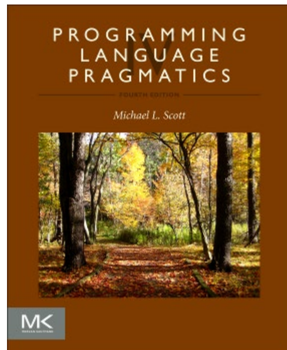
Composite Types

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 8

Prof. Jonathan Aldrich

Copyright © 2016 Elsevier

Records

- A record has multiple named fields
  - Fields may have different types
  - Order usually doesn't matter to semantics
  - Layout chosen by compiler
    - or fixed by programmer in C – helps match hardware expectations
- Operations
  - Create: specify initial value for each field r = { x:5, y:10 }
  - Dereference: read a field r.x; *// evaluates to 5*
  - Assign: update a field r.x := 7
    - We'll model assignment separately later, using references
    - Keeps assignment (and state) orthogonal
- Typing
  - Simple, orthogonal approach: a type for each field

ELSEVIER

Records

- Syntax
  - Note shorthand for values – *v* is a subset of *e*
  - Notation: overbar indicates a list

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \{\, \overline{f = e} \,\} \mid e.f \\
v & ::= & n \mid x : \tau \Rightarrow e \mid \{\, \overline{f = v} \,\} \\
\tau & ::= & \ldots \mid \{\, \overline{f : \tau} \,\}
\end{array}
$$

- Field initialization and dereference

$$
\frac{e_i \rightarrow e_i'}{\{\, \overline{f_{1..i-1} = v_{1..i-1}}, f_i = e_i, \overline{f_{i+1..n} = e_{i+1..n}} \,\} \rightarrow \{\, \overline{f_{1..i-1} = v_{1..i-1}}, f_i = e_i', \overline{f_{i+1..n} = e_{i+1..n}} \,\}} \ \textit{congruence-record}
$$

$$
\frac{}{\{\, \overline{f_{1..i-1} = v_{1..i-1}}, f_i = v_i, \overline{f_{i+1..n} = v_{i+1..n}} \,\}.f_i \rightarrow v_i} \ \textit{step-field}
$$

ELSEVIER

Records

- Typing

$$\frac{\Gamma \vdash \overline{e : \tau}}{\Gamma \vdash \{\ \overline{f = e}\ \} : \{\ \overline{f : \tau}\ \}}\ \textit{T-record}$$

$$\frac{\Gamma \vdash e : \{\ \overline{f : \tau}\ \}}{\Gamma \vdash e.f_i : \tau_i}\ \textit{T-field}$$

- Subtyping
  - Depth subtyping example

{ x:int, y:int } ≤ { x:real, y:real }

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\ \overline{f : \tau}\ \} \leq \{\ \overline{f : \tau'}\ \}}\ \textit{S-depth}$$

  - Width subtyping example

$$\frac{}{\{\ \overline{f : \tau}, \overline{g : \tau'}\ \} \leq \{\ \overline{f : \tau}\ \}}\ \textit{S-width}$$

{ x:int, y:int, z:int } ≤ { x:int, y:int }

Records (Structures)

- Memory layout and its impact (structures)

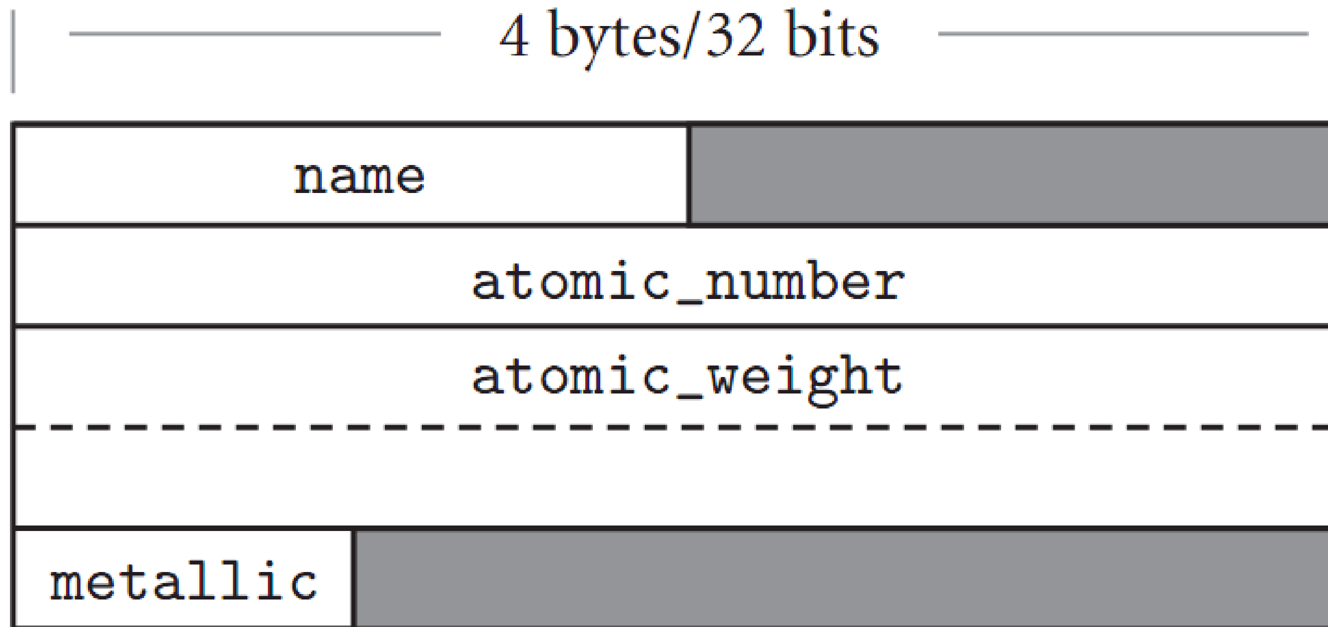


Figure 8.1 Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded “holes.”

ELSEVIER

Records (Structures)

- Memory layout and its impact (structures)

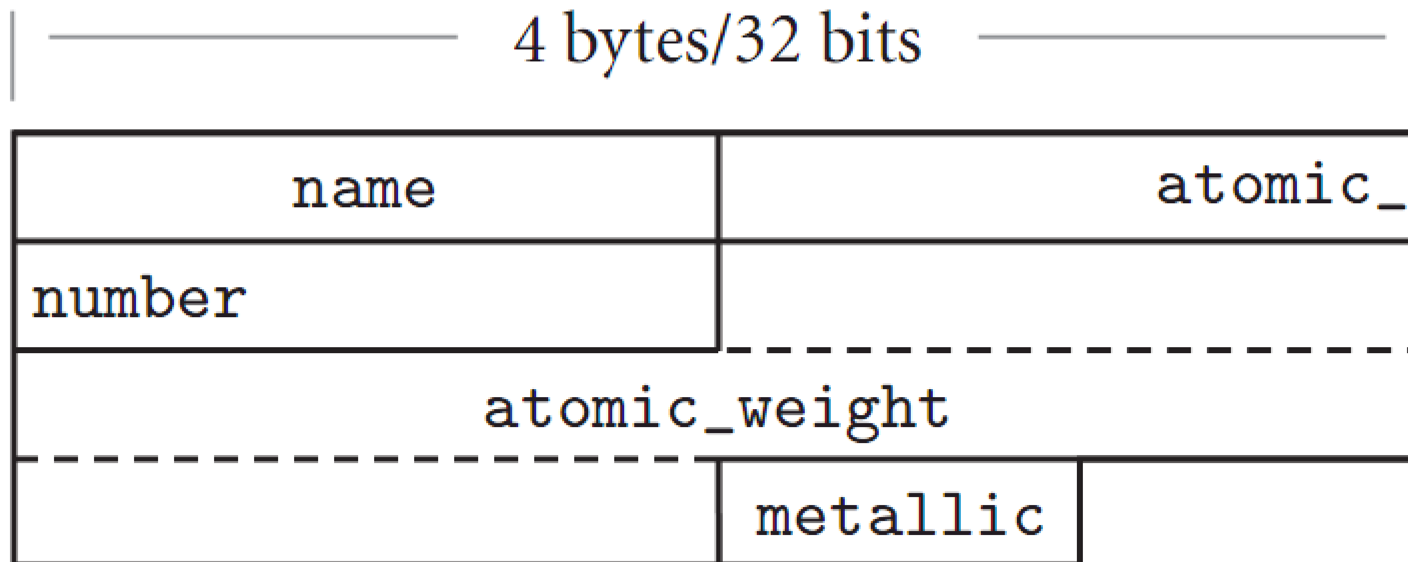


Figure 8.3 Likely memory layout for packed element records. The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

