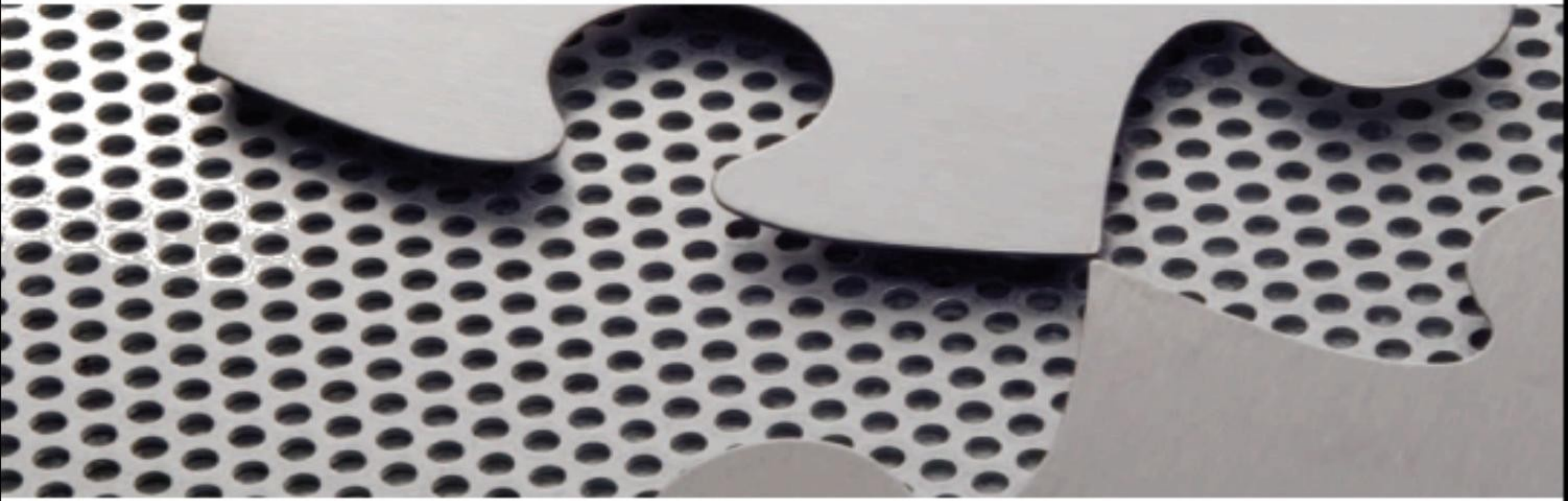
Programming Languages
Third Edition



Chapter 8

Data Types

Objectives

- Understand data types and type information
- Understand simple types
- Understand type constructors
- Be able to distinguish type nomenclature in sample languages
- Understand type equivalence

Objectives (cont'd.)

- Understand type checking
- Understand type conversion
- Understand polymorphic type checking
- Understand explicit polymorphism
- Perform type checking in TinyAda

Introduction

- Every program uses data, either explicitly or implicitly, to arrive at a result
- Data type: the basic concept underlying the representation of data in programming languages
- Data in its most primitive form is simply a collection of bits
  - This does not provide the kinds of abstraction necessary for large programs
- Programming languages include a set of simple data entities and mechanisms for constructing new ones

Introduction (cont'd.)

- Machine dependencies are often part of the implementation of these abstractions
- **Finitude** of data:
  - In mathematics, integer set is infinite
  - In hardware, there is always a largest and smallest integer
- Much disagreement among language designers on the extent to which type information should be made explicit and used to verify program correctness

Introduction (cont'd.)

- Reasons to have some form of static type-checking:
  - **Execution efficiency**: allows compilers to allocate memory efficiently
  - **Translation efficiency**: static types allow the compiler to reduce the amount of code to be compiled
  - **Writability**: allows many common programming errors to be caught early
  - **Security** and **reliability**: reduces the number of execution errors

Introduction (cont'd.)

- Reasons to have some form of static type-checking (cont'd.):
  - **Readability**: explicit types help to document data design
  - **Remove ambiguities**: explicit types can be used to resolve overloading
  - **Design tool**: explicit types highlight design errors and show up as translation-time errors
  - **Interface consistency** and **correctness**: explicit data types help in verification of large programs
- **Data type**: the basic abstraction mechanism

Data Types and Type Information

- Program data can be classified according to their **types**
- Type name represents the possible values that a variable of that type can hold and the way those values are represented internally
- **Data type** (definition 1): a set of values
  - `int x;` means the same as value of $x \in \mathit{Integers}$
- **Data type** (definition 2): a set of values, together with a set of operations on that values having certain properties
  - A data type is actually a mathematical algebra

Data Types and Type Information (cont'd.)

