Chapter 7: Data Type

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

Data Types

- Two principal purposes
  - provide implicit context for
    - operators and subroutine calls in general
    - e.g. a+b, new p(), overloading
  - limit the set of operations that may be performed
    - e.g. add a character and a record?
    - type systems help to catch typing (and thinking) errors

Chapter contents

- Meaning and purpose of types
- Type equivalence & compatibility
  - Are types T1 and T2 the same?
  - Can we use a value of type T1 in a context expecting a value of type T2?
- Syntactic, semantic & pragmatic issues of most common (and important) types
  - records
  - arrays
  - pointers (also naming issues & heap management)
  - strings, sets, files (also I/O in general)

Type Systems

- Computer hardware
  - can *interpret* bit sequences in various ways
    - instructions, addresses, characters
    - integer & real numbers (of various lengths)
  - machine does not *know* which interpretation is the correct one
  - → assembly languages can operate the memory locations in any way they wish
- High-level languages
  - always associate variables with types with some *type system*
  - to provide the context & to check errors

Components of a type system

- Mechanism
  - to define types and
  - to associate them with other language constructs
- Rules for
  - type equivalence,
  - type compatibility, and
  - type *inference*
    - derive the type of an expression
    - from its parts and from its context

What ‘things’ must have types?

- Anything that may have a value or refer to something having a value
  - constants (named & explicit literals)
  - variables
  - record fields
  - parameters & return values
  - subroutines themselves (if 1st or 2nd class)
  - all expressions containing these
- ‘type of name’ and ‘type of the object named’ can be different!
  - but usually type-compatible
  - important in polymorphic (e.g. o-o) languages
  - we can use the same name to refer to objects of different types

Type checking

- "Process of ensuring that the program follows the type rules"
  - violation = *type clash*
- *Strongly typed languages* (p321)
  - informally: language implementation prevents inappropriate use of objects
- *Statically typed languages*
  - strongly typed and
  - type checking can be carried out at compile-time
  - used often even when some of the tests are run-time (Ada)
- Statically type > strongly type

Some example languages

- Java: strongly typed but not statically typed (type casts)
- Ada: ‘almost’ statically typed
- Pascal: variant records create a loophole in strong typing
- ANSI C: union types, subroutines with varargs, array/pointer –interoperability
- ‘good old’ C: implementations check rarely anything at run-time
- Dynamic scope, late binding → dynamic type checking
  - LISP, Scheme, SmallTalk
- Polymorphism does not necessarily imply dynamic checking
  - Eiffel & type inheritance
  - ML, Haskell & type inference

Definition of types

- Type declaration
  - gives a name to some type
  - happens in some scope
- Type definition
  - describes the type itself
- Declaration <> definition
  - although they quite often appear combined
  - e.g. TYPE intvec = ARRAY[1..10] OF Integer;
- Declaring without defining
  - forward declarations, opaque types, abstract data types, ...
- Defining without declaring
  - ‘anonymous’ types
  - e.g. VAR x: ARRAY[1..10] OF Integer;

Denotational view to types

- Type = set of values
  - Also known as (a.k.a) *domain*
  - the values the objects of that type can take
  - if constant value $v \in T$ then v is of type T
  - if $v \in T$ for all values v of x then x is of type T
- Widely used in formalizing the semantics of programming languages
  - record: n-tuple, array: function
  - assignment: mapping store → store

Constructive view

- Tells ‘how the type is built’
- Built-in types
  - a.k.a primitive, pre-defined
  - integer, Boolean, ...
- Composite types
  - created by applying a type constructor to one or more simpler types
  - ‘simpler types’ may be composite, too
  - typical constructors: record, array, set
- Rest of the chapter focuses on this constructive point of view

Abstraction-based view

- Type is an *interface*
  - set of operations allowed for that type
  - explains the meaning and purpose of the type
- Operations should
  - have well-defined semantics (pre- & post-conditions)
  - respect the data invariant of the type

Built-in types...

- Note: some (but not many) languages may have exceptions to what is said here
- Built-in = same as the ones the hardware supports
- Booleans
  - implemented as 1-byte quantities
  - 0: false, 1: true (other values illegal)
  - C: no boolean type (int 0 = false, anything else = true)
- Characters
  - ASCII encoding → one byte
  - UNICODE → 2 bytes (Java)
  - C++: char & wide char

...Built-in types

- Integers
  - different lengths (C, Fortran)
  - signed and unsigned (Modula-2: cardinal)
- Floating-point numbers
  - different lengths (→ precision & magnitude)

Some non-common builtins...