ELSEVIER

Records (Structures)

- Memory layout and its impact (structures)

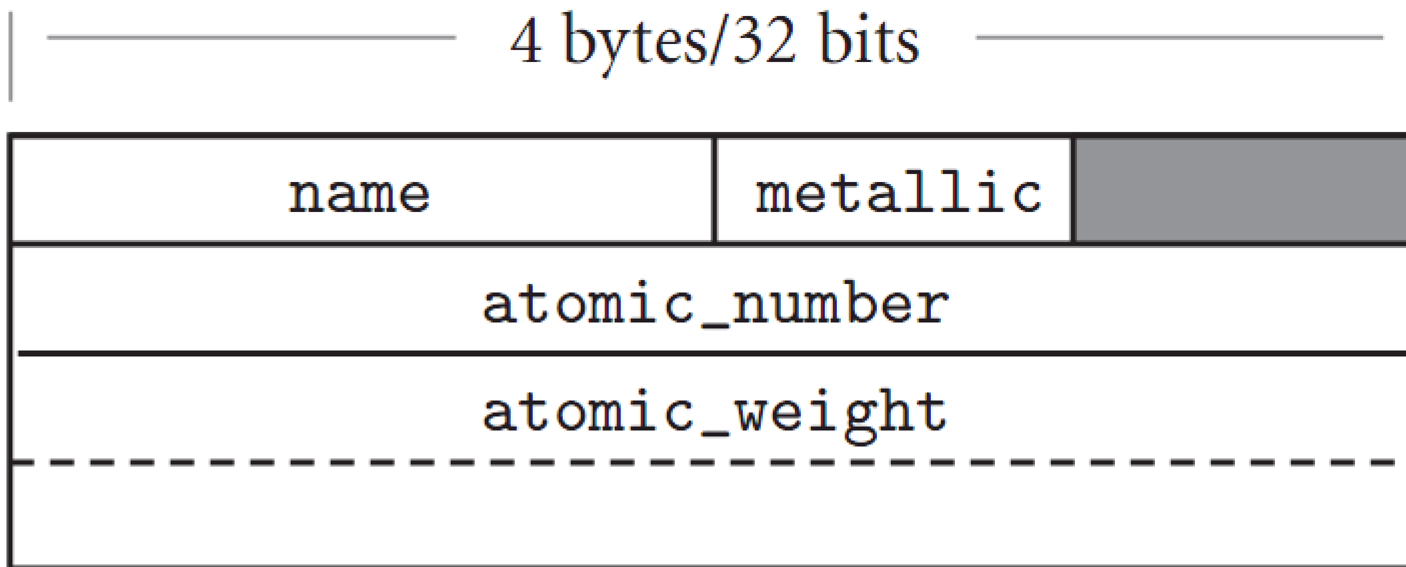


Figure 8.4 Rearranging record fields to minimize holes. By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

ELSEVIER

Unions (a.k.a. datatypes, …)

- A construct that has 2 or more *variants*
  - Every instance is one variant or the other
  - Comes in two forms:
    - Tagged: The runtime uses a tag to keep track of which variant you have, allows you to test the tag; may enforce consistency
    - Untagged: You have to know which variant of the union is intended. You can "roll your own tag" if needed. May be unsafe.

- Example from C

```
struct address {
   int is_street;   // we use this as a tag
   union {
         int po_box;
         char *street_address;
   } address_details;
}
```

- Example from OCaml

```
// OCaml tracks the tag for us
type address =
   po_box of int
|  street_address of string
```

Formalizing Unions as Sum Types

- Syntax for "sum types" – simple unions with tags
  - We model just two possibilities – easy to generalize
  - Instead of arbitrary names, we use "right" and "left"
  - **inr** *e* "injects" a value into a union using the right (r) variant
  - A **case** construct tests the tag and evaluates e_l or e_r

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \\
v & ::= & \ldots \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\
\tau & ::= & \ldots \mid \tau_l + \tau_r
\end{array}
$$

Dynamic Semantics of Sums

$$
\begin{aligned}
e &::= \ldots \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \\
v &::= \ldots \mid \texttt{inl}\ v \mid \texttt{inr}\ v \\
\tau &::= \ldots \mid \tau_l + \tau_r
\end{aligned}
$$

- Congruence rules handle evaluation when injecting into a union or evaluating the input to a case
- The step rules test the tag and run one body or the other—like an if statement

$$\frac{e \to e'}{\texttt{inl}\ e \to \texttt{inl}\ e'}\ \textit{congruence-inl}$$

$$\frac{e \to e'}{\texttt{inr}\ e \to \texttt{inr}\ e'}\ \textit{congruence-inr}$$

$$\frac{e \to e'}{\texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \to \texttt{case}\ e'\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r}\ \textit{congruence-case}$$

$$\frac{}{\texttt{case}\ \texttt{inl}\ v\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \to [v/x]e_l}\ \textit{step-case-inl}$$

$$\frac{}{\texttt{case}\ \texttt{inr}\ v\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r \to [v/x]e_r}\ \textit{step-case-inr}$$

Example of using sums

- Consider modeling addresses as above. The left variant will be PO boxes and the right is street addresses. When we ship, we must use USPS for PO boxes. This function implements that:

ship(address) = **case** address **of inl** n => usps(n), **of inr** a => fedex(a)

ship(**inr** "5000 Forbes Ave") *// ship to CMU!*
→ **case** (**inr** "5000 Forbes Ave") **of inl** n => usps(n), **of inr** a => fedex(a)
→ fedex("5000 Forbes Ave")

ship(**inl** 1492) *// 1492 is the PO box we are shipping to*
→ **case** (**inl** 1492) **of inl** n => usps(n), **of inr** a => fedex(a)
→ usps(1492)

Typechecking Sums

- If we inject a value of type int, the sum type can be int + anything
  - In real languages we know what it is; in the formalism we "guess"
  - We could avoid "guessing" by annotating the **inl** with the expected sum type
- The case rule expects *e* to be a sum, and types the branches assuming the variable has the left and right type, respectively.
  - The branches must have the same type as each other – that way the program can use the result no matter which branch is chosen

$$\frac{\Gamma \vdash e : \tau_l}{\Gamma \vdash \texttt{inl}\ e : \tau_l + \tau_r}\ \textit{T-inl}$$

$$\frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash \texttt{inr}\ e : \tau_l + \tau_r}\ \textit{T-inr}$$

$$\frac{\Gamma \vdash e : \tau_l + \tau_r \quad \Gamma, x : \tau_l \vdash e_l : \tau \quad \Gamma, x : \tau_r \vdash e_r : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{inl}\ x \Rightarrow e_l, \texttt{inr}\ x \Rightarrow e_r : \tau}\ \textit{T-case}$$

ELSEVIER

Sum Subtyping

- Just like depth subtyping for records, one sum is a subtype of another if the component types are in the same relationship. “If I’m expecting a dog or a cat, and you give me a Poodle or a Siamese, I’ll be OK with that”

$$\frac{\tau_l \leq \tau_l' \quad \tau_r \leq \tau_r'}{\tau_l + \tau_r \leq \tau_l' + \tau_r'} \; \textit{S-sum}$$

- If we were modeling sums with more than 2 variants, then a sum with n variants would be a subtype of a sum with m>n variants that includes the n from the first sum. “If I’m expecting a cat, dog, or horse, and you give me a cat or dog, I’m OK with that. But not vice versa!”