- **Type checking**: the process a translator goes through to determine whether type information in a program is consistent
- **Type inference**: the process of attaching types to expressions
- Type constructors: mechanisms used with a group of basic types to construct more complex types
  - Example: Array takes a base type and a size or range indication and constructs a new data type
- **User-defined types**: types created using type constructors

Data Types and Type Information (cont'd.)

- **Type declaration** (or **type definition**): used to associate a name with a new data type
- **Anonymous type**: a type with no name
  - Can use `typedef` in C to assign a name
- **Type equivalence**: rules for determining if two types are the same
- **Type system**: methods for constructing types, the type equivalence algorithm, type inference rules, and type correctness rules

Data Types and Type Information (cont'd.)

- **Strongly typed**: a language that specifies a statically applied type system that guarantees all data-corrupting errors will be detected at the earliest possible point
  - Errors are detected at translation time, with a few exceptions (such as array subscript bounds)
- **Unsafe programs**: programs with data-corrupting errors
- **Legal programs**: proper subset of safe programs; those programs accepted by a translator

Data Types and Type Information (cont'd.)

- **Weakly-typed language**: one that has loopholes that may allow unsafe programs
- **Untyped** (or **dynamically typed**) languages: languages without static type systems
  - All safety checking is performed at execution time
- **Polymorphism**: allows names to have multiple types while still permitting static type checking

Simple Types

- **Predefined types**: those types supplied with a language, from which all other types are constructed
  - Generally specified using either keywords or predefined identifiers
  - May include some variations on basic types, such as for numeric types
- **Simple types**: have no other structure than their inherent arithmetic or sequential structure
  - Usually includes predefined types
  - Includes **enumerated types** and **subrange types**

Simple Types (cont'd.)

- **Enumerated types**: sets whose elements are named and listed explicitly
  - Example: In C: `enum Color {Red, Green, Blue};`
  - Are **ordered** in most languages: order in which the values are listed is important
  - Most languages include a predefined **successor** and **predecessor** operation for enumerated types
  - No assumptions are made about how the listed values are represented internally

Simple Types (cont'd.)

```
with Text_IO; use Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Enum is
  type Color_Type is (Red,Green,Blue);
  -- define Color_IO so that Color_Type values can be -- printed
  package Color_IO is new Enumeration_IO(Color_Type);
  use Color_IO;
  x : Color_Type := Green;
begin
  x := Color_Type'Succ(x); -- x is now Blue
  x := Color_Type'Pred(x); -- x is now Green
  put(x); -- prints GREEN
  new_line;
end Enum;
```

Figure 8.1 An Ada program demonstrating the use of an enumerated type

Simple Types (cont'd.)

```c
#include <stdio.h>

enum Color {Red,Green,Blue};
 enum NewColor {NewRed = 3, NewGreen = 2, NewBlue = 2};

main(){
  enum Color x = Green; /* x is actually 1 */
  enum NewColor y = NewBlue; /* y is actually 2 */
  x++; /* x is now 2, or Blue */
  y--; /* y is now 1 -- not even in the enum */
  printf("%d\n",x); /* prints 2 */
   printf("%d\n",y); /* prints 1 */
  return 0;
}
```

Figure 8.2 A C program demonstrating the use of an enumerated type, similar to Figure 8.1

Simple Types (cont'd.)

- **Subrange types**: contiguous subsets of simple types specified by giving least and greatest elements
  - Example:

```
type Digit_Type is range 0..9;
```

- **Ordinal typ** … order on the set of values
  - All numeric integer types are ordinal types
  - Always have comparison operators
  - Often have successor and predecessor operations
- Real numbers are not ordinal; they have no successor and predecessor operations

Simple Types (cont'd.)

