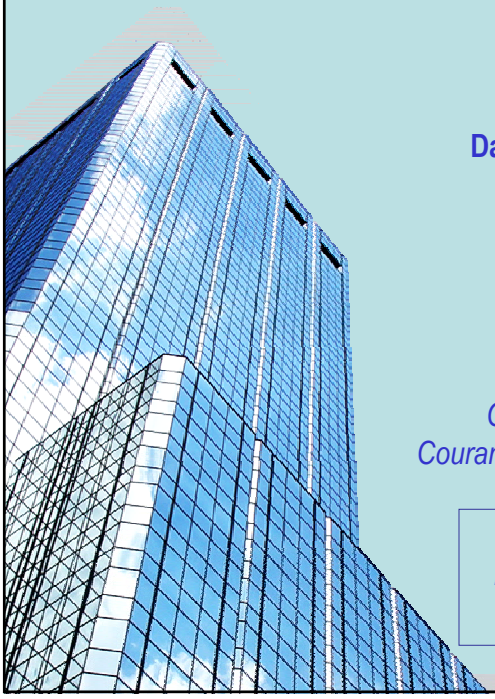


Programming Languages

Session 6 – Main Theme
Data Types and Representation
and
Introduction to ML

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

What is the course about?

- Course description and syllabus:
  - » http://www.nyu.edu/classes/jcf/g22.2110-001
  - » http://www.cs.nyu.edu/courses/fall10/G22.2110-001/index.html
- Textbook:
  - » ***Programming Language Pragmatics (3rd Edition)***

Michael L. Scott
Morgan Kaufmann
ISBN-10: 0-12374-514-4, ISBN-13: 978-0-12374-514-4, (04/06/09)

- Additional References:
  - » Osinski, Lecture notes, Summer 2010
  - » Grimm, Lecture notes, Spring 2010
  - » Gottlieb, Lecture notes, Fall 2009
  - » Barrett, Lecture notes, Fall 2008

Session Agenda

- Session Overview
- Data Types and Representation
- ML Overview
- Conclusion

Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance

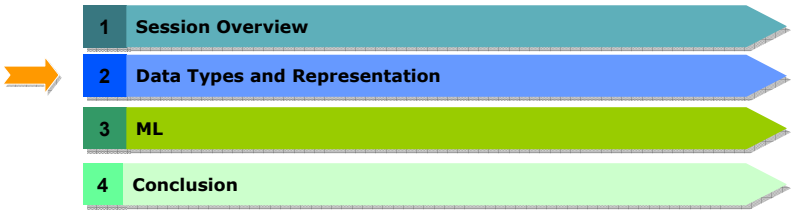
Alignment

Solution Approach

Session 5 Review

- Historical Origins
- Lambda Calculus
- Functional Programming Concepts
- A Review/Overview of Scheme
- Evaluation Order Revisited
- High-Order Functions
- Functional Programming in Perspective
- Conclusions

Agenda



Data Types and Representation

- Data Types
  - » Strong vs. Weak Typing
  - » Static vs. Dynamic Typing
- Type Systems
  - » Type Declarations
- Type Checking
  - » Type Equivalence
  - » Type Inference
  - » Subtypes and Derived Types
- Scalar and Composite Types
  - » Records, Variant Records, Arrays, Strings, Sets
- Pointers and References
  - » Pointers and Recursive Types
- Function Types
- Files and Input / Output

Data Types

- We all have developed an intuitive notion of what types are; what's behind the intuition?
  - » collection (set) of values from a "domain" (the denotational approach)
  - » internal structure of a bunch of data, described down to the level of a small set of fundamental types (the structural approach)
  - » equivalence class of objects (the implementor's approach)
  - » collection of well-defined operations that can be applied to objectsof that type (the abstraction approach)
- The compiler/interpreter defines a mapping of the “values” associated to a type onto the underlying hardware

Data Types – Points of View Summarized

- **Denotational**
  - » type is a set T of values
  - » value has type T if it belongs to the set
  - » object has type T if it is guaranteed to be bound to a value in T
- **Constructive**
  - » type is either built-in (int, real, bool, char, etc.) or
  - » constructed using a type-constructor (record, array, set, etc.)
- **Abstraction-based**
  - » Type is an interface consisting of a set of operations

Data Types

- What are types good for?
  - » implicit context
  - » checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Data Types – Strong vs. Weak Typing

- Strong Typing
  - » has become a popular buzz-word like *structured programming*
  - » informally, it means that the language prevents you from applying an operation to data on which it is not appropriate
  - » more formally, it means that the language does not allow variables to be used in a way inconsistent with their types (no loopholes)
- Weak Typing
  - » Language allows many ways to bypass the type system (e.g., pointer arithmetic)
  - » Trust the programmer vs. not

Data Types – Static vs. Dynamic Typing

- Static Typing
  - » variables have types
  - » compiler can do all the checking of type rules at compile time
  - » ADA, PASCAL, ML
- Dynamic Typing
  - » variables do not have types, values do
  - » Compiler ensures that type rules are obeyed at run time
  - » LISP, SCHEME, SMALLTALK, scripting languages
- A language can have a mixture
  - » e.g., Java has mostly a static type system with some runtime checks
- Pros and Cons:
  - » Static is faster
    - Dynamic requires run-time checks
  - » Dynamic is more flexible, and makes it easier to write code
  - » Static makes it easier to refactor code (easier to understand and maintain code), and facilitates error checking

Data Types – Assigning Types

- Programming languages support various methods for assigning types to program constructs:
  - » determined by syntax: the syntax of a variable determines its type (FORTRAN 77, ALGOL 60, BASIC)
  - » no compile-time bindings: dynamically typed languages
  - » explicit type declarations: most languages

Type Systems

- A type system consists of:
  - a mechanism for defining types and associating them with language constructs
  - a set of rules for:
    - type equivalence: when do two objects have the same type?
    - type compatibility: where can objects of a given type be used?
    - type inference: how do you determine the type of an expression from the types of its parts
- What constructs are types associated with?
  - Constant values
  - Names that can be bound to values
  - Subroutines (sometimes)
  - More complicated expressions built up from the above

Type Systems

- Examples
  - Common Lisp is strongly typed, but not statically typed
  - Ada is statically typed
  - Pascal is almost statically typed
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically

Type Systems – Scalar Types Overview

- discrete types
  - must have clear successor, predecessor
  - Countable
  - One-dimensional
    - integer
    - boolean
    - character
- floating-point types, real
  - typically 64 bit (double in C); sometimes 32 bit as well (float in C)
- rational types
  - used to represent exact fractions (Scheme, Lisp)
- complex
  - Fortran, Scheme, Lisp, C99, C++ (in STL)
  - Examples
    - enumeration
    - subrange

Type Systems – Discrete Types

- integer types
  - often several sizes (e.g., 16 bit, 32 bit, 64 bit)
  - sometimes have signed and unsigned variants (e.g., C/C++, Ada, C#)
  - SML/NJ has a 31-bit integer
- boolean
  - Common type; C had no boolean until C99
- character
  - See next slide
- enumeration types

Type Systems – Other Intrinsic Types

- character, string
  - » some languages have no character data type (e.g., Javascript)
  - » internationalization support
    - Java: UTF-16
    - C++: 8 or 16 bit characters; semantics implementation dependent
  - » string mutability
    - Most languages allow it, Java does not.
- void, unit
  - » Used as return type of procedures;
  - » void: (C, Java) represents the absence of a type
  - » unit: (ML, Haskell) a type with one value: ()

Type Systems – Enumeration Types (Abstraction at its best)

- trivial and compact implementation:
  - » literals are mapped to successive integers
- very common abstraction: list of names, properties
  - » expressive of real-world domain, hides machine representation
  - » Example in Ada:

```
type Suit is (Hearts , Diamonds , Spades , Clubs );
type Direction is (East , West , North , South );
```

  - » Order of list means that Spades > Hearts, etc.
  - » Contrast this with C#:

> "arithmetics on enum numbers may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error"

Type Systems – Enumeration Types and Strong Typing

Ada again:

```
type Fruit is (Apple , Orange , Grape , Apricot );
type Vendor is (Apple , IBM , HP , Dell );
My_PC : Vendor ;
Dessert : Fruit ;
...
My_PC := Apple ;
Dessert := Apple ;
Dessert := My_PC ; -- error
```

Apple is overloaded. It can be of type Fruit or Vendor.

Overloading is allowed in C#, JAVA, ADA

Not allowed in PASCAL, C

Type Systems – Subranges

- Ada and Pascal allow types to be defined which are subranges of existing discrete types

```
type Sub is new Positive range 2 .. 5; -- Ada
V: Sub ;

type sub = 2 .. 5; (* Pascal *)
var v: sub ;
```

- Assignments to these variables are checked at runtime:

```
V := I + J; -- runtime error if not in range
```

Type Systems – Composite Types

- Records
- variants, variant records, unions
- arrays, strings
- classes
- pointers, references
- sets
- Lists
- maps
- function types
- files

Type Systems

- ORTHOGONALITY is a useful goal in the design of a language, particularly its type system
  - » A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined (analogy to vectors)

Type Systems

- For example
  - Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance)
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

Type Checking

- Type checking is the process of ensuring that a program obeys the type system's type compatibility rules.
  - A violation of the rules is called a type clash.
- Languages differ in the way they implement type checking:
  - strong vs weak
  - static vs dynamic
- A TYPE SYSTEM has rules for
  - type equivalence (when are the types of two values the same?)
  - type compatibility (when can a value of type A be used in a context that expects type B?)
  - type inference (what is the type of an expression, given the types of the operands?)