- **Exercise: write a rule for this! (assume n-ary sums $\tau_1 + \ldots + \tau_n$)**

ELSEVIER

Sum Subtyping

- Just like depth subtyping for records, one sum is a subtype of another if the component types are in the same relationship. "If I'm expecting a dog or a cat, and you give me a Poodle or a Siamese, I'll be OK with that"

$$\frac{\tau_l \leq \tau_l' \quad \tau_r \leq \tau_r'}{\tau_l + \tau_r \leq \tau_l' + \tau_r'} \; \textit{S-sum}$$

- If we were modeling sums with more than 2 variants, then a sum with n variants would be a subtype of a sum with m>n variants that includes the n from the first sum. "If I'm expecting a cat, dog, or horse, and you give me a cat or dog, I'm OK with that. But not vice versa!"

- **Answer to exercise: write a rule for this (assuming n-ary sums $\tau_1 + \ldots + \tau_n$)**

$$\frac{}{\tau_1 + \ldots + \tau_n \leq \tau_1 + \ldots + \tau_n + \ldots + \tau_m} \; \textit{S-sum-width}$$

- **Note that this is the "opposite" of width subtyping for records!**

ELSEVIER

Records (Structures) and Variants (Unions)

- Memory layout and its impact (unions)

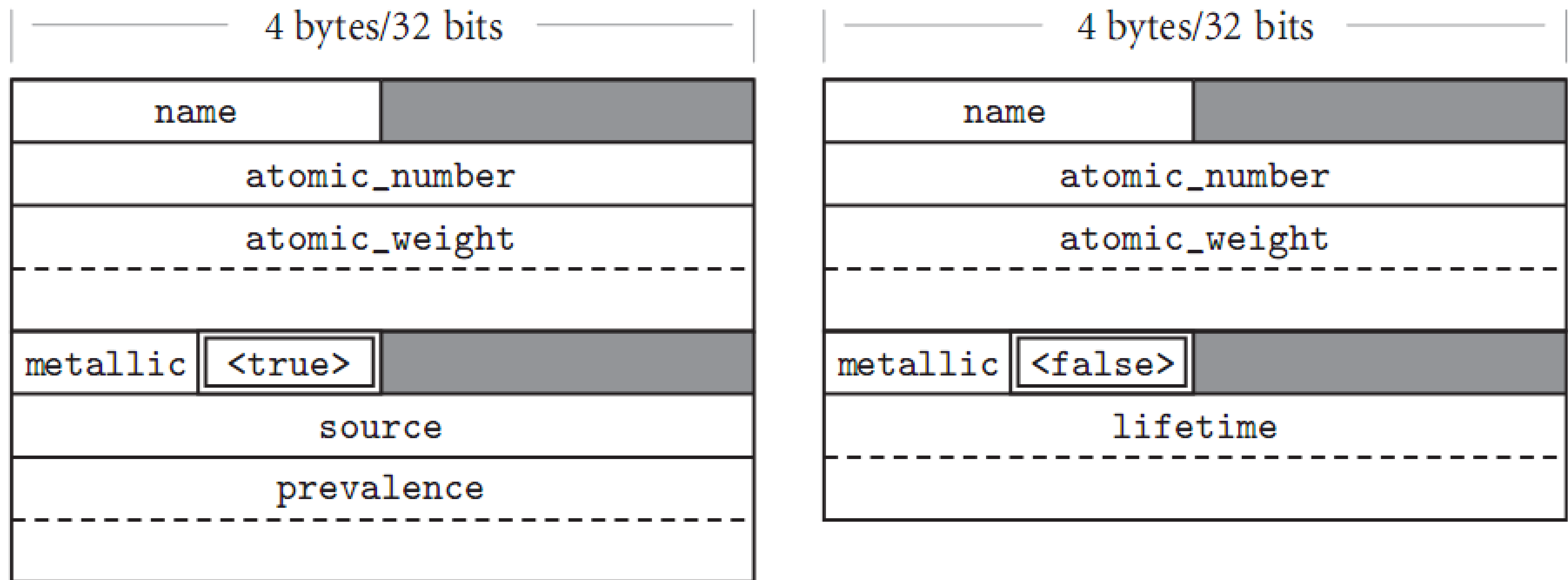


Figure 8.16 (CD) Likely memory layouts for element variants. The value of the naturally_ occurring field (shown here with a double border) is intended to indicate which of the interpretations of the remaining space is valid. Field source is assumed to point to a string that has been independently allocated.

ELSEVIER

Pointers

- Pointers serve two purposes:
  - Efficient access to objects on the stack (as in C)
    - Can be unsafe if not carefully managed
    - Rust has a type system that enforces safety
  - Dynamic creation of linked data structures, in conjunction with a heap storage manager
    - Can also be unsafe if dangling pointers are dereferenced
    - Garbage collection can ensure safety

- Languages like Java provide a higher level “reference” model, “building in” pointers
  - We can model references with pointers though

Modeling Pointers

- We model pointers with three constructs:
  - A **new** operation, as in C++ or Java
  - A C-style dereference operation, *p
  - C-style pointer assignments, *p = e
- Types include pointer types τ* (read from right to left, as in C)
- For modeling execution, we'll track locations *ℓ* on the heap
- A store *S* maps locations to values
- We track the types of locations in the store in a store typing Σ

$$
\begin{array}{lll}
e & ::= & \ldots \mid \texttt{new}\ e \mid *e \mid *e := e \\
v & ::= & \ldots \mid \ell \\
\tau & ::= & \ldots \mid \tau * \\
S & : & Location \rightarrow Value \\
\Sigma & : & Location \rightarrow Type
\end{array}
$$

Pointer Evaluation Rules

- Congruence rules do the expected thing
- But the program is now a combination of an expression and a store!
- Other rules
  - Create references and add them to the store S, creating a new store S'
  - Dereference a value, looking it up in the store S
  - Assign a new value, updating the store

$$\frac{S, e \rightarrow S', e'}{S, \texttt{new}\ e \rightarrow S', \texttt{new}\ e'} \textit{ congruence-new}$$

$$\frac{S, e \rightarrow S', e'}{S, *e \rightarrow S', *e'} \textit{ congruence-deref}$$

$$\frac{S, e_1 \rightarrow S', e_1'}{S, *e_1 := e_2 \rightarrow S', *e_1' := e_2} \textit{ congruence-assign-left}$$

$$\frac{S, e_2 \rightarrow S', e_2'}{S, *v_1 := e_2 \rightarrow S', *v_1 := e_2'} \textit{ congruence-assign-right}$$

$$\frac{\ell \notin domain(S) \quad S' = [\ell \mapsto v]S}{S, \texttt{new}\ v \rightarrow S', \ell} \textit{ step-new}$$

$$\frac{S[l] = v}{S, *\ell \rightarrow S, v} \textit{ step-deref}$$

$$\frac{S' = [\ell \mapsto v]S}{S, *\ell := v \rightarrow S', v} \textit{ step-assign}$$

