RECORD TYPES

• A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names

• Design issues:

– What is the syntactic form of references to the field?

– Are elliptical references allowed?

Definition of Records in COBOL

• COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
    02 EMP-NAME.
        05 FIRST PIC X(20).
        05 MID PIC X(10).
        05 LAST PIC X(20).
    02 HOURLY-RATE PIC 99V99.
```

Definition of Records in Ada

• Record structures are indicated in an orthogonal way

```
type Emp_Rec_Type is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
    Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

References to Records

• Record field references

1. COBOL

field_name OF record_name_1 OF ... OF record_name_n

2. Others (dot notation)

record_name_1.record_name_2. ...

record_name_n.field_name

• Fully qualified references must include all record names

Elliptical References

• Elliptical references allow leaving out record names as long as the reference is unambiguous,

for example in COBOL

FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

Operations on Records

• Assignment is very common if the types are identical

• Ada allows record comparison

• Ada records can be initialized with aggregate literals

• COBOL provides MOVE CORRESPONDING

– Copies a field of the source record to the corresponding field in the target record

Evaluation and Comparison to Arrays

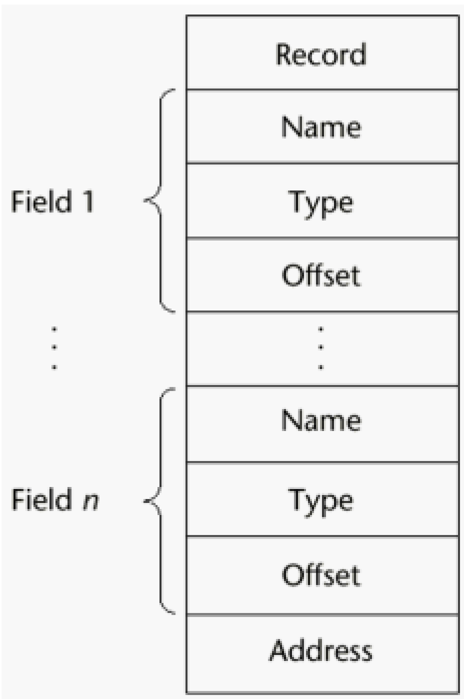
• Records are used when collection of data values is heterogeneous

• Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)

• Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Tuple Types

• A tuple is a data type that is similar to a record, except that the elements are not named

• Used in Python, ML, and F# to allow functions to return multiple values

– Python

- Closely related to its lists, but immutable
- Create with a tuple literal

myTuple = (3, 5.8, ′apple′)

Referenced with subscripts (begin at 1) .Catenation with + and deleted with del

Tuple Types in Python

- Closely related to its lists, but immutable
- Create with a tuple literal

  myTuple = (3, 5.8, ′apple′)

- Referenced with subscripts (begin at 1)
- Catenation with + and deleted with del

Tuple Types in ML

Given

val myTuple = (3, 5.8, ′apple′);

Access as follows:

#1(myTuple) is the first element

A new tuple type can be defined

type intReal = int * real;

Tuple Types in F#

let tup = (3, 5, 7)

let a, b, c = tup

This assigns a tuple to a tuple pattern (a, b, c)

List Types

- Lists in LISP and Scheme are delimited by parentheses and use no commas

  (A B C D) and (A (B C) D)

- Data and code have the same form

As data, (A B C) is literally what it is

As code, (A B C) is the function A applied to the parameters B and C

• The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

′(A B C) is data

List Operations in Scheme

• CAR returns the first element of its list parameter

(CAR ′(A B C)) returns A

• CDR returns the remainder of its list parameter after the first element has been removed

(CDR ′(A B C)) returns (B C)

• CONS puts its first parameter into its second parameter, a list, to make a new list

(CONS ′A (B C)) returns (A B C)

• LIST returns a new list of its parameters

(LIST ′A ′B ′(C D)) returns (A B (C D))

List Operations in ML

• Lists are written in brackets and the elements are separated by commas

• List elements must be of the same type

• The Scheme CONS function is a binary operator in ML, ::

3 :: [5, 7, 9] evaluates to [3, 5, 7, 9]

• The Scheme CAR and CDR functions are named hd and tl, respectively

Lists n F# and ML

• F# Lists

– Like those of ML, except elements are separated by semicolons and hd and tl are methods of the List class

• Python Lists

– The list data type also serves as Python's arrays

– Unlike Scheme, Common LISP, ML, and F#, Python's lists are mutable

– Elements can be of any type

– Create a list with an assignment

myList = [3, 5.8, "grape"]