- Note: languages which don't have these as built-ins quite commonly provide them via libraries
- Fixed-point numbers (Ada)
  - can be implemented as integers
  - fast summation (if same precision)
  - can express large magnitudes compared to floating point numbers with same number of bits
- Decimal types (Cobol, PL/I)
  - some processors support BCD arithmetics

...some non-common builtins

- Complex numbers (Fortran, LISP)
  - implemented as a pair of floating point numbers
- Rational numbers (Scheme, LISP)
  - pair of integers
- Arbitrary precision integers (SmallTalk)

Some terminology

- *Discrete* type (also called *ordinal* type), e.g., integers, booleans, characters
  - countable domain
  - each element has a successor and a predecessor (except min & max element)
- *Scalar* type
  - elements of the type can 'express scale'
  - all numeric types

Enumeration types

- Ordered set of named elements
  - comparisons make sense
  - predecessor, successor
  - enumeration-controlled loops
  - each element has its unique *ordinal value* → mappings
    - Pascal: Ord(c) → ASCII code of c (if c is of type Char)
    - Ada: weekday'pos(mon), weekday'val(1)
  - Ada, ANSI C: ordinal values other than 'default ones'
  - Ada: overloading of enum names is allowed
- Why not use just integers?
  - more readable programs
- Why not just use integer constants?
  - C enum is just syntactic sugar
  - compiler can catch errors when enumerations are real types on their own
  - e.g. one can not use an integer in the place of an enumeration type

Subrange types

- Values comprise a contiguous subset of another discrete type
  - base type, parent type
  - integer, character, enumeration, another subrange
- Ada makes a distinction between
  - derived types (not assignment compatible)
  - constraint subtypes
- Advantages of subranges
  - ‘automatic documentation’ of an integer range
  - compiler can generate range checking code
  - compiler can ‘compress’ the subrange (120..125 needs only 3 bits)
  - usually the implementation takes the ‘expected’ amount

Common composite types

- Records (structures)
  - collection of *fields*
  - Cartesian product of (field) domains
- Arrays (vectors, tables)
  - function from *index* type to *component* type
  - strings are quite often ‘just’ arrays of characters with some special operations
- Variant records
  - union of field types
  - alternative fields under one name, only one alternative is valid at a time

...common composite types

- Sets
  - powerset of its (discrete) base type
- Pointers (l-values)
  - references to objects of pointer's base type
  - often implemented as machine addresses (not necessary!)
  - requirement for recursive data structures
- Lists
  - sequences of elements (like arrays)
  - recursive definition instead of an indexing function
  - variable length
  - fundamental to functional & logic languages
- Files
  - data on mass storage devices
  - like arrays (if 'seek' allowed) with known 'current position'
  - like lists (if only sequential access allowed)

Type checking

- Typed objects
  - every definition of an object must specify also the object’s type
- Typed contexts
  - rules of the language tell what types are allowed in each context
  - sometimes finding this out requires *type inference*
- Type checking
  - may an object of type T be used in some given context?
  - if types are *equivalent* (same): yes
  - if types are *compatible* : depends on the language
    - casts / conversions
    - coercion
    - nonconverting casts

Type equivalence

- Two principal ways
- *Structural equivalence*
  - based on the *content* of definitions
  - (roughly put) types are the same if they
    - consist of same components and
    - they are composed in the same way
  - Algol 68, Modula-3, C & ML (with various ‘wrinkles’)
- *Name equivalence*
  - based on the lexical occurrence of definitions
  - each definition defines a new type
  - more popular in recent languages (Java, Ada)
- Note: separate compilation creates some problems
  - see section 9.6

What is structurally equivalent?

- See examples on page 331
- What differences are important and what not?
  - format of declaration
  - order of fields in a record
  - representations of same constant values
  - index values of an array
- Algorithm to decide structural equivalence
  - expand all definitions until no user-defined types are left
  - check if the 2 expanded definitions are the same
  - recursive types give some trouble (must match graphs)

Problems with structural eq

- Unintentional equivalence (p. 332)
  - programmer defines 2 types that have nothing in common
    - different name
  - but the type system thinks they are the same
    - same internal structure
- Name equivalence resolves this
  - ‘if programmer takes the effort to define 2 types then he most probably has the intention that those types are different’ (otherwise he would define only one)

Name equivalence

- Aliasing
  - define a type using just the name of another type
- Problem
  - are these 2 types the same (name equivalent) or not?
  - essential for Modula-2 example to work (p. 332)
  - but sometimes we do *not* want this (p. 333)
- Strict name equivalence
  - aliased types are distinct
- Loose name equivalence (Pascal, Modula-2)
  - aliased types are considered equivalent
- Ada: ‘best of both worlds’
  - *derived* type: incompatible with base type
  - *subtype*: compatible
- Modula-3: branded types (otherwise structural eq)

Strict and loose

- TYPE A = B
  - strict name equivalence: a language in which aliased types are considered distinct (declaration and definition are distinct)
  - loose name equivalence: a language in which aliased types are considered equivalent (just a declaration, A shares the definition of B)
- Example on p.333 (bottom of the page)
  - strict: p & q & t, r & u
  - loose: r & s & u, p & q & t
  - structural: all 6 variables

Type conversions

- Contexts expecting values of a specific type
  - assignment
  - expressions with overloaded operators
  - subroutine calls
- Suppose types must match exactly
  - → explicit type conversions required
- Conversion depends on the types
  - types are structurally equivalent, conversion just makes them name equivalent
    - → no run-time code
  - different subsets of values, common values are represented in the same way
    - e.g. signed & unsigned integers
    - → check that the value is in the common area, then use the machine representation as such
  - different low-level representations
    - → must use some mapping routine
    - 32 bit integer → 64 bit float: ok
    - opposite direction: loss of precision (round/trunc), overflow

Nonconverting type casts

- Change the type of the value *without* changing the underlying implementation
  - occasionally useful in systems programming
  - example 1: memory allocation
    - heap is allocated as an array of (say) integers
    - it can contain addresses and different user-defined data structures
  - example 2: high-performance arithmetics
    - treat IEEE floating point number as a record
    - use exponent, sign & mantissa as integers

...nonconverting casts

- Ada
  - generic subroutine ‘unchecked_conversion’
- C
  - type cast → run-time conversion with no checking
  - nonconverting casts possible by ‘clever’ use of pointers
  - also possible with union types (and variant records in other languages)
- C++
  - static_cast: type conversion
  - reinterpret_cast: nonconverting
  - dynamic_cast: run-time check
- Dangerous!