Pointer Typing Rules

- A new expression has pointer type
- To type a dereference, we look up the type of the pointer and take away the *
- In an assignment, we require *p on the left, where p has pointer type
- The right hand side's type must be a subtype of the pointer type

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{new}\ e : \tau *}\ \textit{T-new}$$

$$\frac{\Gamma \vdash e : \tau *}{\Gamma \vdash *e : \tau}\ \textit{T-deref}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 * \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash *e_1 := e_2 : \tau_2}\ \textit{T-assign}$$

Pointer Subtyping

- As mentioned, we can assign a subtype value to a variable that's a pointer to its supertype:

```
float *r = new 5.0;
*r = 7;   // the compiler will insert a coercion here
```

- But, we can't assign an int * to a float *, or vice versa! That's because int and float have different representations; if we write via one pointer and read from the other, the compiler won't know to insert a conversion, and we'll get garbage.
- Thus, $\tau_1* \leq \tau_2*$ only if $\tau_1 = \tau_2$
- When we study objects, we'll see that in C++ a Dog* is a subtype of an Animal*, but that only works if we use the pointers in a limited way as object references, and do not assign into them.

Recursive Types

- Recursive types refer to a type inside its definition
  - Required to describe recursive data structures
- In practice, combined with other type features
  - C structs are records + recursion
  - OCaml datatypes are unions + recursion
- Running example (Ocaml) – integer lists
  - A datatype with a record in one variant

```
type IntList =
   Cons of { value:int, next:IntList }
|   Nil
```

ELSEVIER

Modeling Recursive Types

type IntList =
  Cons **of** { value:**int**, next:IntList }
| Nil

- We add named recursive types to our type grammar
  - Must also be able to refer to the name

$$\tau \quad ::= \quad \ldots \mid \mathbf{rec}\ T.\tau \mid T$$

- Now we can model lists as follows
  - We use recursive types, sum types, and a record type
  - The names Cons and Nil are just right and left branches of the sum type

rec IntList . { value:**int**, next:IntList } + **unit**

Semantics of Recursive Types

- There are two ways to model the semantics of recursive types
- Both involve *unfolding*
  - We unfold a type by taking the body of the recursive type, and substituting the recursive type for the name everywhere it appears

$$unfold(\mathbf{rec}\ \ T.\tau) = [\mathbf{rec}\ \ T.\tau/T]\tau$$

- The simplest approach, conceptually, is *equi-recursive* types
  - Equi-recursive means the recursive type is equivalent to its unfolding

$$\mathbf{rec}\ \ T.\tau \equiv [\mathbf{rec}\ \ T.\tau/T]\tau$$

- An example of this equivalence for IntList:

rec IntList . { value:**int**, next:IntList } + **unit**

=

{ value:**int**, next:**rec** IntList . { value:**int**, next:IntList } + **unit** } + **unit**

ELSEVIER

Iso-Recursive Types

- Equi-recursive types are attractive, but hard to implement
  - When does the compiler apply the fold/unfold equality?
- A more common approach is *iso-recursive* types
  - Here, a recursive type is *isomorphic* to its unfolding
  - Isomorphic means they behave the same way, but you have to convert between them
  - The compiler inserts a fold when you create an instance of a recursive type; it inserts an unfold when you access it (e.g. with a case or field dereference)
- An operational way to think about fold and unfold:
  - **fold** makes an object into a recursive type, so we can put it in a data structure
  - **unfold** converts an object back to a sum or record, so we can get its contents

Formalizing Iso-Recursive Types

- We now have fold and unfold in expressions. A fold around a value is a value. Remember, these are inserted by the compiler—you don't write them in any real language.

$$
\begin{aligned}
e &::= \ldots \mid \mathbf{fold}_\tau\ e \mid \mathbf{unfold}\ e \\
v &::= \ldots \mid \mathbf{fold}_\tau\ v \\
\tau &::= \ldots \mid \mathbf{rec}\ T.\tau \mid T
\end{aligned}
$$

Formalizing Iso-Recursive Types

- Let's look at how OCaml IntLists turn into iso-recursive types:

type IntList = Cons **of** { value:**int**, next:IntList } | Nil
let list = Cons { value = 3, next = Nil }
in match list **with**
    Cons r => r.value
    Nil => 0

Object created; compiler inserted folds

➔

let list = $\textbf{fold}_{\mathit{IList}}$(**inl** { value = 3, next = $\textbf{fold}_{\text{IntList}}$(**inr** ()) }
in case $\textbf{unfold}_{\mathit{IList}}$(list) **of inl** r => r.value, **of inr** u => 0

datatype match; compiler inserts unfold

where I've abbreviated the single-unfolded IntList type as
IList = { value:**int**, next:**rec** IntList . { value:**int**, next:IntList } + **unit**} + **unit**

ELSEVIER

Formalizing Iso-Recursive Types

- Congruence allows evaluation inside fold/unfold
- When we unfold something that is folded, they cancel:

$$\frac{e \to e'}{\mathtt{fold}_\tau\ e \to \mathtt{fold}_\tau\ e'}\ \textit{congruence-fold}$$

$$\frac{e \to e'}{\mathtt{unfold}\ e \to \mathtt{unfold}\ e'}\ \textit{congruence-unfold}$$

$$\frac{}{\mathtt{unfold}\ \mathtt{fold}_\tau\ v \to v}\ \textit{step-unfold}$$

let list = **fold**$_{IList}$(**inl** { value = 3, next = **fold**$_{IList}$(**inr** ()) }
in case unfold$_{IList}$(list) **of inl** r => r.value, **of inr** u => 0
→
case unfold$_{IList}$(**fold**$_{\text{IntList}}$(**inl** { value = 3, next = **fold**$_{IList}$(**inr** ()) })
of inl r => r.value, **of inr** u => 0

Formalizing Iso-Recursive Types

- Congruence allows evaluation inside fold/unfold
- When we unfold something that is folded, they cancel:

$$\frac{e \rightarrow e'}{\texttt{fold}_\tau\ e \rightarrow \texttt{fold}_\tau\ e'}\ \textit{congruence-fold}$$

$$\frac{e \rightarrow e'}{\texttt{unfold}\ \ e \rightarrow \texttt{unfold}\ \ e'}\ \textit{congruence-unfold}$$

$$\frac{}{\texttt{unfold}\ \ \texttt{fold}_\tau\ v \rightarrow v}\ \textit{step-unfold}$$

case unfold$_{IList}$(**fold**$_{IList}$(**inl** { value = 3, next = **fold**$_{\text{IntList}}$(**inr** ()) })
of inl r => r.value, **of inr** u => 0

→

case (**inl** { value = 3, next = **fold**$_{\text{IntList}}$(**inr** ()) }) **of inl** r => r.value, **of inr** u => 0

→

{ value = 3, next = **fold**$_{IList}$(**inr** ()) }.value

→ 3

Now the typing rules are easy

$$\frac{\Gamma \vdash e : [\texttt{rec}\ \ T.\tau / T]\tau}{\Gamma \vdash \texttt{fold}_\tau\ e : \texttt{rec}\ \ T.\tau}\ \textit{T-fold}$$

$$\frac{\Gamma \vdash e : \texttt{rec}\ \ T.\tau}{\Gamma \vdash \texttt{unfold}\ \ e : [\texttt{rec}\ \ T.\tau / T]\tau}\ \textit{T-unfold}$$