Lists in Python

• List elements are referenced with subscripting, with indices beginning at zero

x = myList[1] Sets x to 5.8

• List elements can be deleted with del del myList[1]

• List Comprehensions – derived from set notation

[x * x for x in range(6) if x % 3 == 0]

range(12) creates [0, 1, 2, 3, 4, 5, 6]

Constructed list: [0, 9, 36]

List Comprehensions - Example

• Haskell's List Comprehensions

– The original

[n * n | n <- [1..10]]

• F#'s List Comprehensions

let myArray = [|for i in 1 .. 5 -> [i * i) |]

• Both C# and Java supports lists through their generic heap-dynamic collection classes, List and ArrayList, respectively

UNIONS TYPES

• A union is a type whose variables are allowed to store different type values at different times during execution

• Design issues

– Should type checking be required?

– Should unions be embedded in records?

Discriminated vs. Free Unions

• Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called free union

• Type checking of unions require that each union include a type indicator called a discriminant

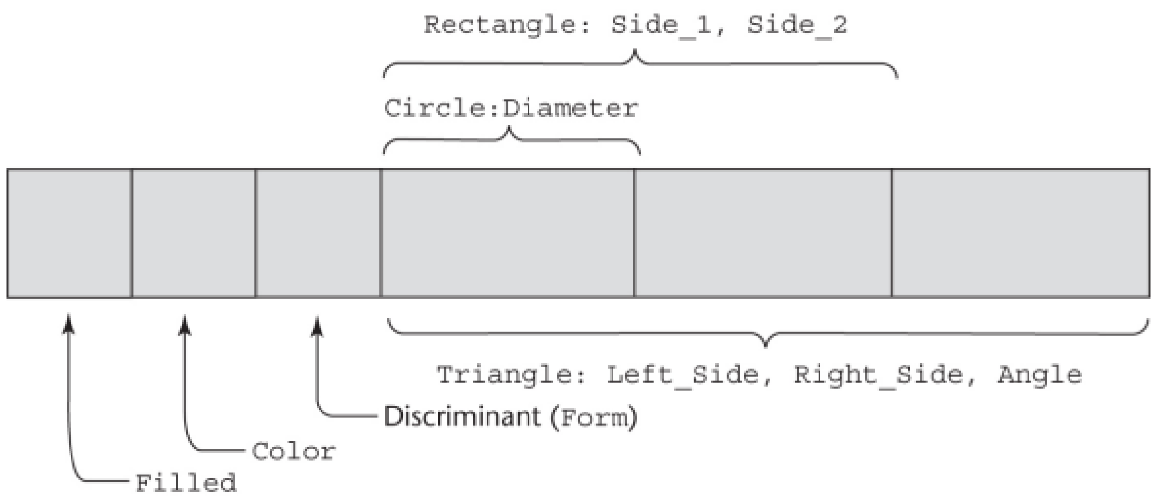
– Supported by Ada

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
```

```
    end case;
end record;
```

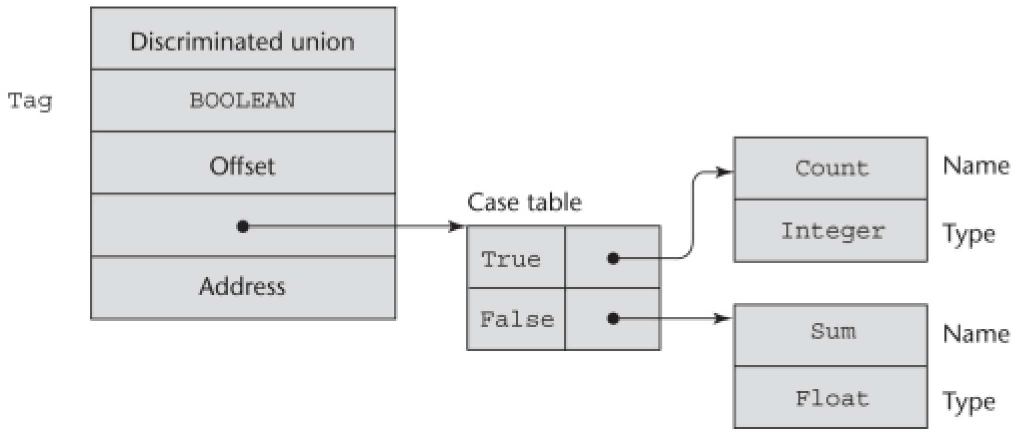
Ada Union Type Illustrated



A discriminated union of three shape variables

Implementation of Unions