Type Checking – Type Compatibility

- Type compatibility / type equivalence
  - » Compatibility is the more useful concept, because it tells you what you can DO
  - » The terms are often (incorrectly, but we do it too) used interchangeably
  - » Most languages do not require type equivalence in every context.
  - » Instead, the type of a value is required to be compatible with the context in which it is used.
  - » What are some contexts in which type compatibility is relevant?
    - assignment statement type of lhs must be compatible with type of rhs
    - built-in functions like +: operands must be compatible with integer or floating-point types
    - subroutine calls types of actual parameters (including return value) must be compatible with types of formal parameters

Type Checking – Type Compatibility

- Definition of type compatibility varies greatly from language to language.
- Languages like ADA are very strict. Types are compatible if:
  - » they are equivalent
  - » they are both subtypes of a common base type
  - » both are arrays with the same number and types of elements in each dimension
- Other languages, like C and FORTRAN are less strict. They automatically perform a number of type conversions
- An automatic, implicit conversion between types is called type coercion
- If the coercion rules are too liberal, the benefits of static and strong typing may be lost

Type Checking

- Certainly format does not matter:

```
struct { int a, b; }
```

is the same as

```
struct {
    int a, b;
}
```

We certainly want them to be the same as

```
struct {
    int a;
    int b;
}
```


Type Checking – Type Equivalence

- Two major approaches: structural equivalence and name equivalence
  - » Name equivalence is based on declarations
    - Two types are the same only if they have the same name. (Each type definition introduces a new type)
      - strict: aliases (i.e. declaring a type to be equal to another type) are distinct
      - loose: aliases are equivalent
    - Carried to extreme in Ada:
      - "If a type is useful, it deserves to have a name"
  - » Structural equivalence is based on some notion of meaning behind those declarations
    - Two types are equivalent if they have the same structure
  - » Name equivalence is more fashionable these days
  - » Most languages have mixture, e.g., C: name equivalence for records (structs), structural equivalence for almost everything else

Type Checking – Type Equivalence Examples

- Name equivalence in Ada:

```
type t1 is array (1 .. 10) of boolean ;
type t2 is array (1 .. 10) of boolean ;
v1: t1;
v2: t2; -- v1 , v2 have different types
x1 , x2: array (1 .. 10) of boolean ;
-- x1 and x2 have different types too !
```

- Structural equivalence in ML:

```
type t1 = { a: int , b: real };
type t2 = { b: real , a: int };
(* t1 and t2 are equivalent types *)
```


Type Checking – Accidental Structural Equivalence

```
type student = {
    name : string ,
    address : string
}
type school = {
    name : string ,
    address : string
}
type age = float ;
type weight = float ;
```

- With structural equivalence, we can accidentally assign a school to a student, or an age to a weight

Type Checking – Type Conversion

- Sometimes, we want to convert between types:
  - » if types are structurally equivalent, conversion is trivial (even if language uses name equivalence)
  - » if types are different, but share a representation, conversion requires no run-time code
  - » if types are represented differently, conversion may require run-time code (from int to float in C)
- A nonconverting type cast changes the type without running any conversion code. These are dangerous but sometimes necessary in low-level code:
  - » unchecked_conversion in ADA
  - » reinterpret_cast in C++

Type Checking

- There are at least two common variants on name equivalence
  - » The differences between all these approaches boils down to where you draw the line between important and unimportant differences between type descriptions
  - » In all three schemes described in the book, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

Type Checking

- Structural equivalence depends on simple comparison of type descriptions substitute out all names
  - » expand all the way to built-in types
- Original types are equivalent if the expanded type descriptions are the same

Type Checking

- Coercion
  - » When an expression of one type is used in a context where a different type is expected, one normally gets a type error
  - » But what about

```
var a : integer; b, c : real;
    ...
c := a + b;
```

Type Checking

- Coercion
  - Many languages allow things like this, and COERCE an expression to be of the proper type
  - Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
  - Fortran has lots of coercion, all based on operand type

Type Checking

- C has lots of coercion, too, but with simpler rules:
  - all `float`s in expressions become `double`s
  - short `int` and `char` become `int` in expressions
  - if necessary, precision is removed when assigning into LHS

Type Checking

- In effect, coercion rules are a relaxation of type checking
  - » Recent thought is that this is probably a bad idea
  - » Languages such as Modula-2 and Ada do not permit coercions
  - » C++, however, goes hog-wild with them
  - » They're one of the hardest parts of the language to understand

Type Checking

- Make sure you understand the difference between
  - » type conversions (explicit)
  - » type coercions (implicit)
  - » sometimes the word 'cast' is used for conversions (C is guilty here)

Type Checking - Type Coercion

- Coercion in C
  - The following types can be freely mixed in C:
    - char
    - (unsigned) (short, long) int
    - float, double
  - Recent trends in type coercion:
    - static typing: stronger type system, less type coercion
    - user-defined: C++ allows user-defined type coercion rules

Type Checking - Polymorphism

- Polymorphism allows a single piece of code to work with objects of multiple types:
  - Subclass polymorphism:
    - The ability to treat a class as one of its superclasses
    - The basis of OOP
    - Class polymorphism: the ability to treat a class as one of its superclasses (special case of subtype polymorphism)
  - Subtype polymorphism:
    - The ability to treat a value of a subtype as a value of a supertype
    - Related to subclass polymorphism
  - Parametric polymorphism:
    - The ability to treat any type uniformly
      - types can be thought of as additional parameters
        - implicit: often used with dynamic typing: code is typeless, types checked at run-time (LISP, SCHEME) - can also be used with static typing (ML)
        - explicit: templates in C++, generics in JAVA
    - Found in ML, Haskell, and, in a very different form, in C++ templates and Java generics
  - Ad hoc polymorphism:
    - Multiple definitions of a function with the same name, each for a different set of argument types (overloading)

Type Checking – Parametric Polymorphism Examples

- SCHEME

```
(define (length l)
        (cond
                ((null? l) 0)
                (#t (+ (length (cdr l)) 1))))
```

The types are checked at run-time

- ML

```
fun length xs =
        if null xs
        then 0
        else 1 + length (tl xs)
```

length returns an int, and can take a list of any element type, because we don't care what the element type is. The type of this function is written 'a list -> int

How can ML be statically typed and allow polymorphism?
It uses type variables for the unknown types. The type of this function is written 'a list -> int.

Type Checking - Subtyping

- A relation between types; similar to but not the same as subclassing
- Can be used in two different ways:
  - » Subtype polymorphism
  - » Coercion
- Subtype examples:
  - » A record type containing fields a, b and c can be considered a subtype of one containing only a and c
  - » A variant record type consisting of fields a or c can be considered a subtype of one containing a or b or c
  - » The subrange 1..100 can be considered a subtype of the subrange 1..500.

Type Checking – Subtype Polymorphisms and Coercion

- subtype polymorphism:
  - » ability to treat a value of a subtype as a value of a supertype
- coercion:
  - » ability to convert a value of a subtype to a value of
- Example:
  - » Let's say type s is a subtype of r.

```
var vs: s;
var vr: r;
```

  - » Subtype polymorphism:

```
function [t  r] f (x: t): t { return x; }
f(vr ); // returns a value of type r
f(vs ); // returns a value of type s
```

  - » Coercion:

```
function f (x: r): r { return x; }
f(vr ); // returns a value of type r
f(vs ); // returns a value of type r
```


Type Checking – Overloading and Coercion

Overloading: Multiple definitions for a name, distinguished by their types

Overload resolution: Process of determining which definition is meant in a given use

- » Usually restricted to functions
- » Usually only for static type systems
- » Related to coercion. Coercion can be simulated by overloading (but at a high cost). If type a has subtypes b and c, we can define three overloaded functions, one for each type. Simulation not practical for many subtypes or number of arguments

Overload resolution based on:

- » number of arguments (Erlang)
- » argument types (C++, Java)
- » return type (Ada)

Type Checking – Overloading and Coercion

- What's wrong with this C++ code?

```
void f(int x);
void f(string *ps);
f(NULL);
```

- Depending on how NULL is defined, this will either call the first function (if NULL is defined as 0) or give a compile error (if NULL is defined as ((void*)0)).
  - » This is probably not what you want to happen, and there is no easy way to fix it. This is an example of ambiguity resulting from coercion combined with overloading
- There are other ways to generate ambiguity:

```
void f(int);
void f(char);
double d = 6.02;
f(d);
```


Type Checking - Constness

- Ability to declare that a variable will not be changed:
  - » C/C++: const
  - » Java: final
- May or may not affect type system: C++: yes, Java: no

Type Checking – Type Inference

- Type checking:
  - » Variables are declared with their type
  - » Compiler determines if variables are used in accordance with their type declarations
- Type inference: (ML, Haskell)
  - » Variables are declared, but not their type
  - » Compiler determines type of a variable from its initialization/usage
- In both cases, type inconsistencies are reported at compile time

```
fun f x =
    if x = 5 (* There are two type errors here *)
    then hd x
    else tl x
```


Type Checking – Type Inference

- How do you determine the type of an arbitrary expression?
- Most of the time it's easy:
  - » the result of built-in operators (i.e. arithmetic) usually have the same type as their operands
  - » the result of a comparison is Boolean
  - » the result of a function call is the return type declared for that function
  - » an assignment has the same type as its left-hand side
- Some cases are not so easy:
  - » operations on subranges
    - Consider this code:

      ```
      type Atype = 0..20;
      Btype = 10..20;
      var a : Atype;
      b : Btype;
      ```

    - What is the type of a + b?
      - – Cheap and easy answer: base type of subrange, integer in this case
      - – More sophisticated: use bounds analysis to get 10..40
    - What if we assign to a an arbitrary integer expression?
      - – Bounds analysis might reveal it's OK (i.e. (a + b) / 2)
      - – However, in many cases, a run-time check will be required
      - – Assigning to some composite types (arrays, sets) may require similar run-time checks
  - » operations on composite types

Scalar and Composite Types – Records and Variant Records

- Records
  - A record consists of a set of typed fields.
  - Choices:
    - Name or structural equivalence? Most statically typed languages choose name equivalence
    - ML, Haskell are exceptions
  - Nested records allowed?
    - Usually, yes. In FORTRAN and LISP, records but not record declarations can be nested
  - Does order of fields matter?
    - Typically, yes, but not in ML
  - Any subtyping relationship with other record types?
    - Most statically typed languages say no
    - Dynamically typed languages implicitly say yes
    - This is know as duck typing

      "if it walks like a duck and quacks like a duck, I would call it a duck"

      -James Whitcomb Riley

Scalar and Composite Types – Records and Variant Records

- Records (Structures)
  - usually laid out contiguously
  - possible holes for alignment reasons
  - smart compilers may re-arrange 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

Scalar and Composite Types – Records Syntax

- PASCAL:

```
type element = record
   name : array[1..2] of char;
   atomic_number : integer;
   atomic_weight : real;
end;
```

- C:

```
struct element {
   char name[2];
   int atomic_number;
   double atomic_weight;
};
```

- ML:

```
type element = {
   name: string,
   atomic_number: int,
   atomic_weight: real
};
```


Scalar and Composite Types – Records and Variant Records

- Unions (Variant Records)
  - A variant record is a record that provides multiple alternative sets of fields, only one of which is valid at any given time
    - Also known as a discriminated union
  - Each set of fields is known as a variant.
  - Because only one variant is in use at a time, the variants can share / overlay space
    - causes problems for type checking
  - In some languages (e.g. ADA, PASCAL) a separate field of the record keeps track of which variant is valid.
  - In this case, the record is called a discriminated union and the field tracking the variant is called the tag or discriminant.
  - Without such a tag, the variant record is called a nondiscriminated union.
- 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

Scalar and Composite Types – Nondiscriminated Unions

- Nondiscriminated or free unions can be used to bypass the type model:

```
union value {
    char *s;
    int i; // s and i allocated at same address
};
```

- Keeping track of current type is programmer's responsibility.
  - » Can use an explicit tag if desired:

```
struct entry {
    int discr;
    union { // anonymous component, either s or i.
            char *s; // if discr = 0
            int i; // if discr = 1, but system won't check
    };
};
```

Note: no language support for safe use of variant!

Scalar and Composite Types – Records Memory Layout

- The order and layout of record fields in memory are tied to implementation trade-offs:
  - » Alignment of fields on memory word boundaries makes access faster, but may introduce holes that waste space
  - » If holes are forced to contain zeroes, comparison of records is easier, but zeroing out holes requires extra code to be executed when the record is created
  - » Changing the order of fields may result in better performance, but predictable order is necessary for some systems code

Scalar and Composite Types – Records and Variant Records

- Memory layout and its impact (structures)

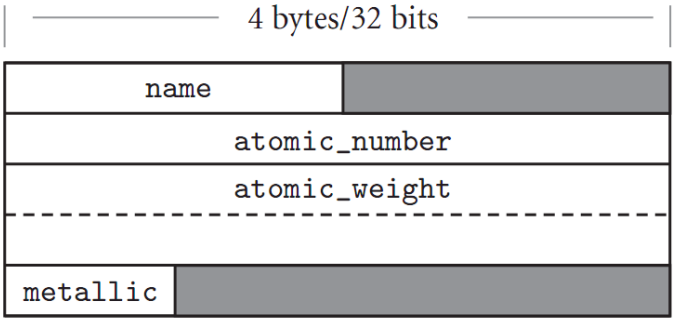


Figure 7.1 Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded "holes."

Scalar and Composite Types – Records and Variant Records

- Memory layout and its impact (structures)

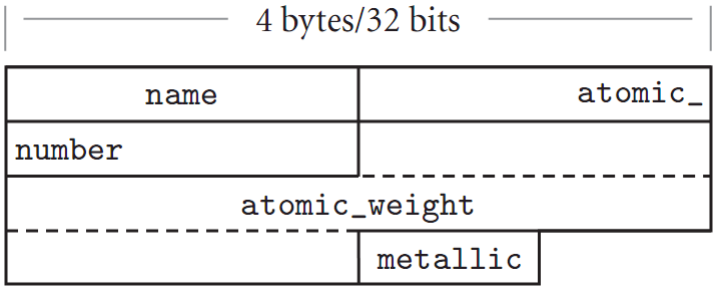


Figure 7.2 Likely memory layout for packed element records. The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

Scalar and Composite Types – Records and Variant Records

- Memory layout and its impact (structures)

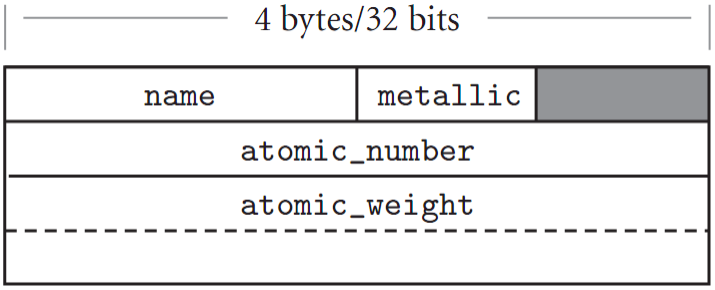


Figure 7.3 Rearranging record fields to minimize holes. By sor ting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

Scalar and Composite Types – Records and Variant Records

- Memory layout and its impact (unions)

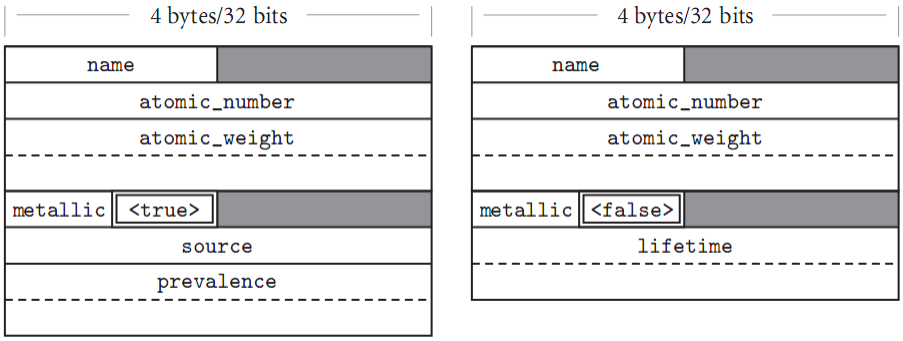


Figure 7.15 (CD) Likely memory layouts for element variants. The value of the naturally occurring field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type string_ptr is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.

Scalar and Composite Types –Variant Records in Ada

- Need to treat group of related representations as a single type:

```
type Figure_Kind is (Circle , Square , Line );
type Figure ( Kind : Figure_Kind ) is record
    Color : Color_Type ;
    Visible : Boolean ;
    case Kind is
            when Line => Length : Integer ;
                         Orientation : Float ;
                         Start : Point ;
            when Square => Lower_Left ,
                           Upper_Right : Point ;
            when Circle => Radius : Integer ;
                           Center : Point ;
    end case ;
end record ;
```


Scalar and Composite Types – Discriminant Checking – Part 1

```
C1: Figure ( Circle ); -- discriminant provides constraint
S1: Figure ( Square );
...
C1. Radius := 15;
if S1. Lower_Left = C1. Center then ...
function Area (F: Figure ) return Float is
    -- applies to any figure , i.e., subtype
begin
    case F. Kind is
        when Circle => return Pi * Radius ** 2;
    ...
end Area
```

Scalar and Composite Types – Discriminant Checking – Part 2

```
L : Figure ( Line );
F : Figure ; -- illegal , don 't know which kind
P1 := Point ;
...
C := ( Circle , Red , False , 10, P1 );
    -- record aggregate
... C. Orientation ...
    -- illegal , circles have no orientation
C := L;
    -- illegal , different kinds
C. Kind := Square ;
    -- illegal , discriminant is constant
```

- Discriminant is a visible constant component of object.