- So, we can typecheck a folded object as follows:

fold$_{IList}$(**inr** ()) } : **rec** IntList . { value:**int**, next:IntList } + **unit**

again, I've abbreviated

IList = { value:**int**, next:**rec** IntList . { value:**int**, next:IntList } + **unit**} + **unit**

Composite Types

- Today we covered the semantics of a number of different composite types
  - Records
  - Unions, datatypes, and sums
  - Reference types
  - Recursive types

Records (Structures) and Variants (Unions)

- Records
  - usually laid out contiguously
  - possible holes for alignment reasons
  - smart compilers may rearrange fields to minimize holes (C compilers promise not to)
  - implementation problems are caused by records containing dynamic arrays
    - we won't be going into that in any detail

Records (Structures) and Variants (Unions)

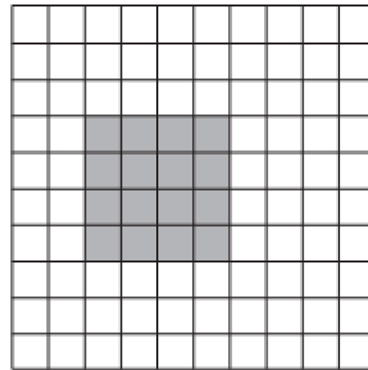
- Unions (variant records)
  - overlay space
  - cause problems for type checking
- Lack of tag means you don't know what is there
- Ability to change tag and then access fields hardly better
  - can make fields "uninitialized" when tag is changed (requires extensive run-time support)
  - can require assignment of entire variant, as in Ada

ELSEVIER

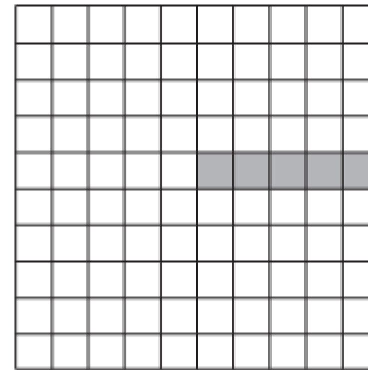
Arrays

- Arrays are the most common and important composite data types
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous
- Semantically, they can be thought of as a mapping from an *index type* to a *component* or *element type*
- A *slice* or *section* is a rectangular portion of an array (See figure 8.5)

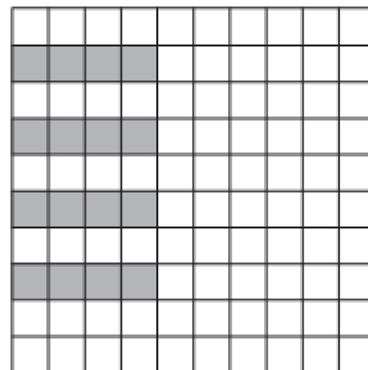
Arrays



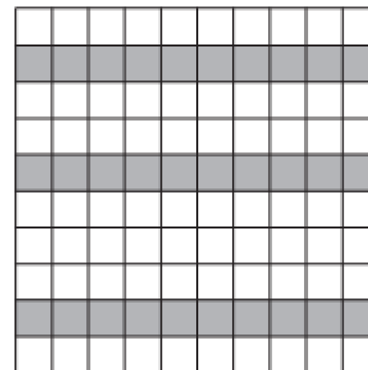
`matrix(3:6, 4:7)`



`matrix(6:, 5)`



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



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

Figure 8.5 Array slices(sections) in Fortran90. Much like the values in the header of an enumeration-controlled loop (Section6.5.1), a: b: c in a subscript indicates positions a, a+c, a+2c, ...through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. It is even possible to use negative values of c in order to select positions in reverse order. The slashes in the second subscript of the lower right example delimit an explicit list of positions.

Arrays

- Dimensions, Bounds, and Allocation
  - ***global lifetime, static shape*** — If the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory
  - ***local lifetime, static shape*** — If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine's stack frame at run time.
  - ***local lifetime, shape bound at elaboration time***

Arrays

```
-- 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];
    ...
}
```

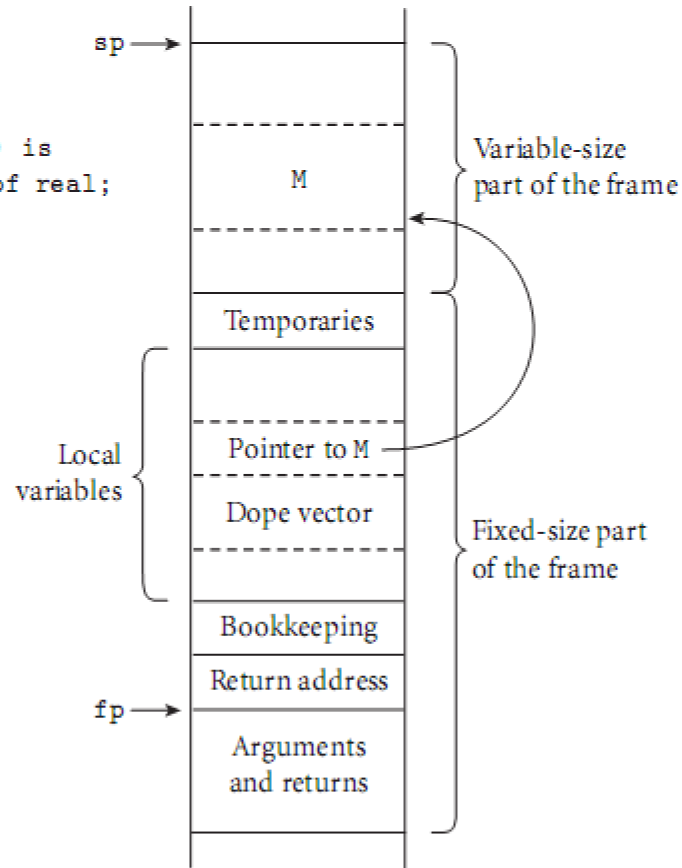


Figure 8.7 Elaboration-time allocation of arrays in Ada or C99

Arrays

- Contiguous elements (see Figure 8.8)
  - column major - only in Fortran
  - row major
    - used by everybody else
    - makes array [a..b, c..d] the same as array [a..b] of array [c..d]

Arrays

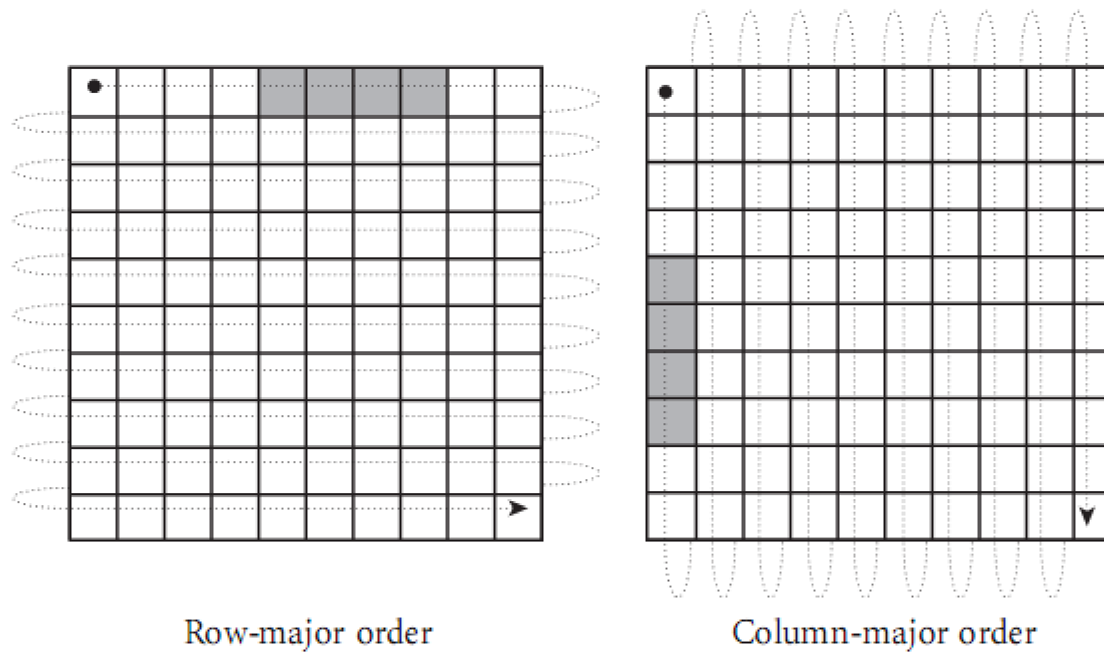


Figure 8.8 Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,9], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