Why type compatibility?

- A := B
  - type of B must be compatible with the type of A
- A + B
  - types of A & B must be compatible with integer type or with float type
- C := p(A,B)
  - types of A & B must be compatible with the types of the formal parameters of p
  - return value of p must be type compatible with C

Examples of type compatibility

- Ada: type S is compatible with type T iff
  - S & T are equivalent or
  - S is a subtype of T (or vice versa) or
  - S & T are subtypes of the same type or
  - S & T are arrays with same dimensions, ranges and component types
- Pascal
  - integers can be used in the place of reals

Implementing type compatibility

- Scenario
  - A & B are type compatible → A := B allowed
  - A & B have different semantics (e.g. subrange) → compiler must generate type checking code
  - A & B have different low-level representation → compiler must convert B to the type of A
- Coercion
  - implicit type conversion provided automatically by the compiler
  - may require run-time code
    - checks (Ada coercions need only these)
    - actual conversions

To coerce or not?

- Coercion
  - allows types to be mixed without explicit indication from the programmer
  - weakens significantly type security
  - ‘the weaker the type system, the more coercions the language provides’ (Fortran & C)
    - most numeric types can be intermixed
    - compiler coerces results ‘back and forth’ when necessary
- Example on page 338

...to coerce or not

- Most modern languages try to
  - get closer to strong typing and
  - further from coercions
- But not C++
  - motivation: coercions are the natural way to support data abstraction & program extensibility
  - extremely rich programmer-extensible set of coercion rules
  - programmer can define coercion functions for his own classes
  - add overloading and templates to this and you'll have the most complicated type system ever created

Type Inference

- Type checking ensures that
  - components of an expression
  - are type compatible with the expected component types of that expression
  - but how to find out the 'type of an expression'?
- Often easy
  - function call: corresponding function result type
  - assignment statement: type of assigned value
- Problematic case: operations that do not preserve the types of their operands
  - operations on subranges
  - operations on (some) composite types

Arithmetics on subranges

- See example on p. 341
  - what is the type of ‘a + b’?
    - new range 10..40?
  - Pascal (and descendants)
    - base type of the subrange (integer in this case)
- for-loop in Ada
  - subrange tells the type of the index variable
  - for compatibility: type = base type of range bounds
- avoiding run-time checks
  - compiler can keep track on min/max bounds
  - some checks may be avoided this way (or half of the check)
  - sometimes we may catch even semantic errors (low bound 1 > high bound 2)
  - not always possible (user-defined functions, p. 342)

Operations on composite types

- Result of operation is different from types of operands
- Example: strings in Ada (p. 343)
  - string is an ‘incomplete’ type
  - string of length n is compatible with *any* array of characters of length n
  - the actual range does not matter
  - → the type of the result of string catenation depends on the context

Records and Variants

- We skip subsection 7.2.5
- Structures and unions (p.351)
  - C++: struct is a special for of a class (or vice versa)
  - Java: class is the only ‘struct-like’ type constructor
- Pascal & C syntax for records (p. 351)
  - records consist of named *fields*
  - anonymous fields → tuple (ML)
- Referring to fields
  - usually referred using the ‘dot notation’
    - Fortran 90: %-notation
  - some languages use functional notation
    - projection functions
    - ML: #fieldname record-object
- Nested definitions (p. 352)
  - directly (Pascal) or using intermediate structures (F90)

Implementation

- Prime reason why the order of the fields in a record should matter
  - fields are usually stored after each other
- Accessing a record field
  - find base pointer (frame/global)
  - add to that
    - record's offset from the base and
    - field's offset in the record
  - generate corresponding load/store instruction
  - assumes *alignment,* i.e. fields start at memory word boundaries
- Example: Figure 7.1
  - alignment creates 'holes' in the memory layout
  - array of such records would allocate 20 bytes for each

Packed records

- Pascal keyword PACKED
  - can be applied to record, array, file, set
  - tells the compiler to use minimum amount of memory
  - ‘push fields together’
  - accessing fields is slower
    - collect pieces and reassemble them to registers
    - we trade memory for speed
- Example in Figure 7.2 (p. 354)
  - array of these would allocate 16 bytes for each
  - PACKED array would allocate 15

Record operations...

- Assignment r1 := r2
  - most languages allow this
  - naive implementation: copy each field separately
  - fast implementation: use block memory transfers
    - just transfer all bits of r2 into r1
    - **`block_copy(source, dest, length)`**
    - hardware support

...Record operations

- Comparison r1 = r2
  - most languages do NOT support this
    - exception: Ada
    - in C++ (and many others) one can program own equality tests for own classes
  - implementation
    - block compare
      - problem: also the garbage in the holes gets tested
      - → always fill holes with zeroes (takes time)
    - field-by-field comparison

Saving space

- Holes in records waste space
  - packing → heavy cost in access time
- Compromise solution
  - rearrange fields so that wasting caused by word-alignment is minimal
  - greedy heuristics for this minimization
    - sort fields according to their (alignment) size
    - place smallest fields first
      - bytes, half-words, words, double words, arrays, ...
    - → larger fields are never (unnecessarily) split over several words
  - Compare examples in Figures 7.1, 7.2, and 7.3

Does the ordering matter?

