable **x** refer to data **X**, the address of (reference to) **X** is written in the location of **x**, which becomes a pointer.
  - Can be modeled by requiring that **Loc** is contained in **Sval**
  - Expressions of "reference types" must return a location

Denotational Semantics of Reference Memory Model on Value Memory Model

Semantic interpretation functions

D: Decl → Env → Store → (Env x Store)

C: Cmd → Env → Store → Store

***E*: Exp → Env → Store → (Eval x Store)**

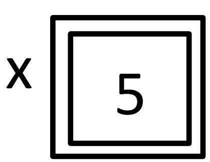
Env = Ide → Dval

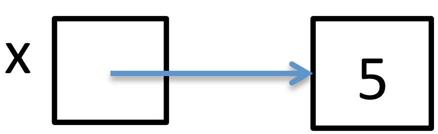
Store = Loc → Sval

Dval = … + Loc + …

Eval = … + Loc + Sval + …

Sval = … + Loc + …





Semantics: declaration

D{**var** *x* $=_{ref}$ *e*} r s = (r[l/x], s1[n/l])

where l = *newloc*(s)

and (n,s1) = *E*{*e*} r s

and (n ***as Loc***)

Allocates a new location bound to x and referring to n

Special Cases of Assignments

- Assignment by *variable initialization*
  - Use of *uninitialized variable* is source of many problems, sometimes compilers are able to detect this but with programmer involvement e.g. *definite assignment* requirement in Java
  - Implicit initialization, e.g. 0 or NaN (not a number) is assigned by default when variable is declared
- Combinations of *assignment operators* (**`+=, -=, *=, ++, --`**…)
  - In C/C++ **`a+=b`** is equivalent to **`a=a+b`** (but **`a[i++]+=b`** is different from **`a[i++]=a[i++]+b`**, !)
  - Compiler produces better code, because the address of a variable is only calculated once
- *Multiway assignments* in Clu, ML, and Perl
  - **`a,b := c,d //`** assigns **`c`** to **`a`** and **`d`** to **`b`** simultaneously,
    - e.g. **`a,b := b,a`** swaps **`a`** with **`b`**
  - **`a,b := f(c) // f`** returns a pair of values

Assignment of composite values

- What happens when a composite value is assigned to a variable of the same type?
- **Value model**: all components of the composite value are copied into the corresponding components of the composite variable.
- **Reference model**: the composite variable is made to contain a reference to the composite value.
- **Note:** this makes no difference for basic or immutable types.

- C and Ada adopt value model
- Java adopts value model for primitive values, reference model for objects.
- Functional languages usually adopt the reference model

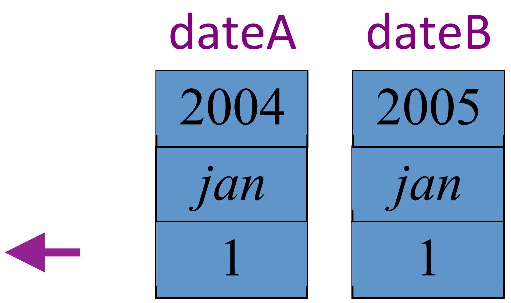
Example: Ada value model (1)

- Declarations:

```
type Date is
     record
        y: Year_Number;
        m: Month;
        d: Day_Number;
     end record;
dateA: Date := (2004, jan, 1);
dateB: Date;
```

- Effect of copy semantics:

```
dateB := dateA;
dateB.y := 2005;
```



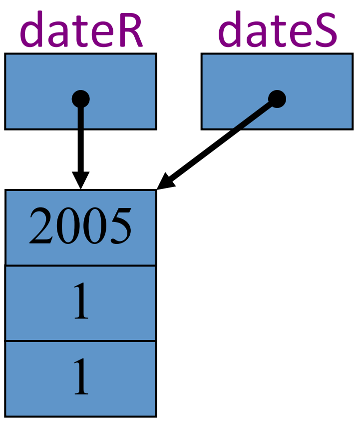
Example: Java reference model (1)

- Declarations:

```
class Date {
  int y, m, d;
  public Date (int y, int m, int d)
{ … }
}
Date dateR = new Date(2004, 1, 1);
Date dateS = new Date(2004, 12, 25);
```

- Effect of reference semantics:

```
dateS = dateR;
dateR.y = 2005;
```



Ada reference model with pointers (2)

- We can achieve the *effect* of reference model in Ada by using explicit ***pointers***:

```
type Date_Pointer is access Date;
Date_Pointer dateP = new Date;
Date_Pointer dateQ = new Date;
…
dateP.all := dateA;
dateQ := dateP;
```