Arrays

- Two layout strategies for arrays (Figure 8.9):
  - Contiguous elements
  - Row pointers
- Row pointers
  - an option in C
  - 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

```
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 */
```

S	u	n	d	a	y				
M	o	n	d	a	y				
T	u	e	s	d	a	y			
W	e	d	n	e	s	d	a	y	
T	h	u	r	s	d	a	y		
F	r	i	d	a	y				
S	a	t	u	r	d	a	y		

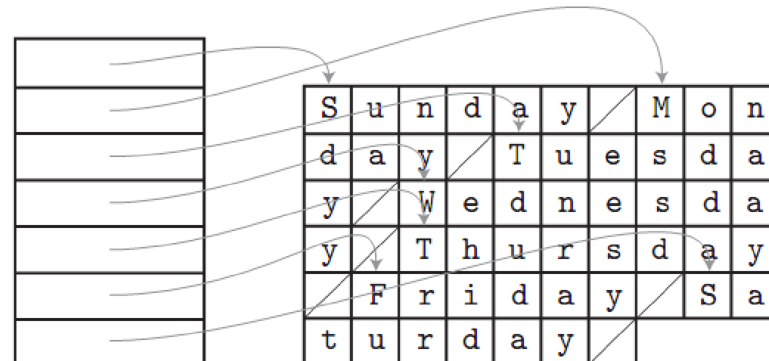


Figure 8.9 Contiguous array allocation v. row pointers in C. The declaration on the left is a true two-dimensional array. The slashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is a ragged array of pointers to arrays of characters. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

Arrays

- **Example 8.25 Indexing a contiguous array**: Suppose

```
A : array [L1..U1] of array [L2..U2] of array
   [L3..U3] of elem;
   D1 = U1-L1+1
   D2 = U2-L2+1
   D3 = U3-L3+1
        Let
```

S3 = size of elem

S2 = D3 * S3

S1 = D2 * S2

Arrays

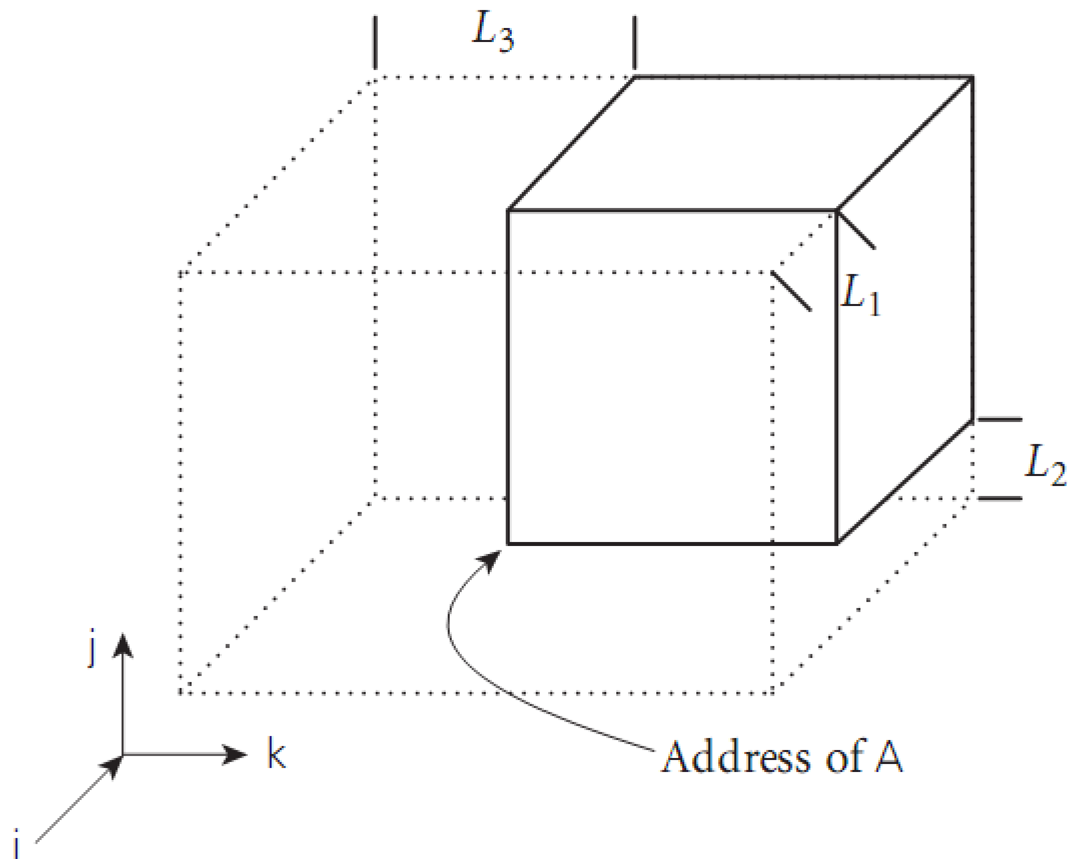


Figure 8.10 Virtual location of an array with nonzero lower bounds. By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory, but whose lower bounds are all zero.

Arrays

- **Example 8.25** (continued)

  We could compute all that at run time, but we can make do with fewer subtractions:

```
== (i * S1) + (j * S2) + (k * S3)
     + address of A
  - [(L1 * S1) + (L2 * S2) + (L3 * S3)]
```

  The stuff in square brackets is compile-time constant that depends only on the type of A

ELSEVIER

Strings

- In some languages, strings are really just arrays of characters
- In others, they are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general
  - It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and (more importantly) non-circular

Sets

- We learned about a lot of possible implementations
  - Bitsets are what usually get built into programming languages
  - Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
  - Some languages place limits on the sizes of sets to make it easier for the implementor
    - There is really no excuse for this