Scalar and Composite Types – Variants and Classes

- discriminated types and classes have overlapping functionalities
- discriminated types can be allocated statically
- run-time code uses less indirection
- compiler can enforce consistent use of discriminants
- adding new variants is disruptive; must modify every case statement
- variant programming: one procedure at a time
- class programming: one class at a time

Scalar and Composite Types – Free Unions

- Free unions can be used to bypass the type model:

```
union value {
  char *s;
  int i; // s and i allocated at same address
};
```

- Keeping track of current type is programmer's responsibility.
  - » Can use an explicit tag:

```
struct entry {
   int discr ;
   union { // anonymous component , either s or i.
            char *s; // if discr = 0
            int i; // if discr = 1, but system won 't check
   };
};
```


Scalar and Composite Types – Discriminated Unions/Dynamic Typing

- In dynamically-typed languages, only values have types, not names.

```
S = 13.45 # a floating - point number
...
S = [1 ,2 ,3 ,4] # now it 's a list
```

- Run-time values are described by discriminated unions.
  - » Discriminant denotes type of value.

```
S = X + Y # arithmetic or concatenation
```

Scalar and Composite Types – 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 7.4)

Scalar and Composite Types – Arrays

- index types
  - most languages restrict to an integral type
  - Ada, Pascal, Haskell allow any scalar type
- index bounds
  - many languages restrict lower bound:
  - C, Java: 0, Fortran: 1, Ada, Pascal: no restriction
- when is length determined
  - Fortran: compile time; most other languages: can choose
- dimensions
  - some languages have multi-dimensional arrays (Fortran, C)
  - many simulate multi-dimensional arrays as arrays of arrays (Java)
- literals
  - C/C++ has initializers, but not full-fledged literals
  - Ada: (23, 76, 14) Scheme: #(23, 76, 14)
- first-classness
  - C, C++ does not allow arrays to be returned from functions
- a slice or section is a rectangular portion of an array
  - Some languages (e.g. FORTRAN, PERL, PYTHON, APL) have a rich set of array operations for creating and manipulating sections.

Scalar and Composite Types – Array Literals

- ADA: (23, 76, 14)
- SCHEME: #(23, 76, 14)
- C and C++ have initializers, but not full-fledged literals:

```
int v2[] = { 1, 2, 3, 4 }; //size from initializer
char v3[2] = { 'a', 'z'}; //declared size
int v5[10] = { -1 }; //default: other components = 0
struct School r =
   { "NYU", 10012 }; //record initializer
char name[] = "Scott"; //string literal
```


Scalar and Composite Types - Arrays

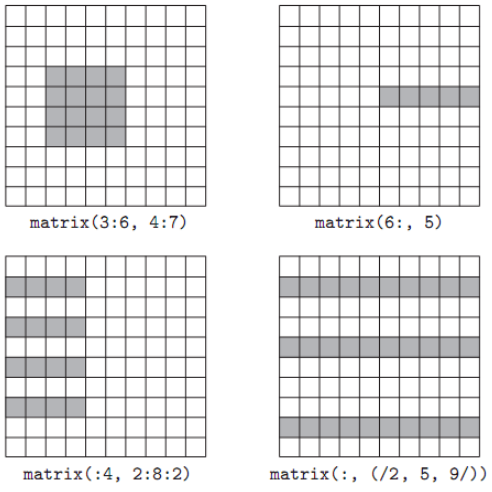


Figure 7.4 Array slices(sections) in Fortran90. Much like the values in the header of an enumeration-controlled loop (Section6.5.1), a: b: c in a subscript indicates positions a, a+c, a+2c, ...through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. 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.

Scalar and Composite Types – Arrays Shapes

- Dimensions, Bounds, and Allocation
- The shape of an array consists of the number of dimensions and the bounds of each dimension in the array.
- The time at which the shape of an array is bound has an impact on how the array is stored in memory:
  - » ***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 run/elaboration time*** - variable-size part of local stack frame
  - » ***arbitrary lifetime, shape bound at runtime*** - allocate from heap or reference to existing array
  - » ***arbitrary lifetime, dynamic shape*** - also known as dynamic arrays, must allocate (and potentially reallocate) in heap

Scalar and Composite Types - 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];
    ...
}
```

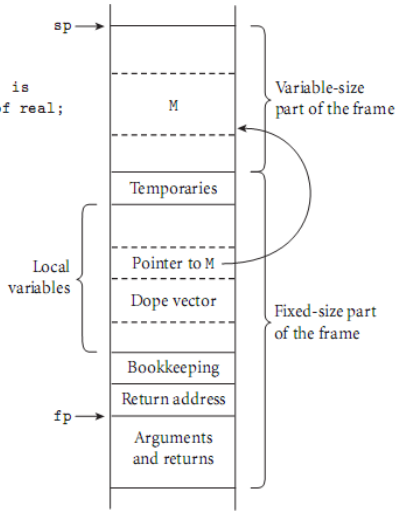


Figure 7.6 Elaboration-time allocation of arrays in Ada or C99.

Scalar and Composite Types – Arrays Memory Layout

- Two-dimensional arrays
  - » Row-major layout: Each row of array is in a contiguous chunk of memory
  - » Column-major layout: Each column of array is in a contiguous chunk of memory
  - » Row-pointer layout: An array of pointers to rows lying anywhere in memory
- If an array is traversed differently from how it is laid out, this can dramatically affect performance (primarily because of cache misses)
- A dope vector contains the dimension, bounds, and size information for an array. Dynamic arrays require that the dope vector be held in memory during run-time
- Contiguous elements (see Figure 7.7)
  - » 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]

Scalar and Composite Types - Arrays

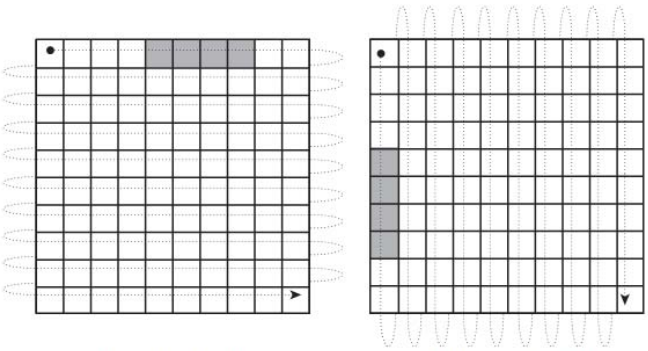


Figure7.7 Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte 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.

Scalar and Composite Types - Arrays

- Two layout strategies for arrays (Figure 7.8):
  - » 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

Scalar and Composite Types - 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 */
```

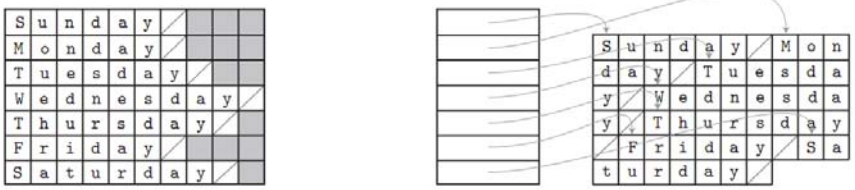


Figure 7.8 Contiguous array allocation v. row pointers in C. The declaration on the left is a tr ue two-dimensional array. The slashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is a ragged array of pointers to arrays of character s. In both cases, we have omitted bounds 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.

- **Example**: 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
```

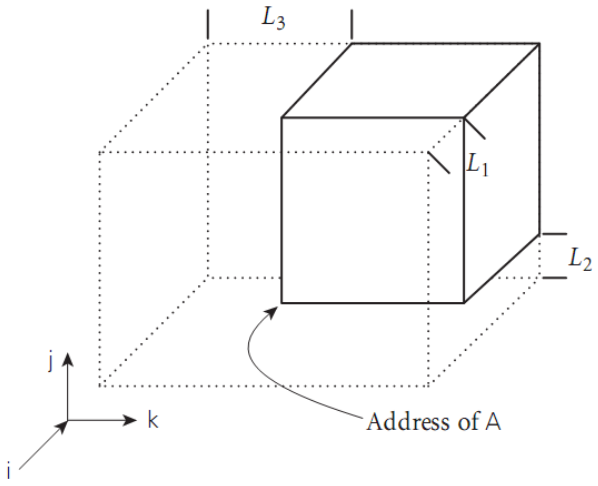



Figure 7.9 Virtual location of an array with nonzero lower bounds. By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory, but whose lower bounds are all zero.

Scalar and Composite Types - Arrays

- **Example** (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

Scalar and Composite Types - Composite Literals

- Does the language support these?
  - » array aggregates

    A := (1, 2, 3, 10); -- *positional*

    A := (1, others => 0); -- *for default*

    A := (1..3 => 1, 4 => -999); -- *named*
  - » record aggregates

    R := (name => "NYU ", zipcode => 10012);

Scalar and Composite Types - Strings

- Strings are really just arrays of characters
- 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 important) non-circular

Scalar and Composite Types - 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

Scalar and Composite Types – Initializers in C++

- Similar notion for declarations:

```
int v2[] = { 1, 2, 3, 4 }; // size from initializer
char v3[2] = { 'a', 'z'}; // declared size
int v5[10] = { -1 }; // default: other components = 0
struct School r =
    { "NYU", 10012 }; // record initializer
char name[] = "Algol"; // string literals are aggregates
```

- C has no array assignments, so initializer is not an expression (less orthogonal)

Pointers and Recursive Types - Pointers

- Pointers serve two purposes:
  - » efficient (and sometimes intuitive) access to elaborated objects (as in C)
  - » dynamic creation of linked data structures, in conjunction with a heap storage manager
- Several languages (e.g. Pascal) restrict pointers to accessing things in the heap
- Pointers are used with a value model of variables
  - » They aren't needed with a reference model

Pointers and Recursive Types – Pointers and References

- Related (but distinct) notions:
  - » a value that denotes a memory location
    - value model pointer has a value that denotes a memory location (C, PASCAL, ADA)
  - » a dynamic name that can designate different objects
    - names have dynamic bindings to objects, pointer is implicit (ML, LISP, SCHEME)
  - » a mechanism to separate stack and heap allocation

    type Ptr is access Integer ; -- *Ada : named type*

    typedef int * ptr ; *// C, C++*
  - » JAVA uses value model for built-in (scalar) types, reference model for user-defined types

Pointers and Recursive Types – Pointers and Dereferencing

- Need notation to distinguish pointer from designated object
  - » in Ada: Ptr vs Ptr.all
  - » in C: ptr vs *ptr
  - » in Java: no notion of pointer
- For pointers to composite values, dereference can be implicit:
  - » in Ada: C1.Value equivalent to C1.all.Value
  - » in C/C++: c1.value and c1->value are different

Pointers and Recursive Types - Pointers

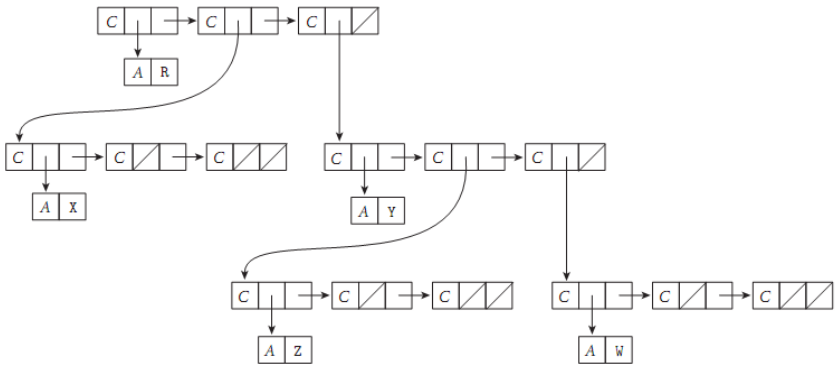


Figure 7.11 Implementation of a tree in Lisp. A diagonal slash through a box indicates a null pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

Pointers and Recursive Types - Pointers

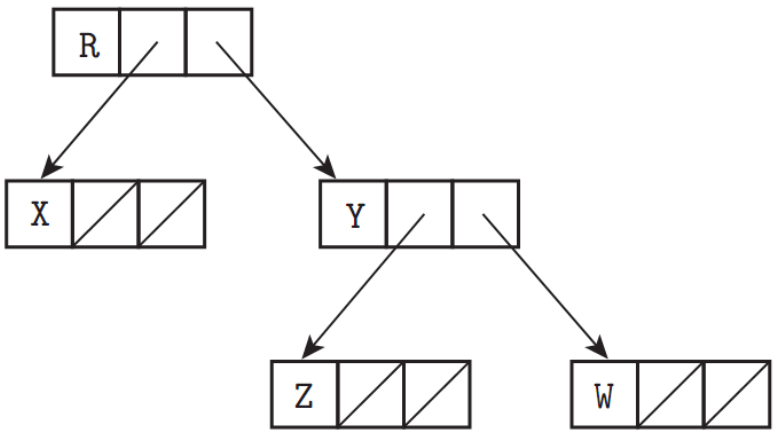


Figure 7.12 Typical implementation of a tree in a language with explicit pointers. As in Figure 7.11, a diagonal slash through a box indicates a null pointer.

Pointers and Recursive Types – Extra Pointer Capabilities

- Questions:
  - » Is it possible to get the address of a variable?
  - » Convenient, but aliasing causes optimization difficulties (the same way that pass by reference does)
  - » Unsafe if we can get the address of a stack allocated variable.
- Is pointer arithmetic allowed?
  - » Unsafe if unrestricted.
  - » In C, no bounds checking:

```
// allocate space for 10 ints
int *p = malloc (10 * sizeof (int ));
p += 42;
... *p ... // out of bounds , but no check
```


Pointers and Recursive Types - Pointers

- 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 - Pointers

- 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 – "Generic" Pointers

- A pointer used for low-level memory manipulation, i.e., a memory address.
  - In C, void is requisitioned to indicate this.
  - Any pointer type can be converted to a void *.

```
int a [10];
void *p = &a [5];
```

  - A cast is required to convert back:

```
int *pi = (int *)p; // no checks
double *pd = ( double *)p;
```

Pointers and Recursive Types – "Generic" Reference Types

- An object of generic reference type can be assigned an object of any reference type.

  void * in C and C++

  Object in JAVA
- How do you go back to a more specific reference type from a generic reference type?
  - » Use a type cast, i.e., down-cast
  - » Some languages include a tag indicating the type of an object as part of the object representation (JAVA, C#, MODULA-3, C++), hence the down-cast can perform a dynamic type check
  - » Others (such as C) simply have to settle for unchecked type conversions, i.e., trust the programmer not to get lost

Pointers and Recursive Types – Pointers and Arrays in C/C++

- In C/C++, the notions:
  - » an array
  - » a pointer to the first element of an array

  are almost the same – It is easy to get lost!

```
void f ( int *p) { ... }
int a [10];
f(a); // same as f(&a [0])
int *p = new int [4];
... p[0] ... // first element
... *p ... // ditto
... 0[p] ... // ditto
... p [10] ... // past the end ; undetected error
```

Pointers and Recursive Types – Pointers and Safety

- Pointers create aliases: accessing the value through one name affects retrieval through the other:

```
int *p1 , *p2;
...
p1 = new int [10]; // allocate
p2 = p1; // share
delete [] p1; // discard storage
p2 [5] = ... // error :
// p2 does not denote anything
```


Pointers and Recursive Types – Pointer Troubles

- Several possible problems with low-level pointer manipulation:
  - » dangling references
  - » garbage (forgetting to free memory)
  - » freeing dynamically allocated memory twice
  - » freeing memory that was not dynamically allocated
  - » reading/writing outside object pointed to

Pointers and Recursive Types - Pointers

- 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 – Dangling References

- If we can point to local storage, we can create a reference to an undefined value:

```
int *f () { // returns a pointer to an integer
int local ; // variable on stack frame of f
...
return & local ; // pointer to local entity
}
int *x = f ();
...
*x = 5; // stack may have been overwritten
```

Pointers and Recursive Types - Pointers

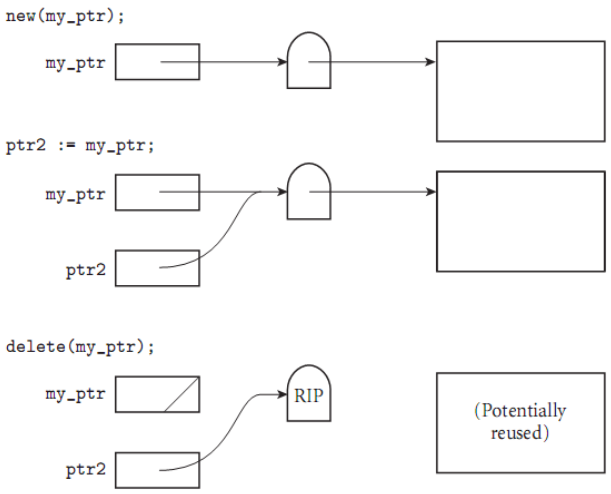


Figure 7.17 (CD) Tombstones. A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

Pointers and Recursive Types - Pointers

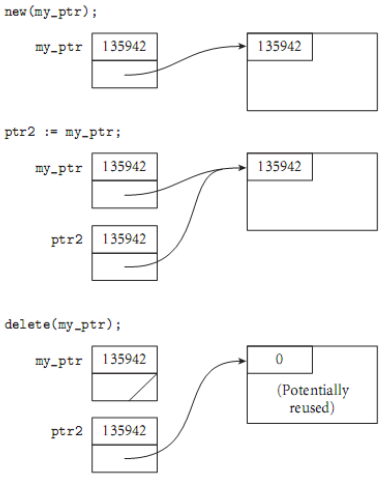


Figure 7.18 (CD) Locks and Keys. A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

Pointers and Recursive Types - Pointers

- 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 - Pointers

- Garbage collection with reference counts

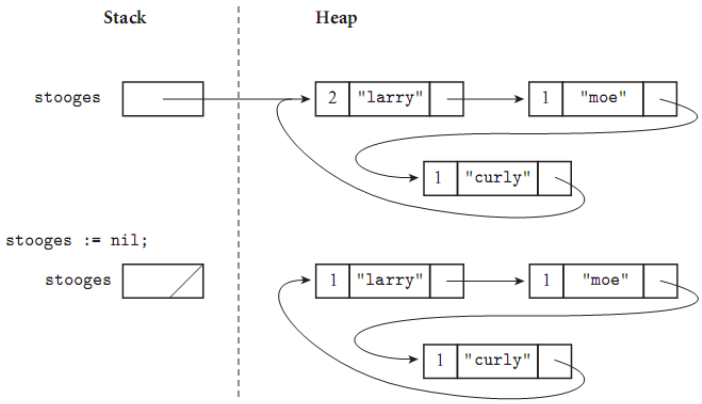


Figure 7.13 Reference counts and circular lists. The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

Pointers and Recursive Types - Pointers

- Mark-and-sweep
  - » commonplace in Lisp dialects
  - » complicated in languages with rich type structure, but possible if language is strongly typed
  - » achieved successfully in Cedar, Ada, Java, Modula-3, ML
  - » 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 - Pointers

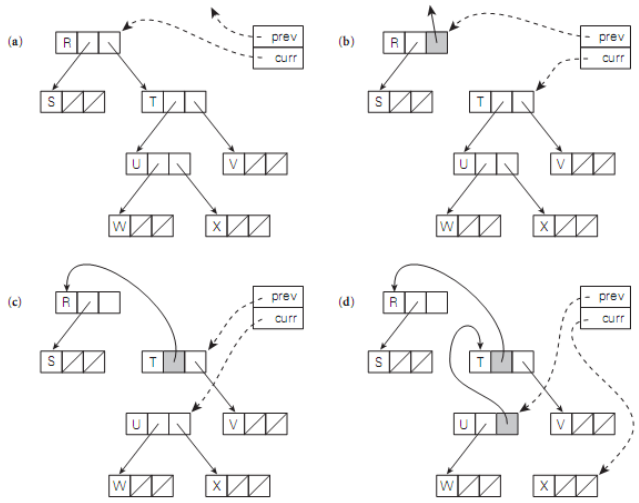


Figure 7.14 Heap exploration via pointer reversal.

Pointers and Recursive Types – Lists, Sets, and Maps

- Recursive Types
  - » list: ordered collection of elements
  - » set: collection of elements with fast searching
  - » map: collection of (key, value) pairs with fast key lookup
- Low-level languages typically do not provide these. High-level and scripting
  - » languages do, some as part of a library.
    - Perl, Python: built-in, lists and arrays merged.
    - C, Fortran, Cobol: no
    - C++: part of STL: list<T>, set<T>, map<K,V>
    - Java: yes, in library
    - Setl: built-in
    - ML, Haskell: lists built-in, set, map part of library
    - Scheme: lists built-in
    - Pascal: built-in sets
      - – but only for discrete types with few elements, e.g., 32

Pointers and Recursive Types - 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

Pointers and Recursive Types – Dynamic Data Structures