Java value model with cloning (2)

- We can achieve the *effect* of copy semantics in Java by cloning:

```
Date dateR = new Date(2004, 4, 1);
dateT = dateR.clone();
```

Pointers

- Thus in a language adopting the *value model*, the *reference model* can be simulated with the use of pointers.
- A **pointer** (value) is a reference to a particular variable.
- A pointer's **referent** is the variable to which it refers.
- A **null pointer** is a special pointer value that has no referent.
- A pointer is essentially the address of its referent in the store, but it also has a *type*. The type of a pointer allows us to infer the type of its referent.
- Pointers mainly serve two purposes:
  - efficient (sometimes intuitive) access to elaborated objects (as in C)
  - dynamic creation of linked data structures, in conjunction with a heap storage manager

Dangling pointers

- A **dangling pointer** is a pointer to a variable that has been destroyed.
- Dangling pointers arise from the following situations:
  - where a pointer to a heap variable still exists after the heap variable is destroyed by a deallocator
  - where a pointer to a local variable still exists at exit from the block in which the local variable was declared.
- A deallocator immediately destroys a heap variable. All existing pointers to that heap variable become dangling pointers.
- Thus deallocators are inherently unsafe.

Dangling pointers in languages

- C is highly unsafe:
  - After a heap variable is destroyed, pointers to it might still exist.
  - At exit from a block, pointers to its local variables might still exist (e.g., stored in global variables).
- Ada and Pascal are safer:
  - After a heap variable is destroyed, pointers to it might still exist.
  - But pointers to local variables may not be stored in global variables.
- Java is very safe:
  - It has no deallocator.
  - Pointers to local variables cannot be obtained.
- Functional languages are even safer:
  - they don't have pointers

Example: C dangling pointers

- Consider this C code:

```
struct Date {int y, m, d;};
struct Date *dateP, *dateQ;
dateP = (struct Date*)malloc(sizeof (struct Date));
dateP->y = 2004; dateP->m = 1; dateP->d = 1;
dateQ = dateP;
free(dateQ);

printf("%d", dateP->y);
dateP->y = 2005;
```

allocates a new heap variable

makes dateQ point to the same heap variable as dateP

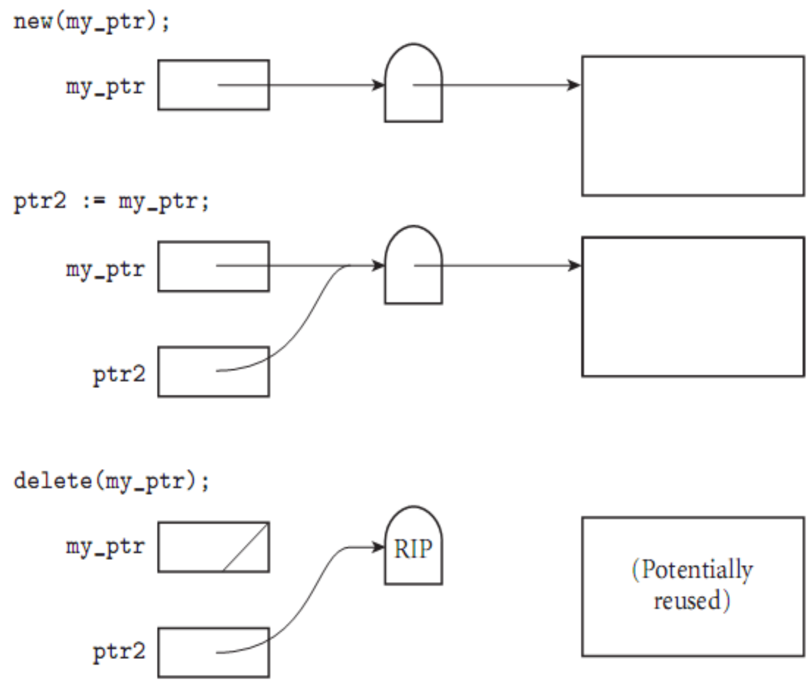
deallocates that heap variable (dateP and dateQ are now dangling pointers)