- Usually not
  - compiler can rearrange fields as it wishes
- Some systems programming tasks
  - require knowledge of the exact location and length of the fields
  - → systems programming languages
    - allow programmer to specify these
    - C, C++ guarantee that the order is not changed anyway

WITH statements & records

- Introduced in Pascal
  - aim: simplify the manipulation of deeply nested structures (x1.f.g.y := x2.f.g.y)
  - example pp. 355-356
- WITH statement opens a new scope
  - fields of the *opened* record become normal variable names
  - formalize the notion of *elliptical references* of Cobol
    - allows the use of a field name as a variable if it's unique

...WITH statements

- Problems
  - How to manipulate the fields of 2 similar records simultaneously?
  - Naming conflicts
    - new scope → local variables inaccessible
  - Long and nested statements
    - which field comes from which WITH record
    - type definition may be very far
- Modula 3 solution
  - WITH creates *aliases* instead of opening records
  - fields are not directly visible but accessible via aliases
  - aliases can be used for other objects, too
  - examples on page 357

WITH without WITH

- C simulation
  - use local pointer variables as aliases
  - needs the capability of
    - declaring variables in nested blocks
    - addressing stack (non-heap) variables
    - Pascal has neither
  - C++: use reference types instead
- → implementation
  - each WITH creates a local 'hidden pointer' to the opened or aliased record
  - access to fields via this pointer & offsets
  - good optimizer *might* 'invent' these automatically

Variant records

- Aim
  - provide 2 or more *alternative* fields
  - only one of them is valid at a given time
- Pascal variant record (p. 358)
  - tag field (naturally_occurring)
  - variants (in parentheses)
- Implementation
  - variants may share the same space (Fig. 7.4)
  - origin: equivalence –statement of Fortran I (use same space for different variables)

Why is 'variant' better than union?

- Pascal integrates variants with records
  - variations only seldom appear elsewhere
  - variant fields can be accessed with standard dot-notation
- C & unions (p. 359)
  - need to create intermediate structures
  - → extra levels of naming to access variant data

Arrays

- ‘Mother of mass-computation’
  - homogenous collection of elements
    - records: heterogenous
  - most common and important composite data type
  - fundamental part of any programming language
- Semantics
  - mapping from an index type to a component (element) type
  - most languages restrict index to be of a discrete type
    - more general arrays require a hash-table implementation
    - C++, Java: maps
  - elements can usually be of any type
    - Fortran 77: components must be scalars

Array syntax...

- Accessing elements
  - Pascal, C, ...: A[3]
    - no confusion with subroutine calls
  - Fortran, Ada: A(3)
    - Fortran: keypunch machines did not have ‘[‘ ‘]’
    - Ada: deliberate design decision
      - arrays are mappings, that is, functions
      - easy to replace an array with the corresponding mapping (or vice versa)
      - see Figure 7.6

...Array syntax

- Declaring array types
  - append subscript notation to a ‘normal’ scalar declaration
    - C: **`char upper[26]`**, lower bound = 0
    - Fortran: **`character(26) upper`**, l.b. = 1
  - use array constructor
    - Pascal: **`upper: ARRAY[‘a’..’z’] OF Char;`**
- Multidimensional arrays
  - syntactic sugar for ‘arrays of arrays’
  - Ada makes a difference between
    - a 2-dimensional array and
    - an array of 1-dimensional arrays
    - the latter is more flexible to use (matrix(3) is a normal array)
  - C: **`int matrix[3][4]`**
    - **`matrix[3]`** is a reference (to int or an array of ints, depends on context)

Array operations

- Selecting & assigning elements
- Slices / sections
  - Fortran-90: many operations
    - slice = rectangular portion of an array
    - Figure 7.7: matrix & some slices
  - Ada supports only 1-dimensional slices
    - slice = contiguous subrange of elements
- Comparing equality
- Ada
  - lexicographic ordering (A < B) for 1-dim arrays of discrete elements
  - OR/AND/XOR on Boolean arrays
- Fortran 90, APL: many built-in array operations
  - A + B, tan(A), ...
  - structural equivalence → same element type & shape (good when using slices)
  - most built-in scalar operations generalize to arrays
  - also ‘array-specific’ operations (like matrix transposition)

Allocating arrays

- Depends on
  - lifetime of the array
  - the time the shape of the array is known
- Possibilities
  - Global lifetime, static shape
    - bounds & dimensions known at compile-time
    - allocate from global memory area
  - Local lifetime, static shape: recursive subroutines
    - allocate from stack frame
  - local, shape bound at elaboration time (Figure 7.8, p.370)
    - divide stack frame to fixed & variable part
    - allocate a pointer from fixed part, array itself from variable
    - nested definitions → delay array allocation
  - arbitrary, elaboration time (e.g. Java): use heap
  - dynamic shape
    - must use heap (array may grow from both ends)
    - re-allocation & copy when necessary

Memory layout

- Elements in contiguous locations
  - possible alignment holes (esp. with records)
- Multidimensional arrays
  - row-major order
    - ‘last’ dimension grows first in consecutive locations
    - A[1,1], A[1,2], ..., A[1,max2], A[2,1], ...
    - most languages use this
  - column-major order
    - ‘first’ dimension grows first in consecutive locations
    - A[1,1], A[2,1], ..., A[max1,1], A[1,2], ...
    - Fortran
  - straightforward generalization to m > 2 dimensions

Row- or column order?