```
type Node (Tag : Boolean) is
    record
        case Tag is
            when True => Count : Integer;
            when False => Sum : Float;
        end case;
    end record;
```



Evaluation of Unions

• Free unions are unsafe

– Do not allow type checking

• Java and C# do not support unions

– Reflective of growing concerns for safety in programming language

• Ada's descriminated unions are safe

POINTER AND REFERENCE TYPES

• A pointer type variable has a range of values that consists of memory addresses and a special value, nil

• Provide the power of indirect addressing

• Provide a way to manage dynamic memory

• A pointer can be used to access a location in the area where storage is dynamically created (usually called a heap)

Design Issues of Pointers

• What are the scope of and lifetime of a pointer variable?

• What is the lifetime of a heap-dynamic variable?

• Are pointers restricted as to the type of value to which they can point?

• Are pointers used for dynamic storage management, indirect addressing, or both?

• Should the language support pointer types, reference types, or both?

Pointer Operations

• Two fundamental operations: assignment and dereferencing

• Assignment is used to set a pointer variable's value to some useful address

• Dereferencing yields the value stored at the location represented by the pointer's value

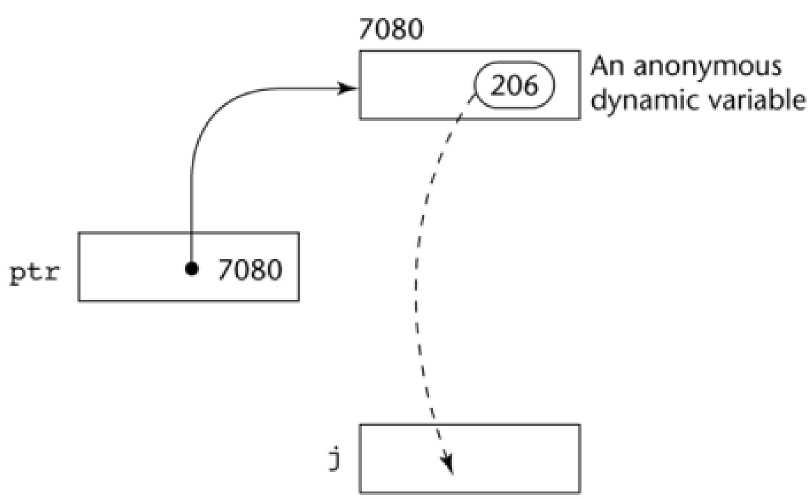
– Dereferencing can be explicit or implicit

– C++ uses an explicit operation via *

j = *ptr

sets j to the value located at ptr

Pointer Assignment Illustrated



The assignment operation `j = *ptr`

Problems with Pointers

• Dangling pointers (dangerous)

– A pointer points to a heap-dynamic variable that has been deallocated

• Lost heap-dynamic variable

– An allocated heap-dynamic variable that is no longer accessible to the user program (often called garbage)

• Pointer p1 is set to point to a newly created heap-dynamic variable

• Pointer p1 is later set to point to another newly created heapdynamic variable

• The process of losing heap-dynamic variables is called memory leakage

Pointers in Ada

• Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope

• The lost heap-dynamic variable problem is not eliminated by Ada (possible with

UNCHECKED_DEALLOCATION)

Pointers in C and C++

• Extremely flexible but must be used with care

• Pointers can point at any variable regardless of when or where it was allocated

• Used for dynamic storage management and addressing

• Pointer arithmetic is possible

• Explicit dereferencing and address-of operators

• Domain type need not be fixed (void *)

void * can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

float stuff[100];

float *p;

p = stuff;

*(p+5) is equivalent to stuff[5] and p[5]

*(p+i) is equivalent to stuff[i] and p[i]

Reference Types

• C++ includes a special kind of pointer type called a reference type that is used primarily for formal parameters

– Advantages of both pass-by-reference and pass-byvalue

• Java extends C++'s reference variables and allows them to replace pointers entirely

– References are references to objects, rather than being addresses

• C# includes both the references of Java and the pointers of C++

Evaluation of Pointers

• Dangling pointers and dangling objects are problems as is heap management

• Pointers are like goto's--they widen the range of cells that can be accessed by a variable

• Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Representations of Pointers

• Large computers use single values

• Intel microprocessors use segment and Offset

Dangling Pointer Problem

• There are several proposed solutions for dangling pointers:

– Tombstone

– Lock and Key

Tombstone

• Tombstone is an extra heap cell that is a pointer to the heap-dynamic variable

• The actual pointer variable points only at tombstones

• When heap-dynamic variable de-allocated, tombstone remains but set to nil

• Costly in time and space

Locks-and-keys

• Locks-and-keys use pointer values that are represented as (key, address) pairs

• Heap-dynamic variables are represented as variable plus cell for integer lock value