can fail

can fail

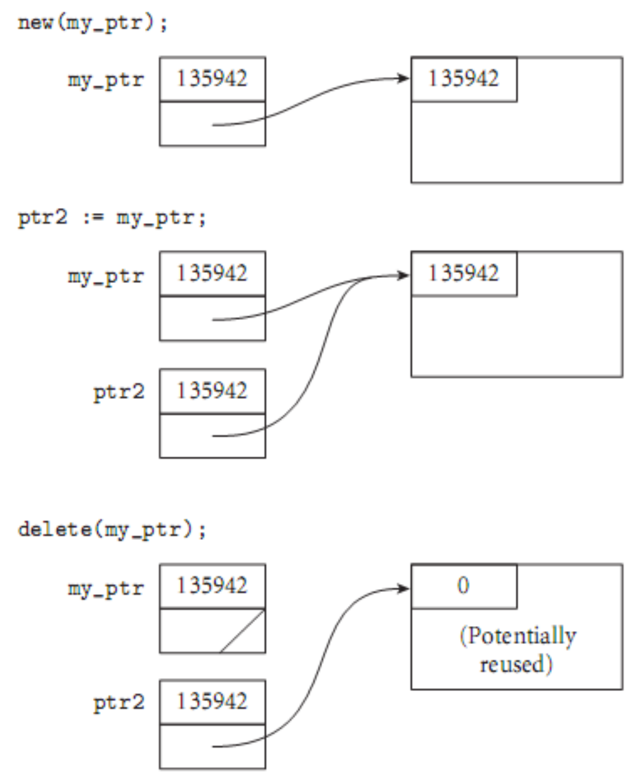
Techniques to avoid dangling pointers

- Tombstones
  - A pointer variable refers to a *tombstone* that in turn refers to an object
  - If the object is destroyed, the tombstone is marked as "expired"



Locks and Keys

- Heap objects are associated with an integer (lock) initialized when created.
- A valid pointer contains a key that matches the lock on the object in the heap.
- Every access checks that they match
- A dangling reference is unlikely to match.



Pointers and arrays in C

- In C, an array variable is a pointer to its first element

```
int *a == int a[]
int **a == int *a[]
```

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension, otherwise it allocates a pointer

`int **a, int *a[]` pointer to pointer to `int`

`int *a[n],` n-element array of row pointers

`int a[n][m],` 2-d array

- Pointer arithmetics: operations on pointers are scaled by the base type size. All these expressions denote the third element of **a**:

`a[2]` **`(a+2)[0]`** **`(a+1)[1]`** **`2[a]`** **`0[a+2]`**

C pointers and recursive types

- C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

```
int *a[n], n-element array of pointers to integer
int (*a)[n], pointer to n-element array of
  integers
```

- Compiler has to be able to tell the size of the things to which you point
  - So the following aren't valid:

```
int a[][]        bad
int (*a)[]       bad
```

Recursive types: Lists

- A **recursive type** is one defined in terms of itself, like lists and trees
- A **list** is a sequence of 0 or more component values.
- The **length** of a list is its number of components. The **empty list** has no components.
- A non-empty list consists of a **head** (its first component) and a **tail** (all but its first component).
- Typical constructor: **cons**: A x A-list -> A-list
- A list is **homogeneous** if all its components are of the same type. Otherwise it is **heterogeneous**.

List operations

- Typical list operations:
  - length
  - emptiness test
  - head selection
  - tail selection
  - concatenation
  - list comprehension

Example: Ada lists

- Type declarations for integer-lists:

```
type IntNode;
type IntList is access IntNode;
type IntNode is record
     head: Integer;
     tail: IntList;
   end record;
```

mutually recursive

- An IntList construction:

```
new IntNode'(2,
    new IntNode'(3,
        new IntNode'(5,
            new IntNode'(7, null)))
```

Example: Java lists

- Class declarations for generic lists:

```
class List<E> {
  public E head;
  public List<E> tail;                recursive
  public List<E> (E el, List<E> t) {
     head = h;  tail = t;
  }
}
```

- A list construction:

```
List<Integer> list =
    new List<Integer>(2,
        new List<Integer>(3,
            new List<integer>(5, null))));
```

Example: Haskell lists

- Haskell has built-in list types:
  - [1, 2, 3]   integer list containing 1, 2, 3
  - [Int] : type of lists of integers. Similarly [Char], [[Int]], [(Int,Char)]
  - 2:[4, 5] == [2, 4, 5]   **cons** is “:”
  - **head** [1, 2, 3] = 1   **tail** [1, 2, 3] = [2, 3]
  - Strings are lists of characters: "foo" == ['f','o','o'] : [Char]
  - **range** [1..10] == [1,2,3,4,5,6,7,8,9,10]
  - **range with step** [3,6..20] == [3,6,9,12,15,18]
  - **range with step** [7,6..1] == [7,6,5,4,3,2,1]
  - **infinite list** [1..] == [1, 2, 3, …]
  - **List comprehension**   [x*y | x <- [2,5,10], y <- [8,10,11]]
    == [16,20,22,40,50,55,80,100,110]