- Row-major
  - easy to define matrix as an array of subarrays
- Computational efficiency
  - better performance if array elements are in cache
  - cache miss → several elements of array are loaded
  - if subsequent indices use these then we are doing well
  - Fig. 7.10: good cache hit ratio with row-order, worse with column order
  - the ‘good’ and ‘bad’ depend on the program!
  - one might implement BOTH orders and use the appropriate one

Row-pointer implementation

- Memory layout
  - rows can be anywhere in the memory
  - an auxiliary array of pointers to rows
  - generalizes to m > 2 dimensions
- Advantages
  - sometimes faster to access row elements
    - may depend on hardware (indirect addressing vs. multiplication)
  - rows can be of different length
- May waste or save space
  - pointer array takes some space
  - ‘dynamic’ lengths of rows may save more
- Languages
  - C & C++ have both row-major & row-pointer (Fig. 7.11)
  - Java uses row-pointer

Address calculations

- Example
  - 3-dimensional array with row-major ordering
    - generalizes easily to any number of dimensions
    - computation is similar for column-major case
  - A: [L1..U1, L2..U2, L3..U3]
  - Define
    - S3 = size of the element type
    - S2 = size of a row = (U3 – L3 + 1)*S3
    - S1 = size of a 2-d plane = (U2 – L2 + 1)*S2
  - address of A[i,j,k]?
    - = &A + (i – L1)*S1 + (j – L2)*S2 + (k - L3)*S3

Faster address calculations

- Previous computation involves
  - 5 multiplications and 10 additions/subtractions
- IF
  - Li & Ui (i=1,2,3) are known at compile-time
- THEN
  - Si (i=1,2,3) are compile-time constants
  - → move substractions of Li out of the formula
  - &A[i,j,k] =
    - &A + i*S1 + j*S2 + k*S3 (runtime computation)
    - - [(L1*S1) + (L2*S2) + (L3*S3)] (compile-time constant)
  - 3 multiplications & 4 additions/subtractions
    - if A is a global/static variable then also &A is a compile-time constant
  - corresponding machine code on page 376

Restricted & generalized cases

- Indexes (i,j,k) may be known at compile-time
  - move to the ‘static part’ of computation
- Lower/upper bounds may be unknown
  - move to the ‘dynamic part’ of computation
- Example (in the paragraph of p.377)
  - L1 not known, k = 3
- C, C++, Java
  - lower bounds always 0 → they never contribute to runtime cost

Static & dynamic address computations

- This far only arrays, but the idea can be used for any structures
- Example (p. 378)
  - V = local array of records R
  - R has a 2-dimensional array in field M
  - &V[i].M[3,j] = ?

Row-pointer addresses

- Computations much simpler
- A[i,j,k] =
  - (*(*A[i])[j])[k] in C notation
  - A[i]^[j]^[k] in Pascal notation
  - instruction sequence on p. 378
- Speed vs. row-major implementation
  - earlier machines had so slow multiplication that indirect addressing was faster

Strings

- (just) an array of characters or
- a special data type with own operators
  - dynamic array
    - even if the language doesn't support them otherwise
  - many applications require strings
  - strings are easier to implement than arrays in general
    - 1 dimension, byte elements

String Literals

- Sequence of characters in quotation marks
  - character literals (char = string of length 1?)
  - escape sequences for non-printable characters
    - C: ‘\t’ (tab) ‘\n’ (newline), ‘\006’ (octal! ascii code)
    - Java: C + numeric escapes ‘\uxxxx’ for Unicode characters

String operations

- Often implementation-dependent
  - size known at elaboration time
    - → contiguous array of characters
    - restricted operability
    - lexicographic ordering (<, >)
    - C: no built-in operations
  - size can change dynamically
    - → heap implementation (block, chain of blocks)
    - concatenation, length
    - substrings, pattern matching
    - ability to define own string-valued functions

Sets

- Collection of elements (like arrays)
  - homogenous
  - element type = *base type* of the set
- Different from arrays
  - unordered
  - all elements are different
  - size arbitrary
- Part of Pascal language
  - many others have library support
  - creation, literals, union, intersection, difference

Implementing sets

- Numerous standard data structures
  - e.g. tree structures
- Usually as a bit vector
  - bit i = 1 → ith element is a member of the set
  - bit i = 0 → ith element is not a member of the set
  - suits only for small base types
    - base domain of size n needs a vector of n bits
    - 32-bit integers → 2^32 bits = 540 Mb of memory
    - typical bound 256 elements (set of Char)
  - easy to implement and/or/xor/not
    - just use the corresponding bit operations