Pointers And Recursive Types

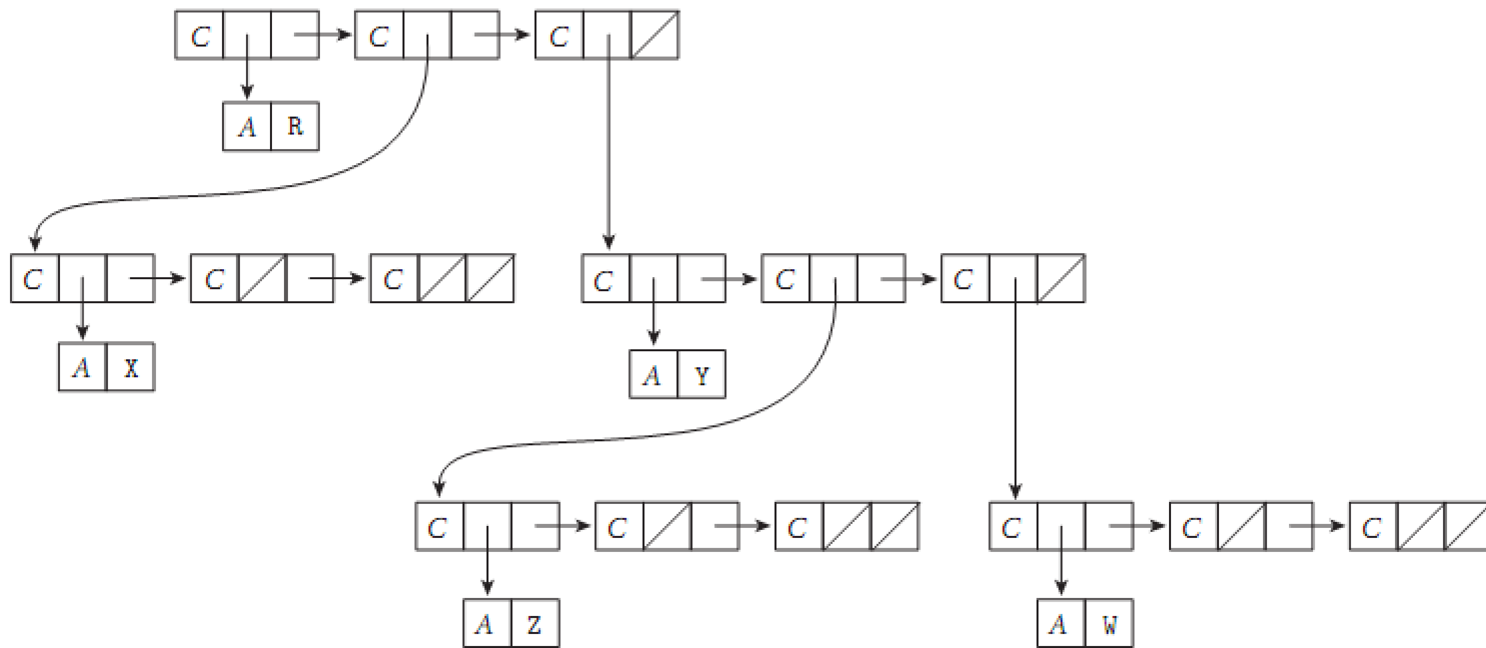


Figure 8.12 Implementation of a tree in LispA diagonal slash through a box indicates a null pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

Pointers And Recursive Types

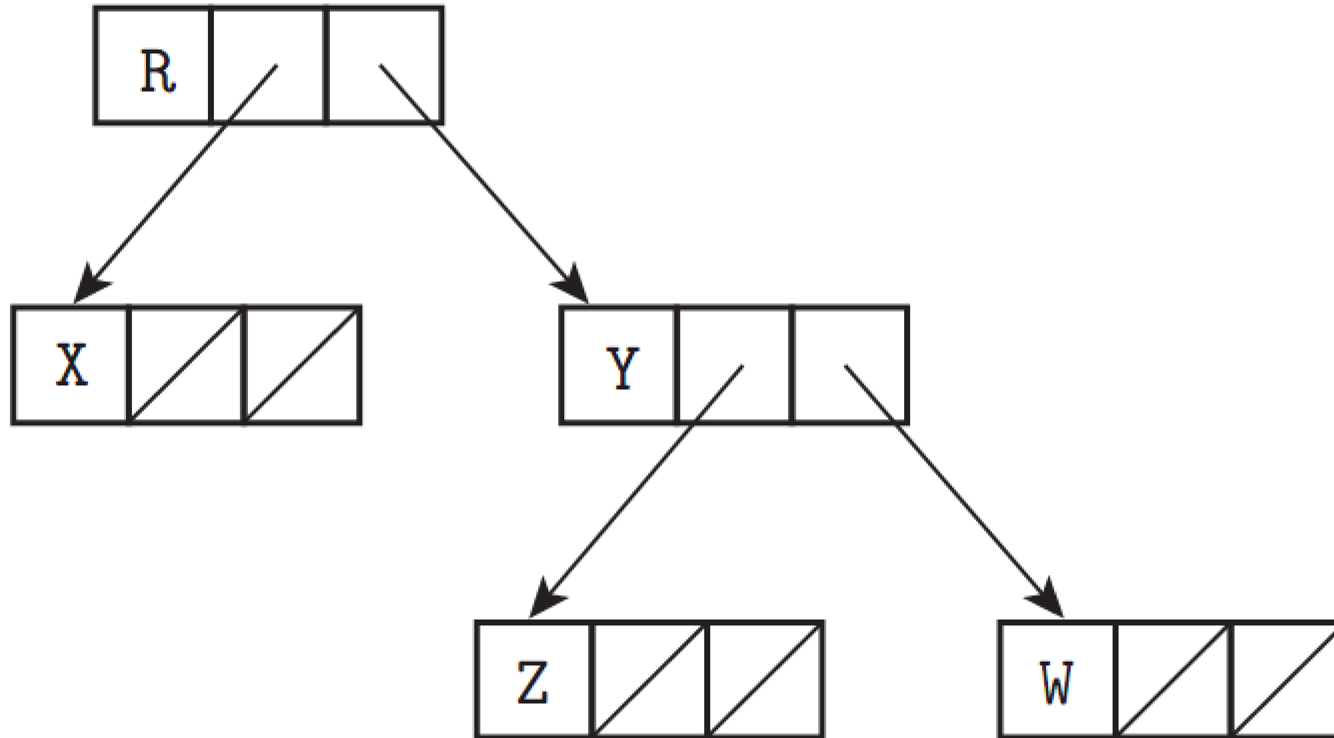


Figure 8.13 Typical implementation of a tree in a language with explicit pointers. As in Figure 8.12, a diagonal slash through a box indicates a null pointer.

ELSEVIER

Pointers And Recursive Types

- C pointers and arrays

```
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

Pointers And Recursive Types

- 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
```

  - 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

Pointers And Recursive Types

- Problems with dangling pointers are due to
  - explicit deallocation of heap objects
    - only in languages that *have* explicit deallocation
  - implicit deallocation of elaborated objects
- Two implementation mechanisms to catch dangling pointers
  - Tombstones
  - Locks and Keys