```
type Cell ; -- an incomplete type
type Ptr is access Cell ; -- an access to it
type Cell is record -- the full declaration
Value : Integer ;
Next , Prev : Ptr ;
end record ;
List : Ptr := new Cell '(10 , null , null );
... -- A list is just a pointer to its first element
List . Next := new Cell '(15 , null , null );
List . Next . Prev := List ;
```


Pointers and Recursive Types – Incomplete Declarations in C++

```
struct cell {
    int value ;
    cell * prev ; // legal to mention name
    cell * next ; // before end of declaration
};
struct list ; // incomplete declaration
struct link {
    link * succ ; // pointers to the
    list * memberOf ; // incomplete type
};
struct list { // full definition
    link * head ; // mutually recursive references
};
```

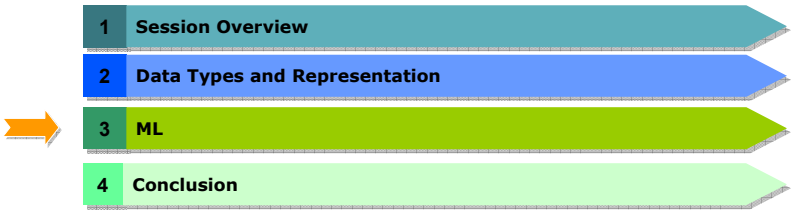
Function Types

- not needed unless the language allows functions to be passed as arguments or returned
- variable number of arguments:
  - » C/C++: allowed, type system loophole, Java: allowed, but no loophole
- optional arguments: normally not part of the type.
- missing arguments in call: in dynamically typed languages, typically OK.

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*

Agenda



What's wrong with Imperative Languages?

- State
  - » Introduces context sensitivity
  - » Harder to reuse functions in different context
  - » Easy to develop inconsistent state

```
int balance = account.getBalance;
balance += deposit;
// Now there are two different values stored in two different places
```

- Sequence of function calls may change behavior of a function
  - Oh, didn't you know you have to call C.init() before you…
  - » Lack of Referential Transparency
- These issues can make imperative programs hard to understand

What is functional programming?

- A style of programming that avoids the use of assignments
  - » Similar to the way structured programming avoided the use of goto statements
- No Assignment, No variables
  - » val a = 3; -- a named constant, initialized to 3
- State changes only in predictable ways
  - » Bindings may be created and destroyed
  - » Values associated with bindings don't change
- Referential Transparency
  - » Easier to understand programs
  - » Easier to reuse parts of programs

Some Sums

x is a vector of integers

Imperative

- Describes *how* to calculate result

```
Iterator it = x.iterator();
      int result = 0;
      while(it.hasNext()) {
        result += it.next();
      }
```

Functional

- Defines what the result is

```
function sum []  = 0
        | sum (x::xs) = x + (sum xs)

+/x
```

History

- Developed at Edinburgh University (Scotland) by Robin Milner & others in the late 1970's
- A **M**eta **L**anguage for theorem proving
- SML is Standard ML – widely used (except in France, where they use CAML)

SML Implementations

- Standard ML of New Jersey can be found on the SMLNJ web site, www.smlnj.org
- Poly/ML is at www.polyml.org

ML: a quasi-functional language with strong typing

- Conventional syntax:

  > val x = 5; (*user input *)
  val x = 5: int (*system response*)

  > fun len lis = if (null lis) then 0 else 1 + len (tl lis); val len = fn : 'a list -> int

- Type inference for local entities
  > x * x * x;
  val it = 125: int (* it denotes the last computation*)

ML's Imperative Features

- Reference types: val p = ref 5
  - » Dereferencing syntax: !p + 1
  - » Changes state
- Statements
  - » if E then E_1 else E_2 – an expression
  - » while E do E_1 – iteration implies state change
  - » Each E_1 E_2 may change state
- *Avoid these*

ML Overview

- originally developed for use in writing theorem provers
- functional: functions are first-class values
- garbage collection
- strict
- strong and static typing; powerful type system
  - » parametric polymorphism
  - » structural equivalence
  - » all with type inference!
- advanced module system
- exceptions
- miscellaneous features:
  - » datatypes (merge of enumerated literals and variant records)
  - » pattern matching
  - » ref type constructor (like “const pointers” (“not pointers to const”))

Sample SML / NJ Interactive Session

```
- val k = 5;                          user input
val k = 5 : int                       system response
- k * k * k;
val it = 125 : int                    'it' denotes the last
                                      computation
- [1, 2, 3];
val it = [1,2,3] : int list
- ["hello", "world"];
val it = ["hello","world"] : string list
- 1 :: [ 2, 3 ];
val it = [1,2,3] : int list - [ 1, "hello"]; error
```

Tuples

- Ordered lists of elements
- Denoted by comma separated list enclosed in parenthesis
  - » (a, b) is a two-tuple, or *pair* of type int * int
  - » (1, 2, 3) is a 3-tuple of type int * int * int
- Elements may be of any type, including other tuples

```
> (“hi”, 1.0, 2, (0, 0));
val it : string * real * int * (int * int)
```


Records

- Records are tuples in which the components – called *fields* – are named
- Records are denoted by a comma separated list of name value bindings enclosed in curly braces

  {name = “Jones”, age = 25, salary = 65000}

- We can define an abstract record type:

```
> type emp = {name : string, age : int, sal : int};
type emp
> fun getSal (e : emp) = #sal e;
val getSal = fn : emp -> int
```

Lists

- A list is a sequence of elements, all of which have the same type
- Lists are denoted by a sequence of comma separated elements enclosed in square brackets:

```
> [1, 2, 3, 4]
val it = [1, 2, 3, 4] : int list
```

- Similar to LISP, with conventional syntax:
  hd, tl, :: instead of car, cdr, cons for head, tail and concatenate element

```
> fun append (x, y) = if null (x) then y
                      else hd (x) :: append (tl (x), y);
  val append = fn: 'a list * 'a list -> 'a list
```

- **(* a function that takes a pair of lists and yields a list *)**
- **'a is a type variable**

Operations on Lists

- null [1, 2];

val it = false : bool

- null [ ];

val it = true : bool

- hd [1, 2, 3];

val it = 1 : int

- tl [1, 2, 3];

val it = [2, 3] : int list

- [ ];

val it = [ ] : 'a list this list is polymorphic

Patterns

```
fun append (x, y) = if null (x) then y
    else hd (x) :: append (tl (x), y);
```

- Now, with patterns

```
> fun append ([], y) = y
  |  append (x::xs, y) = x::append(xs,y);
val append = fn : 'a list * 'a list -> 'a list
```

- Clearly expresses basis and recursive parts of a recursive definition of append

```
append ([1,2,3],[4,5,6]);
    val it = [1, 2, 3, 4, 5, 6] : int list
```


Patterns help replace Imperative statements in Functional Programs

- Selection

```
if x = 0 then y = 1 else y = 2
fun   fif(0) = 1
  |  fif(-) = 2;
```

- Loops generally handled by (tail) recursion

```
fun sum [] = 0
     sum (x::xs) = x + (sum xs)
```

Currying: partial bindings

- Curried functions take one argument

a b c means ((a b) c) (* parentheses are lisp notation*)
a is a function
(a b) yields another function that is applied to c

```
> fun add x y = x + y;
 val add = fn : int -> int -> int
> add 2;
val it = fn : int -> int
 > it 3;
 val it = 5 : int
```

- Keep in mind:
  - » add 2 2 (* Curried function *)
  - » add (2, 2) (* function takes one tuple *)

Even Better than Sums