- Allocation schemes are usually dependent on the underlying hardware for efficiency
- IEEE 754 standard tries to define standard representations

Type Constructors

- Since data types are sets, set operations can be used to construct new types from existing ones
- Set operations that can be used include Cartesian product, union, powerset, function set, and subset
  - These set operations are called type constructors
- Example: subrange type is formed using subset construction
- There are type constructors that do not correspond to mathematical set constructions, and some set operations that do not correspond to type constructors

Cartesian Product

- Given two sets U and V, the Cartesian product (or cross product) consists of all ordered pairs of elements from U and V:

$$U \times V = \{(u, v) \mid u \text{ is in } U \text{ and } v \text{ is in } V\}$$

- In many languages, the Cartesian product type constructor is available as the **record** or **structure construction**

Cartesian Product (cont'd.)

- Example: In C, this `struct` declaration constructs the Cartesian product of type `int` × `char` × `double`.

```
struct IntCharReal{
    int i;
    char c;
    double r;
};
```

Cartesian Product (cont'd.)

- Difference between a Cartesian product and a record structure:
  - In a record structure, components have names
  - In a Cartesian product, they are referred to by position
- Most languages consider component names to be part of the type defined by a record structure
- **Tuple**: a purer form of record structure in ML that is essentially identical to the Cartesian product

```
type IntCharReal = int * char * real;
```

Cartesian Product (cont'd.)

- **Class**: a data type found in object-oriented languages
  - Includes **member functions** or **methods**
  - Closer to the second definition of data type, which includes functions that act on the data

Union

- Union of two types: formed by taking the set theoretic union of the sets of their values
- Two varieties
  - **Discriminated** unions: a **tag** or **discriminator** is added to the union to distinguish the type of its elements
  - **Undiscriminated** unions: lack the tags; assumptions must be made about the type of any value
- A language with undiscriminated unions has an unsafe type system

Union (cont'd.)

- In C and C++, the union type constructor creates undiscriminated unions
- Example:

```
union IntOrReal{
    int i;
    double r;
};
```

  – If `x` is a variable of type `union IntOrReal`, `x.i` is interpreted as an `int`, and `x.r` is interpreted as a `double`

Union (cont'd.)

- Ada has a completely safe union mechanism called a **variant record**

```
type Disc is (IsInt, IsReal);
type IntOrReal (which: Disc) is
record
   case which is
      when IsInt =>  i: integer;
      when IsReal => r: float;
   end case;
end record;
...
x: IntOrReal := (IsReal,2.3);
```

Union (cont'd.)

- In ML, declare an enumeration with the vertical bar for "or":

```
datatype IntOrReal = IsInt of int | IsReal of real;
```

- Then use pattern matching:

```
fun printInt x =
   (print("int: "); print(Int.toString x); print("\n"));

fun printReal x =
   (print("real: "); print(Real.toString x); print("\n"))

fun printIntOrReal x =
   case x of
      IsInt(i) => printInt i |
      IsReal(r) =>printReal r ;
```

Union (cont'd.)

- The tags `IsInt` and `IsReal` in ML are called **data constructors**, since they construct data of each kind within a union
- Unions are useful in reducing memory allocation requirements for structures when different data items are not needed simultaneously
- Unions are not needed in object-oriented languages
  - Use inheritance to represent different non-overlapping data requirements

Subset

- A subset in math is specified by giving a rule to distinguish its elements
- Similar rules can be given in programming languages to establish new types as subsets of known types
- Ada has a subtype mechanism:

```
subtype IntDigit_Type is integer range 0..9;
```

- Variant parts of records can be fixed using subtype

```
subtype IRInt is IntOrReal(IsInt);
subtype IRReal is IntOrReal(IsReal);
```

Subset (cont'd.)

- Such subset types **inherit** operations from their parent types
  - Most languages do not allow the programmer to specify which operations are inherited and which are not
- Inheritance in object-oriented languages can also be viewed as a subtype mechanism
  - With a great deal more control over which operations are inherited

Arrays and Functions

- The set of all functions `f:U→V` can give rise to a new type in two ways:
  - **Array type**
  - **Function type**
- If `U` is an ordinal type, the function `f` can be thought of as an **array** with **index type** `U` and **component type** `V`
  - If `i` is in `U`, then `f(i)` is the `i`th component of the array
  - Whole function can be represented by the sequence or tuples of its values `(f(low),…,f(high))`

Arrays and Functions (cont'd.)

- Arrays are sometimes called **sequence types**
- Typically, array types can be defined with or without sizes
  - To define a variable of an array type, usually necessary to specify size at translation time since arrays are normally allocated statically
- In C, the size of an array must be a literal, not a computed constant
- Cannot dynamically define an array size in C or C++

Arrays and Functions (cont'd.)