Pointers and recursive types

- Recursive types
  - objects contain references to other objects of the same type
  - typically records
    - some data in addition to those references
  - generally used to build linked data structures like lists and trees
- Easy to define with reference variable model
  - everything is a reference anyway
- Value model needs a special *pointer* type
  - value of a pointer = reference to some object
  - restricted to point only to heap objects (Pascal, Modula-3, Ada 83)
    - new pointers created only via memory allocation
  - references to stack objects allowed (C, C++, Ada 95)
    - new pointers also by using ‘address-of’ –operator

Pointers and addresses

- Pointer is a high-level concept
  - a reference to an object
- Address is a low-level concept
  - a location in computer memory
- Pointers *can* be implemented as addresses
  - addresses do not make sense in distributed environments
  - address may be augmented with other information to implement a pointer

Storage reclamation

- How long is the program supposed to run?
  - one short time → just forget
  - long / infinite time → memory leaks are a real problem
- Explicit reclaiming (C, Pascal)
  - programmer's responsibility
  - simplifies implementation
  - dangers
    - we may forget to reclaim unused objects → memory leak
    - we may reclaim used objects → dangling pointers (7.7.2)
- Automatic reclaiming (Java, Ada)
  - garbage collector (7.7.3)
  - how to distinguish garbage from objects?

Pointer assignment

- A := B
  - reference model: A refers to same object as B
  - value model
    - if B is a reference → A refers to B’s object
    - if B is an object → copy contents to A
- Primitive types & reference model
  - inefficient to use pointers
  - number ‘3’ never changes
    - *immutable* types (int, float, char)
    - use the actual object instead of a pointer
  - use pointers only for *mutable* types (e.g., tree node)

Defining recursive data types...

- Reference model languages
  - ML example (Fig. 7.13)
    - tagged tuples
  - Lisp example (Fig. 7.14)
    - everything is a cons-cell or an atom
  - note: data structures of purely functional languages are always acyclic
    - new objects may only point to older ones
    - old ones never change
  - mutually recursive types
    - ML: declare together in a group (p. 386)

...Defining recursive data types

- Value model languages
  - examples (p. 387)
    - forward declarations (Pascal)
    - incomplete declarations (Ada, C)
      - note that in C the type name is ‘struct chr_tree’
    - no ‘aggregates’, structures must be built with programs
  - allocation
    - using built-in functions (Pascal, Ada)
    - using library functions (C)
      - note sizeof & casting
    - using constructors (C++, Java)
      - using new, parameters & overloading

Accessing pointed objects

- Explicit dereferencing
  - Pascal ‘^’, C: ‘*’
- Dereferencing and records
  - recall: recursive data structures are almost always records
  - → justified to provide a special syntax to access fields of pointed records
    - C: r->f
  - Ada: no special notation
    - use pointed records just as standard records
    - implicit dereferencing
    - pseudofield ‘all’ to copy all of the record
- ML language
  - has an imperative part (with side effects)
  - assignment statement allowed but only if l.h.s. is a pointer
  - see example on p. 389

Pointers and arrays in C

- an 1-dimensional array is *almost* same as a pointer to array element
  - see example on p. 389
  - arrays are always passed as pointers to subroutines
- pointer arithmetic
  - add/subtract an integer
  - subtract another pointer
  - compare 2 pointers
  - results are automatically scaled according to the element size
  - common to iterate over arrays using pointers instead of indexes
    - used to be faster
    - ‘more elegant’?
- differences
  - space allocation (and thus the result of sizeof)
  - int *a[n] vs. int a[n][m]

How to read C type declarations?

- (short course)
  - start at the name of the variable
  - loop
    - work right as much as possible (parentheses)
    - work left as much as possible
    - jump out of parentheses
  - until all read
- examples
  - int *a[n]: a is an array of n pointers to int
  - int (*a)[n]: a is a pointer to an array of n ints

Passing array parameters in C

- One-dimensional: pointer to the array
- 2-dimensional, row-pointer layout
  - int *a[] or int **a
- 2-dimensional, contiguous layout
  - int a[][m] or int (*a)[m]
  - the size of the first dimension is irrelevant
  - declaration must contain enough info to compute the sizes of elements
    - int a[][] is not enough (can not compute a+i or a[i])
    - exception: size can be deduced from an aggregate
- 2-dimensional, contiguous layout, sizes not known
  - pass pointer & dimension sizes
  - compute address explicitly with pointer arithmetics (p. 391)

Dangling references

- Created by
  - explicit reclamation (p. 391)
    - dispose, delete (+ destructor)
    - other pointers may still point to the same object
  - references to ‘dead’ stack objects
    - lifetime of reference exceeds the lifetime of the referred object
- Dangers
    - memory area may be allocated to some other object →
    - dangling reference may read or *write* random bits over it
      - data structures are corrupted
      - memory area may even contain heap bookkeeping data

Workarounds

- Algol 68
  - pointer is not allowed to point to an object which has a shorter lifetime than the pointer
    - heap → stack
    - outer subroutine → inner subroutine
  - problem: pointer & object parameters
    - pointers & objects must be augmented with lifetime information
- Ada 95
  - forbids references to objects whose lifetime is briefer than pointer’s *type*
  - can be checked at compile-time in most cases