```
fun reduce f i [] = i
  | reduce f i (x::xs) =  f x (reduce f i xs);

fun add a b = a + b

> reduce add 0 [2, 3, 4];
val it =  9 : int

fun times a b = a * b

> reduce times 1 [2, 3, 4];
val it =  24 : int

fun timesReduce x = reduce times 0
> timesReduce [2, 3, 4];
val it =  24 : int
```

Simple Functions

- A function declaration:

  - fun abs x = if x >= 0.0 then x else –x

  val abs = fn : real -> real

- A function expression:

  - fn x => if x >= 0.0 then x else -x

  val it = fn : real -> real

Functions, II

```
- fun length xs =
    if null xs
    then 0
    else 1 + length (tl xs );
val length = fn : 'a list -> int
```

'a denotes a type variable; length can be applied to lists of any element type

The same function, written in pattern-matching style:

```
- fun length [] = 0
    | length (x:: xs) = 1 + length xs
val length = fn : 'a list -> int
```

Type inference

```
> fun incr x = x + 1;
val incr = fn : int -> int
```

- because of its appearance in (x+1), x must be integer

```
> fun add2 x = incr (incr x);
val add2 = fn : int -> int
```

- incr returns an integer, so add2 does as well
- x is argument of incr, so must be integer

```
val wrong = 10.5 + incr 7;
```

Error: operator and operand don't agree

Type Inference and Polymorphism

- Advantages of type inference and polymorphism:
  - » frees you from having to write types.
  - » A type can be more complex than the expression whose type it is, e.g., flip
- with type inference, you get polymorphism for free:
  - » no need to specify that a function is polymorphic
  - » no need to "instantiate" a polymorphic function when it is applied

Polymorphism

```
> fun len x = if null x then 0
=     else 1 + len (tl x);
```

- works for any kind of list. What is its type expression?
  val len : fn = ‘a list -> int
- ‘a denotes a type variable. Implicit universal quantification:
- **for any a, len applies to a list of a’s.**

```
>  fun copy lis = if null lis then nil
=         else hd (lis) :: copy (tl lis);
```


Type inference and unification

- Type expressions are built by solving a set of equations
- Substituting type expressions for type variables

```
> fun foo [] = 0
    | foo  (x::xs) = x + (foo xs);
 foo : ‘a -> ‘b ?
```

- ‘b must be int, since result of foo for an empty list is an int and the result of foo is also an operand of +
- ‘a must be int list, since x is also operand of +
- val foo = fn : int list -> int

Unification algorithm

- A type variable can be unified with another variable
- ‘a unifies with ‘b => ‘a and ‘b are the same
- A type variable can be unified with a constant
- ‘a unifies with int => all occurences of ‘a mean int
- A type variable can be unified with a type expression
- ‘a unifies with ‘b list
- ‘a does not unify with ‘a list
- A constant can be unified with itself int is int
- An expression can be unified with another expression if the constructors are identical and if the arguments can be unified:
- (int -> int) list unifies with ‘a list, ‘a is a function on integers

Multiple Arguments?

- All functions in ML take exactly one argument
  - » If a function needs multiple arguments, we can
    1. pass a tuple:

```
- (53, "hello"); (* a tuple *)
val it = (53, "hello") : int * string
```

    we can also use tuples to return multiple results
    2. use currying (named after Haskell Curry, a logician)

The Tuple Solution

- Another function; takes two lists and returns their concatenation

```
- fun append1 ([ ], ys) = ys
    | append1 (x::xs , ys) = x :: append1 (xs , ys );
val append1 = fn: 'a list * 'a list -> 'a list
- append1 ([1 ,2 ,3] , [8 ,9]);
val it = [1 ,2 ,3 ,8 ,9] : int list
```


Currying

- The same function, written in curried style:

```
- fun append2 [ ] ys = ys
    | append2 (x:: xs) ys = x :: ( append2 xs ys );
val append2 = fn: 'a list -> 'a list -> 'a list
- append2 [1 ,2 ,3] [8 ,9];
val it = [1 ,2 ,3 ,8 ,9] : int list
- val app123 = append2 [1 ,2 ,3];
val app123 = fn : int list -> int list
- app123 [8 ,9];
val it = [1 ,2 ,3 ,8 ,9] : int list
```

More Partial Application

- But what if we want to provide the other argument instead, i.e., append [8,9] to its argument?
  - » here is one way: (the Ada/C/C++/Java way)

```
fun appTo89 xs = append2 xs [8,9]
```

  - » here is another: (using a higher-order function)

```
val appTo89 = flip append2 [8,9]
```

- flip is a function which takes a curried function f and returns a function that works like f but takes its arguments in the reverse order
  - » In other words, it "flips" f's two arguments.
  - » We define it on the next slide…

Type Inference Example

```
fun flip f y x = f x y
```

The type of flip is (→ →) → → → . Why?

- Consider (f x). f is a function; its parameter must have the same type as x

  f : A → B x : A (f x) : B

- Now consider (f x y). Because function application is left-associative, f x y ≡ (f x) y. Therefore, (f x) must be a function, and its parameter must have the same type as y:

  (f x) : C → D y : C (f x y) : D

- Note that B must be the same as C → D. We say that B must unify with C → D
- The return type of flip is whatever the type of f x y is. After renaming the types, we have the type given at the top

User-defined types and inference

- A user-defined type introduces constructors:
- datatype tree = leaf of int | node of tree * tree
- leaf and node are type constructors

```
> fun sum (leaf (t)) = t
      | sum (node (t1, t2)) = sum t1 + sum t2;
val sum = fn : tree -> int
```


Type Rules

- The type system is defined in terms of inference rules. For example, here is the rule for variables:

$$\frac{(x:\tau) \in E}{E \vdash x : \tau}$$

- and the one for function calls:

$$\frac{E \vdash e_1 : \tau' \to \tau \quad E \vdash e_2 : \tau'}{E \vdash e_1\ e_2 : \tau}$$

- and here is the rule for if expressions:

$$\frac{E \vdash e : \texttt{bool} \quad E \vdash e_1 : \tau \quad E \vdash e_2 : \tau}{E \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Passing Functions

```
- fun exists pred [ ]          = false
    | exists pred (x:: xs)     = pred x orelse
                                 exists pred xs;
val exists = fn : ('a -> bool ) -> 'a list -> bool
```

- pred is a predicate : a function that returns a boolean
- exists checks whether pred returns true for any member of the list

```
- exists (fn i => i = 1) [2, 3, 4];
val it = false : bool
```


Applying Functionals

```
- exists (fn i => i = 1) [2, 3, 4];
val it = false : bool
```

Now partially apply exists:

```
- val hasOne = exists (fn i => i = 1);
val hasOne = fn : int list -> bool
- hasOne [3,2,1];
val it = true : bool
```

Functionals 2

```
fun all pred [ ]          = true
   | all pred (x:: xs)    = pred x andalso
                            all pred xs
fun filter pred [ ] = [ ]
   | filter pred (x:: xs) = if pred x
                            then x :: filter pred xs
                            else filter pred xs
```

$$\texttt{all} : (\alpha \to \texttt{bool}) \to \alpha\,\texttt{list} \to \texttt{bool}$$

$$\texttt{filter} : (\alpha \to \texttt{bool}) \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$$

Block Structure and Nesting

let provides local scope:

```
(* standard Newton - Raphson *)
fun findroot (a, x, acc ) =
   let val nextx = (a / x + x) / 2.0
     (* nextx is the next approximation *)
   in
     if abs (x - nextx ) < acc * x
     then nextx
     else findroot (a, nextx , acc )
   end
```

Let declarations

- Let declarations bind a value with a name over an explicit scope

```
fun fib 0 = 0
  | fib n = let fun fibb (x, prev, curr) = if x=1 then curr
      else fibb (x-1, curr, prev + curr)
in
  fibb(n, 0, 1)
end;
```

- val fib = fn : int -> int

- > fib 20;
- val it = 6765 : int

A Classic in Functional Form: Mergesort

```
fun mrgSort op < [ ] = [ ]
    | mrgSort op < [x] = [x]
    | mrgSort op < (a:: bs) =
        let fun partition (left , right , [ ]) =
                        (left , right ) (* done partitioning *)
              | partition (left , right , x:: xs) =
                        (* put x to left or right *)
                        if x < a
                        then partition (x:: left , right , xs)
                        else partition (left , x:: right , xs)
              val (left , right ) = partition ([ ] , [a], bs)
        in
              mrgSort op < left @ mrgSort op < right
        end
```

$$\texttt{mrgSort} : (\alpha * \alpha \rightarrow \texttt{bool}) \rightarrow \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$$

Another Variant of Mergesort

```
fun mrgSort op < [ ] = [ ]
    | mrgSort op < [x] = [x]
    | mrgSort op < (a:: bs) =
        let fun deposit (x, (left , right )) =
                        if x < a
                        then (x:: left , right )
                        else (left , x:: right )
            val (left , right ) = foldr deposit ([ ] , [a]) bs
        in
            mrgSort op < left @ mrgSort op < right
        end
```

$$\texttt{mrgSort} : (\alpha * \alpha \rightarrow \texttt{bool}) \rightarrow \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$$

The Type System

- primitive types: bool, int, char, real, string, unit
- constructors: list, array, product (tuple), function, record
- “datatypes”: a way to make new types
- structural equivalence (except for datatypes)
  - » as opposed to name equivalence in e.g., Ada
- an expression has a corresponding type expression
- the interpreter builds the type expression for each input
- type checking requires that type of functions’ parameters match the type of their arguments, and that the type of the context matches the type of the function’s result

More on ML Records

- Records in ML obey structural equivalence (unlike records in many other languages).
- A type declaration: only needed if you want to refer to this type by name