- C allows arrays without specified size to be parameters to functions (they are essentially pointers), but the size must be supplied
  – Size of the array is not part of the array in C or C++

```
int array_max (int a[], int size){
    int temp, i;
    assert(size > 0);
    temp = a[0];
    for (i = 1; i < size; i++)
        if (a[i] > temp) temp = a[i];
    return temp;
}
```

Arrays and Functions (cont'd.)

- In Java, arrays are always dynamically (heap) allocated, and the size can be dynamically specified (but cannot change)
  - Size is stored when an array is allocated in its **length** property

```
import java.io.*;
import java.util.Scanner;
public class ArrayTest{

   static private int array_max(int[] a){      // note location of []
      int temp;
      temp = a[0];
      // length is part of a
      for (int i = 1; i < a.length; i++)
         if (a[i] > temp) temp = a[i];
      return temp;
   }
   public static void main (String args[]){   // this placement of [] also
                                              // allowed
      System.out.print("Input a positive integer: ");
      Scanner reader = new Scanner(System.in));
      int size = reader.nextInt();
      int[] a = new int[size] ; // Dynamic array allocation
      for (int i = 0; i < a.length; i++) a[i] = i;
      System.out.println(array_max(a));
   }
}
```

Figure 8.3 A Java program demonstrating the use of arrays

Arrays and Functions (cont'd.)

- Ada allows array types declared without a size, called **unconstrained arrays**, but requires a size when array variables are declared
- Multidimensional arrays are also possible
- Arrays are perhaps the most widely used type constructor
- Implementation is extremely efficient
  - Space is allocated sequentially in memory
  - Indexing is performed by an offset calculation from the starting address

Arrays and Functions (cont'd.)

- For multidimensional arrays, must decide which index to use first in the allocation scheme
  - **Row-major form**: all values of the first row are allocated first, then all values of the second row, etc.
  - **Column-major form**: all values of the first column are allocated first, then all values of the second column, etc.
- Functional languages usually do not supply an array type; most use the **list** in place of an array
  - Scheme has a **vector** type

Arrays and Functions (cont'd.)

- General function and procedure types can be created in some languages
- Example: in C, define a function type from integers to integers: `typedef int (*IntFunction)(int);`
  - Use this type for variables or parameters:

```
int square(int x) { return x*x; }
IntFunction f = square;
int evaluate(IntFunction g, int value)
{   return g(value); }
...
printf("%d\n", evaluate(f,3)); /* prints 9 */
```

Arrays and Functions (cont'd.)

- In ML, you can define a function type:

```
type IntFunction = int -> int;
```

  – Use it in a similar fashion:

```
fun square (x: int) = x * x;
val f = square;
fun evaluate (g: IntFunction, value: int) = g value;
...
evaluate(f,3); (* evaluates to 9 *)
```

Pointers and Recursive Types

- **Reference** or **pointer** constructor: constructs the set of all addresses that refer to a specified type
  - Does not correspond to a set operation
- Example in C: `typedef int* IntPtr;`
  - Constructs the type of all addresses where integers are stored
- Pointers are implicit in languages that perform automatic memory management
  - In Java, all objects are implicitly pointers that are allocated explicitly (using the `new` operator) but deallocated automatically by garbage collection

Pointers and Recursive Types (cont'd.)

- **Reference**: address of an object under control of the system that cannot be used as a value or operated on in any way (except copying)
- **Pointer**: can be used as a value and manipulated by the programming
- References in C++ are created by a **postfix & operator**
- **Recursive type**: a type that uses itself in its declaration

Pointers and Recursive Types (cont'd.)

- Recursive types are important in data structures and algorithms
  - Represent data whose size and structure is not known in advance and may change as computation proceeds
  - Examples: lists and binary trees
- Consider this C-like declaration of lists of characters:

```
struct CharList{
   char data;
   struct CharList next; /* not legal C! */
};
```

Pointers and Recursive Types (cont'd.)