Tombstones

- Mechanism to catch all dangling references at run-time
  - works both for stack & heap references
  - tombstone = an extra level of indirection between the reference and the object
  - all references point to the tombstone
  - tombstone points to the object
  - should be used for all references (even for global data) to avoid special cases
- Reclamation of an object
  - set tombstone to some special value (non-address)

Cost of tombstones

- Time overhead
  - creation (allocation, &)
  - check validity for each access
    - almost free if hardware catches illegal addresses
    - e.g. outside of program memory area
  - double indirection
- Space overhead
  - significant (almost 1 per each live reference)
  - simple implementation: reclaim objects but leave tombstones (tombstones are usually much smaller)
  - augment with reference counters (reclaim when 0)

Benefits of tombstones

- Dangling references are catched
- Easy to rearrange heap objects
  - all references go through tombstone
  - → only the tombstone reference must be updated
  - rearrangement is necessary when compacting the heap (to eliminate external fragmentation)
- book: not widely used in language implementations, Macintosh OS uses them

Locks and keys

- Alternative to tombstones
- Disadvantages
  - works only for heap objects
  - does not give 100% protection
- Advantages
  - avoids the need of ‘keeping tombstones forever’ (or reclaiming them)

Implementing locks & keys

- Every pointer consists of
  - the actual reference
  - and a key
- Every heap object begins with a lock field
- Access is valid if key = lock (Fig. 7.17)
- Allocation → create a new key/lock value
- Reclaim → set lock to some special value
- Why does it work?
  - even if the memory area is used by some other object, it is very unlikely it has the same value as the key in the dangling reference

Cost of locks & keys

- Space overhead
  - extra word to every pointer & heap object
- Time overhead
  - copying pointers
  - each access involves key/lock –comparison
  - unclear whether cheaper than tombstones
    - tombstone: max 2 indirect accesses (and cache misses)
    - lock & key: 1 indirect access + some arithmetics

Language design

- Most languages
  - do not (by default) generate 'catch dangling reference' code
  - 'debug mode' enables checks
- Pascal
  - programmer can enable dynamic checks
  - → compiler uses locks & keys technique for pointers
- C
  - not even optional checks

Garbage collection

- Automatic reclamation of storage
  - essential in functional/logic languages
    - no ‘stack objects’, everything in heap
  - more and more popular in imperative languages
    - difficult to implement
    - convenience of programming!!!
  - slower than explicit ‘manual’ reclamation
    - but eliminates need to check dangling references

Reference counts

- When is an object X ‘not used’?
  - no pointers to X exist
  - → place a counter to each object = number of pointers referring to this object
- Maintaining reference counts
  - object X creation → X.rc = 1
  - assignment p := q
    - decrement p^.rc (if p <> NIL)
    - increment q^.rc
  - subroutine return
    - pointers deallocated with stack frame
    - → decrement rc of each pointed object
  - hierarchical structures → recursive updates to components

Implementing reference counts

- Implementation
  - must 'know' the location of every pointer
  - → must know which parts contain pointers
    - in stack frames (subroutine return)
    - in heap objects (reclaim → update rc in pointed sub-objects)
  - *type descriptor* contains this information
    - for each distinct type (class)
    - for each subroutine
      - epilogue code uses this to update reference counters
    - e.g. a table containing
      - an offset to each pointer
      - pointer to the type descriptor of each pointer
  - counter = 0 → reclaim object (and update sub-objects)
  - each pointer must be initialized to NIL to prevent the garbage collector from following dangling pointers

Cost of reference counts

- Space
  - extra counter field in every heap object
  - may be significant for small objects (e.g. cons cells)
- Time
  - updating reference counts
  - depends on the 'nature' of the program
- Problem
  - object may be useless even if rc > 0 (Fig. 7.18)
  - caused by circular structures
    - not a problem with non-recursive structures (e.g. strings)
    - not a problem in purely functional languages (no cycles)
- Reference counts may be used with tombstones
  - explicit reclaiming of objects
  - automatic reclaiming of tombstones
  - rc > 0 → programmer has *not* reclaimed the referred object (cyclic or not)

Mark-and-sweep collection

- Better definition of "object X is not used"
  - X can not be reached from valid pointers *outside* the heap
  - covers the situation of Fig. 7.18
- Mark-and-sweep garbage collection
  - mark all heap objects as 'useless'
  - mark all reachable objects as 'useful'
    - begin from stack frames & recurse into structures
    - if a block is already marked 'useful' → return
  - move all 'useless' blocks of heap to free list (reclaim)

Potential problems

- Steps 1 & 3
  - collector must know where every ‘in-use’ heap block begins and ends
  - variable sizes → each block must
    - start with its size
    - contain a free/used indicator
- Step 2
  - collector must know the locations of pointers
  - → place a pointer to object’s type descriptor into each heap block

Cost of 'mark-and-sweep'

- Extra space for heap objects
  - address to type descriptor
    - type descriptor contains the size
  - if type descriptor addresses are word-aligned
  - then last 2 bits of the address can be used for
    - 'free' flag and
    - 'useful' flag
- Step 2
  - needs a recursion stack for the exploration
    - garbage collection is done because we are OUT of space!
  - Schorr & Waite -67: no stack needed
    - redirect pointers to find the way back