• When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Heap Management

• A very complex run-time process

• Single-size cells vs. variable-size cells

• Two approaches to reclaim garbage

– Reference counters (eager approach): reclamation is gradual

– Mark-sweep (lazy approach): reclamation occurs when the list of variable space becomes empty

Reference Counter

• Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell

– Disadvantages: space required, execution time required, complications for cells connected circularly

– Advantage: it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins

– Every heap cell has an extra bit used by collection algorithm

– All cells initially set to garbage

– All pointers traced into heap, and reachable cells marked as not garbage

– All garbage cells returned to list of available cells

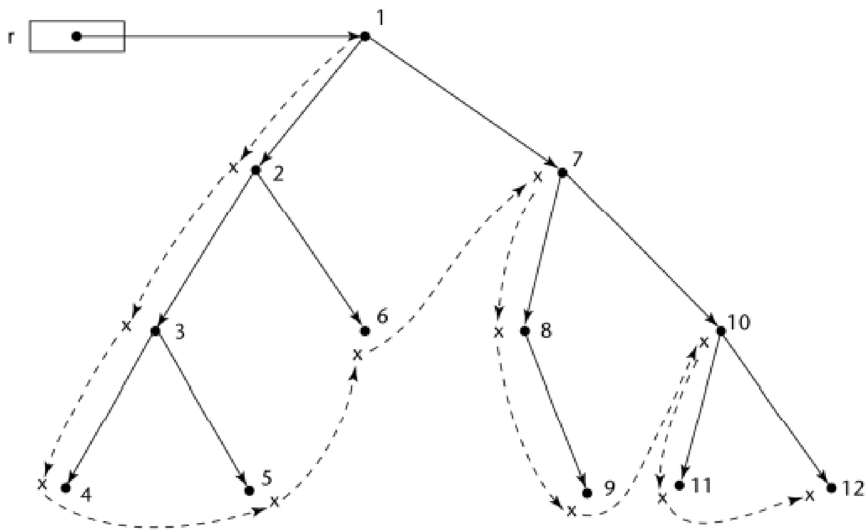
Disadvantages of Mark-Sweep

• In its original form, it was done too infrequently.

• When done, it caused significant delays in application execution.

• Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

• All the difficulties of single-size cells plus more

• Required by most programming languages

• If mark-sweep is used, additional problems occur

• The initial setting of the indicators of all cells in the heap is difficult

• The marking process in nontrivial

• Maintaining the list of available space is another source of overhead

Type Checking

• Generalize the concept of operands and operators to include subprograms and assignments

• Type checking is the activity of ensuring that the operands of an operator are of compatible types

• A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type

– This automatic conversion is called a coercion.

• A type error is the application of an operator to an operand of an inappropriate type

• If all type bindings are static, nearly all type checking can be static

• If type bindings are dynamic, type checking must be dynamic

• A programming language is **strongly typed** if type errors are always detected

• **Advantage of strong typing**: allows the detection of the misuses of variables that result in type errors

Strong Typing – Language Examples

• C and C++ are not: parameter type checking can be avoided; unions are not type checked

• Ada is, almost (UNCHECKED CONVERSION is loophole)

• Java and C# are similar to Ada

Type Coercion

• Coercion rules strongly affect strong typing- -they can weaken it considerably (C++

versus Ada)

• Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Equivalence

• Name type equivalence means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name

• Easy to implement but highly restrictive:

• Subranges of integer types are not equivalent with integer types

• Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

• Structure type equivalence means that two variables have equivalent types if their types have identical structures

• More flexible, but harder to implement

• Consider the problem of two structured types:

– Are two record types equivalent if they are structurally the same but use different field names?

– Are two array types equivalent if they are the same except that the subscripts are different?

(e.g. [1..10] and [0..9])

• Consider the problem of two structured types:

– Are two enumeration types equivalent if their components are spelled differently?

– With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Theory and Data Types

• Type theory is a broad area of study in mathematics, logic, computer science, and philosophy

• Two branches of type theory in computer science:

– Practical – data types in commercial languages

– Abstract – typed lambda calculus

• A type system is a set of types and the rules that govern their use in programs

• Formal model of a type system is a set of types and a collection of functions that define the type rules

– Either an attribute grammar or a type map could be used for the functions

– Finite mappings – model arrays and functions

– Cartesian products – model tuples and records

– Set unions – model union types

– Subsets – model subtypes