Pointers And Recursive Types

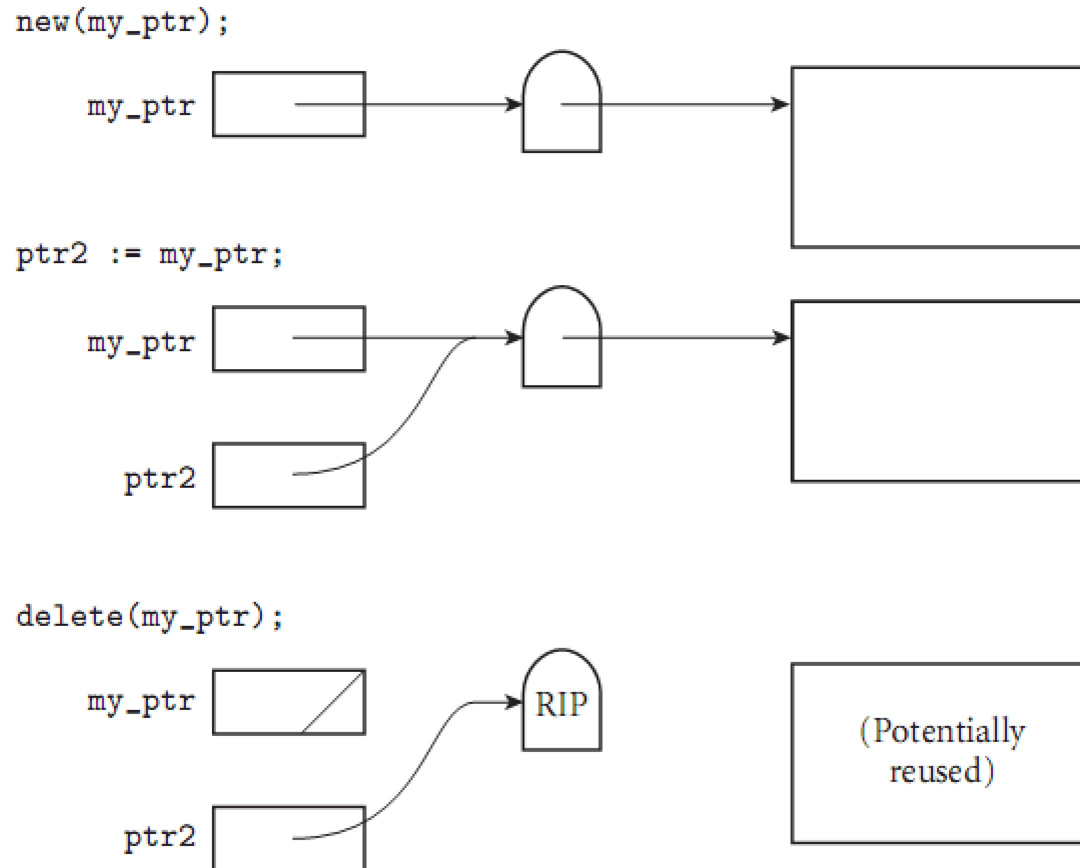


Figure 8.17 (CD) Tombstones. A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

ELSEVIER

Pointers And Recursive Types

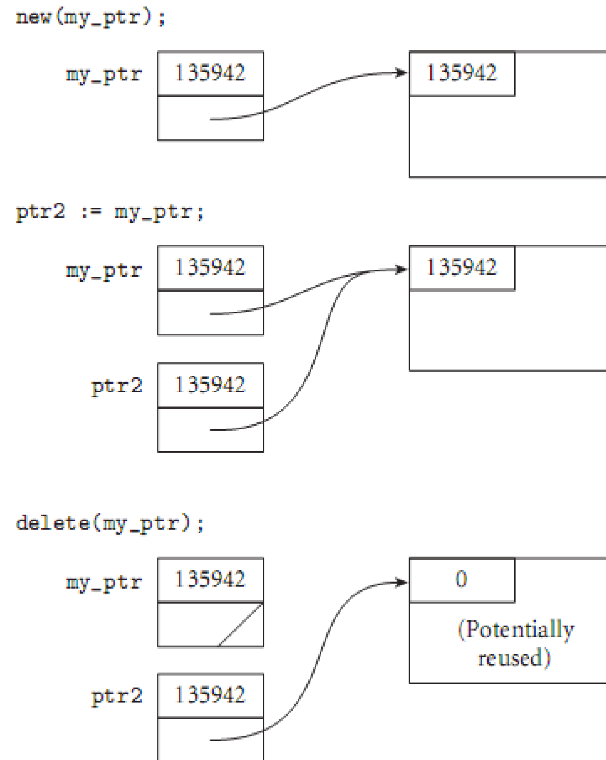


Figure 8.18 (CD) Locks and Keys. A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

ELSEVIER

Pointers And Recursive Types

- Problems with garbage collection
  - many languages leave it up to the programmer to design without garbage creation - this is VERY hard
  - others arrange for automatic garbage collection
  - reference counting
    - does not work for circular structures
    - works great for strings
    - should also work to collect unneeded tombstones

Pointers And Recursive Types

- Garbage collection with reference counts

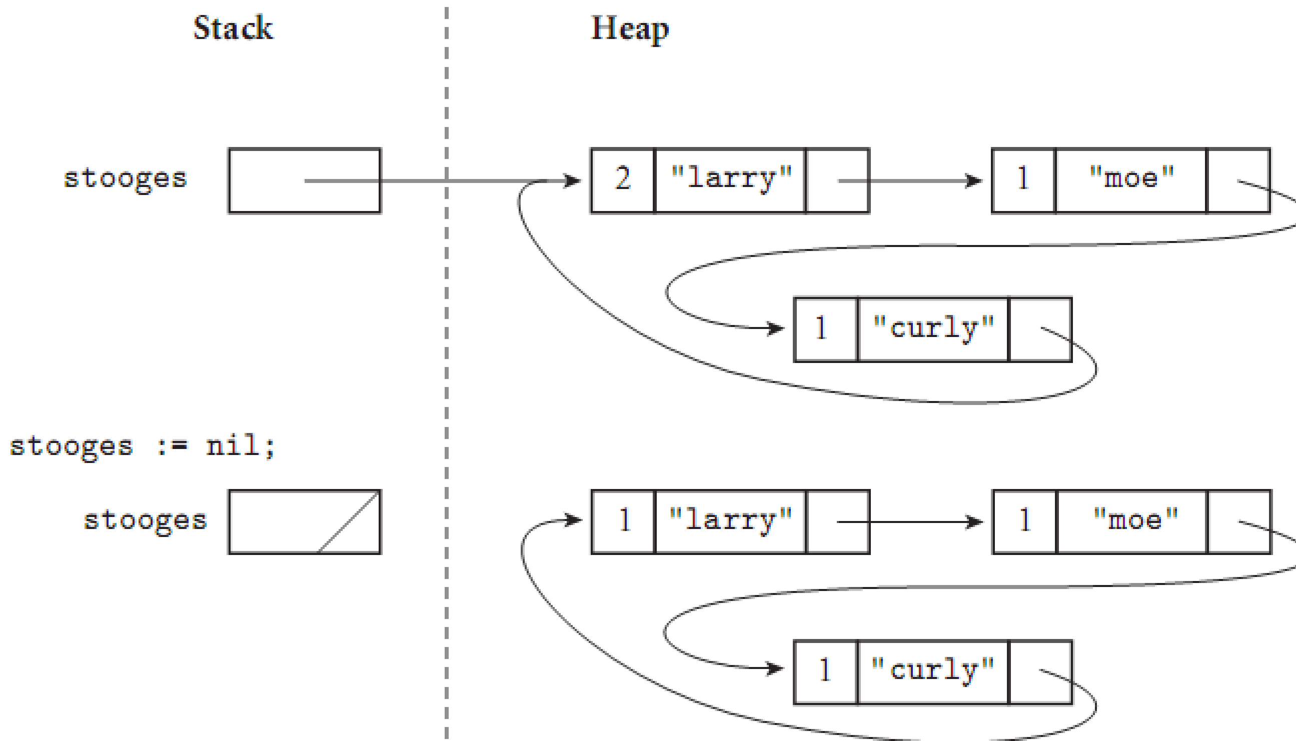


Figure 8.14 Reference counts and circular lists. The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

Pointers And Recursive Types

- Mark-and-sweep
  - commonplace in Lisp dialects
  - complicated in languages with rich type structure, but possible if language is strongly typed
  - achieved successfully in Java, C#, Scala, Go
  - complete solution impossible in languages that are not strongly typed
  - conservative approximation possible in almost any language (Xerox Portable Common Runtime approach)

Pointers And Recursive Types

Figure 8.15 Heap exploration via pointer reversal.

Lists

- A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - Lists are ideally suited to programming in functional and logic languages
    - In Lisp, in fact, a program *is* a list, and can extend itself at run time by constructing a list and executing it
  - Lists can also be used in imperative programs

Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - *interactive* I/O and I/O with files
- Interactive I/O generally implies communication with human users or physical devices
- Files generally refer to off-line storage implemented by the operating system
- Files may be further categorized into
  - *temporary*
  - *persistent*