```
type vec = { x : real , y : real }
```

- A variable declaration:

```
val v = { x = 2.3 , y = 4.1 }
```

- Field selection:

```
#x v
```

- Pattern matching in a function:

```
fun dist {x,y} =
        sqrt (pow (x, 2.0) + pow (y, 2.0))
```


Data Types

- A datatype declaration:
  - » defines a new type that is not equivalent to any other type (name equivalence)
  - » introduces data constructors
    - data constructors can be used in patterns
  - » they are also values themselves

Datatype Example

```
datatype tree = Leaf of int
              | Node of tree * tree
```

- Leaf and Node are data constructors:
  - » Leaf : int ⟶ tree
  - » Node : tree * tree ⟶ tree
- We can define functions by pattern matching:

```
fun sum ( Leaf t) = t
      | sum ( Node (t1 , t2 )) = sum t1 + sum t2
```


Parameterized Data Types

```
> fun flatten (leaf (t)) = [t]
    | flatten (node (t1, t2)) = flatten (t1) @ flatten (t2);
val flatten = fn : tree -> int list
> datatype ‘a gentree = leaf of ‘a
          | node of ‘a gentree * ‘a gentree;

> val names = node (leaf (“this”), leaf (“that”));
val names = … string gentree
```

Parametrized Datatypes

```
fun flatten ( Leaf t)                    = [t]
   | flatten ( Node (t1 , t2 ))  =
     flatten t1 @ flatten t2
```

```
flatten : tree → int list
```

```
datatype 'a gentree =
        Leaf of 'a
   | Node of 'a gentree * 'a gentree
val names = Node ( Leaf " this ", Leaf " that ")
```

```
names : string gentree
```


The Rules of Pattern Matching

- Pattern elements:
  - » integer literals: 4, 19
  - » character literals: #'a'
  - » string literals: "hello"
  - » data constructors: Node (· · ·)
    - depending on type, may have arguments, which would also be patterns
  - » variables: x, ys
  - » wildcard: _
- Convention is to capitalize data constructors, and start variables with lower-case.

More Rules of Pattern Matching

- Special forms:
  - (), {} – the unit value
  - [ ] – empty list
  - [p1, p2, · · ·, pn]
    - means (p1 :: (p2 :: · · · (pn :: [])· · ·))
  - (p1, p2, · · ·, pn) – a tuple
  - {field1, field2, · · · fieldn} – a record
  - {field1, field2, · · · fieldn, ...}
    - a partially specified record
  - v as p
    - v is a name for the entire pattern p

Common Idiom: Option

- option is a built-in datatype:

  datatype 'a option = NONE | SOME of 'a
- Defining a simple lookup function:

```
fun lookup eq key [] = NONE
      | lookup eq key ((k,v):: kvs ) =
        if eq (key , k)
        then SOME v
        else lookup eq key kvs
```

- Is the type of lookup:

$$(\alpha * \alpha \to \texttt{bool}) \to \alpha \to (\alpha * \beta)\,\texttt{list} \to \beta\,\texttt{option?}$$

- No! It's slightly more general:

$$(\alpha_1 * \alpha_2 \to \texttt{bool}) \to \alpha_1 \to (\alpha_2 * \beta)\,\texttt{list} \to \beta\,\texttt{option}$$

Another Lookup Function

- We don't need to pass two arguments when one will do:

```
fun lookup _ [] = NONE
    | lookup checkKey ((k,v):: kvs ) =
      if checkKey k
      then SOME v
      else lookup checkKey kvs
```

- The type of this lookup:

$$(\alpha \to \texttt{bool}) \to (\alpha * \beta)\,\texttt{list} \to \beta\,\texttt{option}$$

Useful Library Functions

- $\texttt{map} : (\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$

```
map (fn i => i + 1) [7, 15, 3]
⟹ [8, 16, 4]
```

- $\texttt{foldl} : (\alpha * \beta \to \beta) \to \beta \to \alpha\,\texttt{list} \to \beta$

```
foldl (fn (a,b) => "(" ^ a ^ "+" ^ b ^ ")")
      "0"  ["1", "2", "3"]
⟹ "(3+(2+(1+0)))"
```

- $\texttt{foldr} : (\alpha * \beta \to \beta) \to \beta \to \alpha\,\texttt{list} \to \beta$

```
foldr (fn (a,b) => "(" ^ a ^ "+" ^ b ^ ")")
      "0"  ["1", "2", "3"]
⟹ "(1+(2+(3+0)))"
```

- $\texttt{filter} : (\alpha \to \texttt{bool}) \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

Overloading

- Ad hoc overloading interferes with type inference:

```
fun plus x y = x + y
```

- Operator '+' is overloaded, but types cannot be resolved from context (defaults to int).
- We can use explicit typing to select interpretation:

```
fun mix1 (x, y, z) = x * y + z : real
fun mix2 (x: real , y, z) = x * y + z
```


Type system does not handle overloading well

```
>   fun plus x y = x + y;
```

- operator is overloaded, cannot be resolved from context (error in some versions, defaults to int in others)

- Can use explicit typing to select interpretation:

```
> fun mix (x, y ,z) = x * y + z : real;
val mix = fn : (real * real * real) -> real
```

Parametric Polymorphism vs. Generics

- a function whose type expression has type variables applies to an infinite set of types
- equality of type expressions means structural not name equivalence
- all applications of a polymorphic function use the same body: no need to instantiate

```
let  val ints = [1, 2, 3];
     val strs = [" this ", " that "];
in
     len ints + (* int list -> int *)
     len strs (* string list -> int *)
end ;
```


ML Signature

- An ML signature specifies an interface for a module

```
signature STACKS =
sig
   type stack
   exception Underflow
   val empty : stack
   val push : char * stack -> stack
   val pop : stack -> char * stack
   val isEmpty : stack -> bool
end
```

Programming in the large in ML

- Need mechanisms for
  - Modularization
  - Information hiding
  - Parametrization of interfaces
- While retaining type inference
- **Modules**: like packages / namespaces
- **Signatures**: like package specifications /Java interfaces
- **Functors**: like generics with formal packages

ML Structure

```
structure Stacks : STACKS =
struct
    type stack = char list
    exception Underflow
    val empty = [ ]
    val push = op ::
    fun pop (c:: cs)     = (c, cs)
        | pop []                 = raise Underflow
    fun isEmpty []       = true
    | isEmpty _                  = false
end
```

Using a structure

```
- use ("complex.ml");
signature Complex :
  sig
  ....
- Complex.prod (Complex.i, Complex.i);
val it = (~1.0, 0.0);

- val pi4 = (0.707, 0.707);
val pi4 ... real * real                    structural equivalence
- Complex.prod (pi4, pi4);
val  it = ... : Complex.t;
```


Multiple implementations

```
structure complex1 : CMPLX =
struct
  type t = real*real;        (* cartesian representation *)
  val zero = (0.0, 0.0);
  val i    = (0.0, 1.0);
...
Structure ComplexPolar: CMPLX =
Struct
    type t = real*real       (*polar representation*)
     val zero = (0.0, 0.0);
     val pi = 3.141592;
     val i := (0.0, pi / 2.0);
```

Information Hiding

- **Only signature should be visible**
- Declare structure to be opaque:

structure polar :> CMPLX = ….

- (Structure can be opaque or transparent depending on context).
- Can export explicit constructors and equality for type. Otherwise, equivalent to limited private types in Ada.
- Can declare as eqtype to export equality

Functors

- Structures and signatures are not first-class objects.
- A program (structure) can be parametrized by a signature

```
functor testComplex (C : CMPLX) =
struct
    open C;              (*equivalent to use clause*)
    fun FFT..
end;

structure testPolar = testComplex (Complexpolar);

 (* equivalent to instantiation with a package *)
```

Imperative Programming in ML

- A real language needs operations with side effects, state, and variables
- Need to retain type inference

```
- val p = ref 5;
val p = ref 5 : int ref ;     (* ref is a  type constructor*)
- !p * 2;                     (* dereference operation *)
val it = 10: int;
- p := !p * 3;
val it = ( ) : unit           (* assignment has no value *)
```

References are equality types (pointer equality)

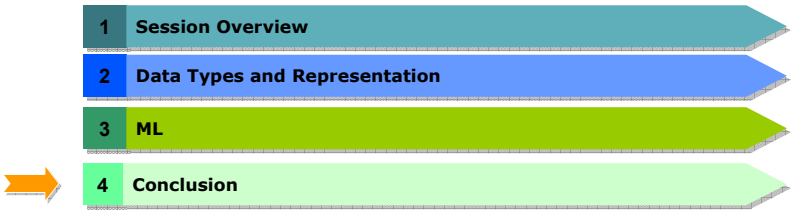
References and polymorphism

```
fun Id x = x;              (* id : 'a -> 'a *)
 val fp = ref Id;          (*a function pointer *)
 fp := not;     !fp 5 ;
```

- (* must be forbidden! *)

- In a top level declaration, all references must be monomorphic.

Agenda



Assignments & Readings

- Readings
  - Chapter Section 7
- Assignment #4
  - See Assignment #4 (i.e., Programming Assignment #2) posted under "handouts" on the course Web site - Ongoing
  - Due on November 15, 2010

Next Session: Program Structure, OO Programming