Schorr-Waite technique

- Figure 7.19
- Embeds the stack in the fields of heap blocks
  - keep track of current & previous block (Y,R)
- Exploring from Y to W
  - *reverse* the pointer to W to point to R
  - set current block to W, previous to Y
- Returning from W to Y
  - use the reversed pointer in Y to find the previous block R
  - flip reversed pointer back to W
  - set current block to Y, previous to R
- Fact: at most one pointer per block is reversed
  - must be marked somehow → bookkeeping data in block

Storage compaction

- Remove external fragmentation
  - easy with tombstones
- Stop-and-copy technique
  - compaction while eliminating steps 1 and 3 of mark-and-sweep algorithm
  - divide heap into 2 halves (virtual memory!), say H1 & H2
  - all allocations are done in H1
  - memory full → copy all reachable data to H2
    - use ‘useful’ flags to keep track of shared structures
    - not ‘useful’ → pointer points to H1 → copy data to H2, update pointer to H2
    - ‘useful’ → pointer points to H2 → just copy the reference
  - swap H1 & H2

Cost of 'stop-and-copy'

- Only half of the heap is in use
  - not a problem with virtual memory
- Time overhead
  - proportional to the amount of non-garbage blocks
  - mark-and-sweep: all blocks

Mark-and-Sweep vs. RC

- Time usage
  - M-a-S has lower overhead than RC in ‘normal’ operation
    - costs only when a GC is made
  - suffers from “stop-the-world” symptom
    - everything freezes at GC
    - execution happens in bursts
    - the more GC is needed the more it costs (lot of heap data)
- Space usages comparable
  - reversed pointer indicator / reference counter
  - address to type descriptor

Improved M-a-S

- Idea: trade GC accuracy to GC speed
  - divide heap to permanent and dynamic half
  - GC is performed only in the dynamic half
  - data is moved to permanent half if it lives over 1 or 2 GCs
  - like ‘stop-and-copy’ but no swapping
  - risk: permanent area may get full
    - should not happen with ‘normal’ programs
- Avoiding ‘stop-the-world’
  - interleave normal execution & GC
  - multiprocessor computers: P1 executes, P2 does GC

GC and weak typing

- Most GC techniques use type descriptors
  - need to find pointers in objects
- Weakly typed languages & GC?
  - probabilistic approach
    - # of block in the heap << # of possible bit patterns in addresses
    - → probability that a non-pointer data area contains a ‘heap address’ is small
    - → assume that everything that looks like a pointer *is* a pointer & apply standard mark-and-sweep algorithm
  - properties
    - never reclaims useful blocks
      - unless programmer ‘hides’ pointers (possible in C)
    - some useless blocks may get marked as useful
    - compaction impossible: we never know which ‘pointers’ should be changed

Lists

- recursive definition: list is
  - an empty list or
  - a pair consisting of an object and a list
- ‘arrays of functional languages’
  - useful in imperative programs, too
  - can be implemented in any language with records and pointers
- homogeneous in typed languages (ML)
  - Lisp lists are heterogeneous (untyped language)

Implementation

- Chain of blocks (ML)
  - component object *may* be contained in the block
    - useful for primitive types
  - or the block contains a pointer to the component
    - must have some ‘tag bit’ to tell which case holds
- Chain of ‘cons-cells’ (Lisp)
  - combination of 2 pointers

Basic operations

- Convenience notations
  - ML: [a,b,c,d]
  - Lisp: (a b c d)
    - also: (a.(b.(c.(d.nil)))) (dotted pair notation)
    - note: (a.b) is NOT a proper list
- List manipulation
  - construction, extraction, concatenation
  - Lisp
    - car, cdr, cons, append
    - car & cdr (coulder) are 'historical accidents'
    - illegal uses just return nil
  - ML
    - hd, tl, ::, @ (infix notation)
    - illegal uses cause runtime exception

List functions

- Typical built-in functions
  - test for emptiness
  - length
  - *n* th element
  - reversal
- Polymorphic functions
  - filter, map, accumulate
- Haskell (successor of ML)
  - list comprehension =
  - convenience notation for combinations of generation, filtering and mapping
  - much like corresponding mathematical definition of sets

Assignment & equality

- Primitive types
  - obvious semantics & implementation
  - bitwise copying
  - bitwise comparison
- Structured types, abstract data types?
- Example: strings s & t, does s=t mean s & t
  - are aliases?
  - occupy a bitwise identical storage?
    - uninteresting (garbage bits)
  - contain same sequence of characters?
  - would appear the same if printed?

Deep and shallow equality & assignment

- E1 = E2 (in reference model)
  - E1, E2 are the same object = *shallow equality*
  - E1 & E2 refer to objects that are (in some sense) equal = *deep equality*
    - may require recursive testing
- E1 := E2 in reference model
  - suppose E2 refers to object O
  - *shallow assignment*
    - make E1 a reference to O
  - *deep assignment*
    - create a *copy,* say C, of O
    - make E1 a reference to C
- E1 := E2 in value model
  - ‘deep’ for primitive types
  - always shallow for pointers

Language design

- Most languages provide only the ‘shallow’ versions
- Scheme (most well-known Lisp dialect)
  - privides 3 equality testing functions
  - eq?, eqv?, equal?
- Deep assignment is rare
  - Clu: copy1, copy
- Languages with ADTs
  - programmer should carefully think which versions to implement