- C requires that each data type have a fixed maximum size determined at translation time
  - Must use pointer to allow manual dynamic allocation to overcome this problem

```
struct CharListNode{
   char data;
   struct CharListNode* next; /* now legal */
};
typedef struct CharListNode* CharList;
```

  - Each individual element in a `CharListNode` now has a fixed size, and they can be strung together to form a list of arbitrary size

Data Types and the Environment

- Pointer types, recursive types, and general function types require space to be allocated dynamically
  - Require fully dynamic environments with automatic allocation and deallocation (garbage collection)
  - Found in the functional languages and the more dynamic object-oriented languages
- More traditional languages (C++ and Ada) restrict these types so that a heap (a dynamic space under programming control) is sufficient
- Environment issues will be discussed in full in Chapter 10

Type Nomenclature in Sample Languages

- Various language definitions use different and confusing terminology to define similar things
- This section gives a brief description of the differences among three languages: C, Java, and Ada

C

- Simple data types are called **basic types**, including:
  - `void` type
  - **Numeric types**:
    - **Integral types**, which are ordinal (12 possible kinds)
    - **Floating types** (3 possible kinds)
- Integral types can be signed or unsigned
- **Derived types**: constructed using type constructors

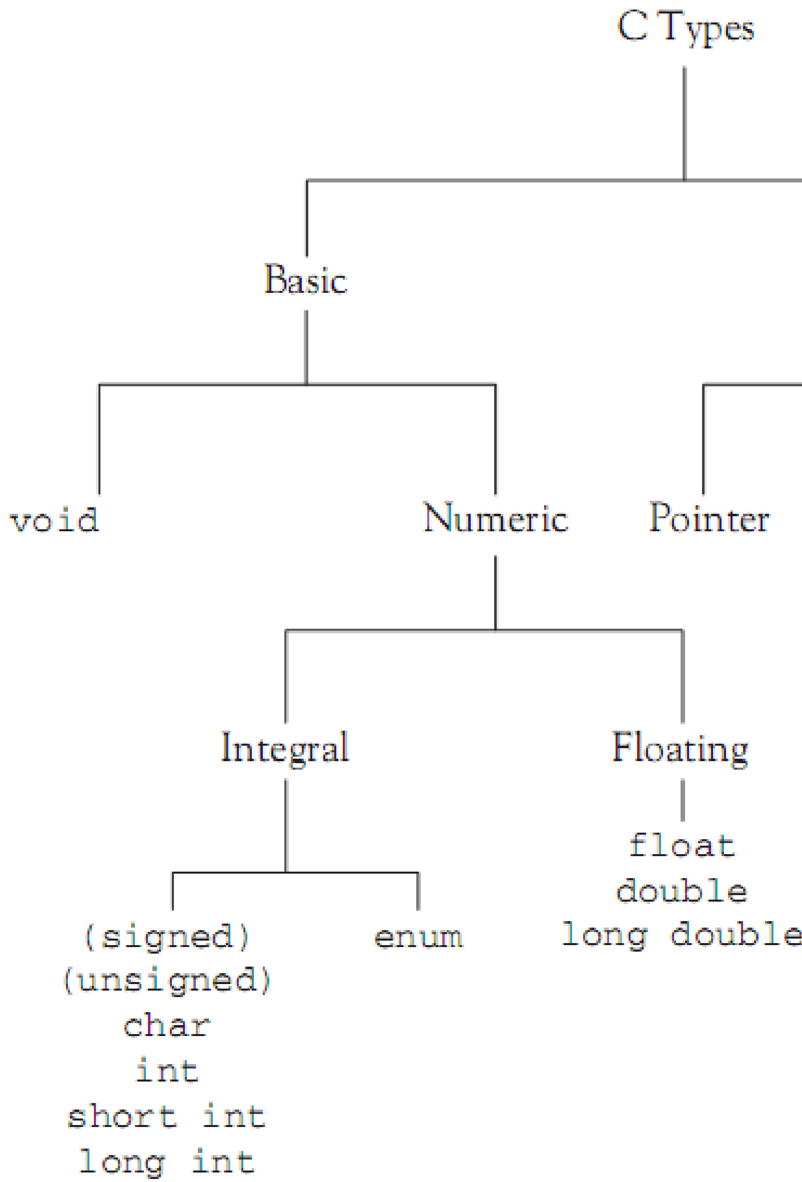


Figure 8.5 The type structure of C

Java

- Simple types are called primitive types, including:
  - Boolean (not numeric or ordinal)
  - Numeric, including:
    - Integral (ordinal)
    - Floating point
- **Reference types**: constructed using type constructors
  - Array
  - Class
  - Interface

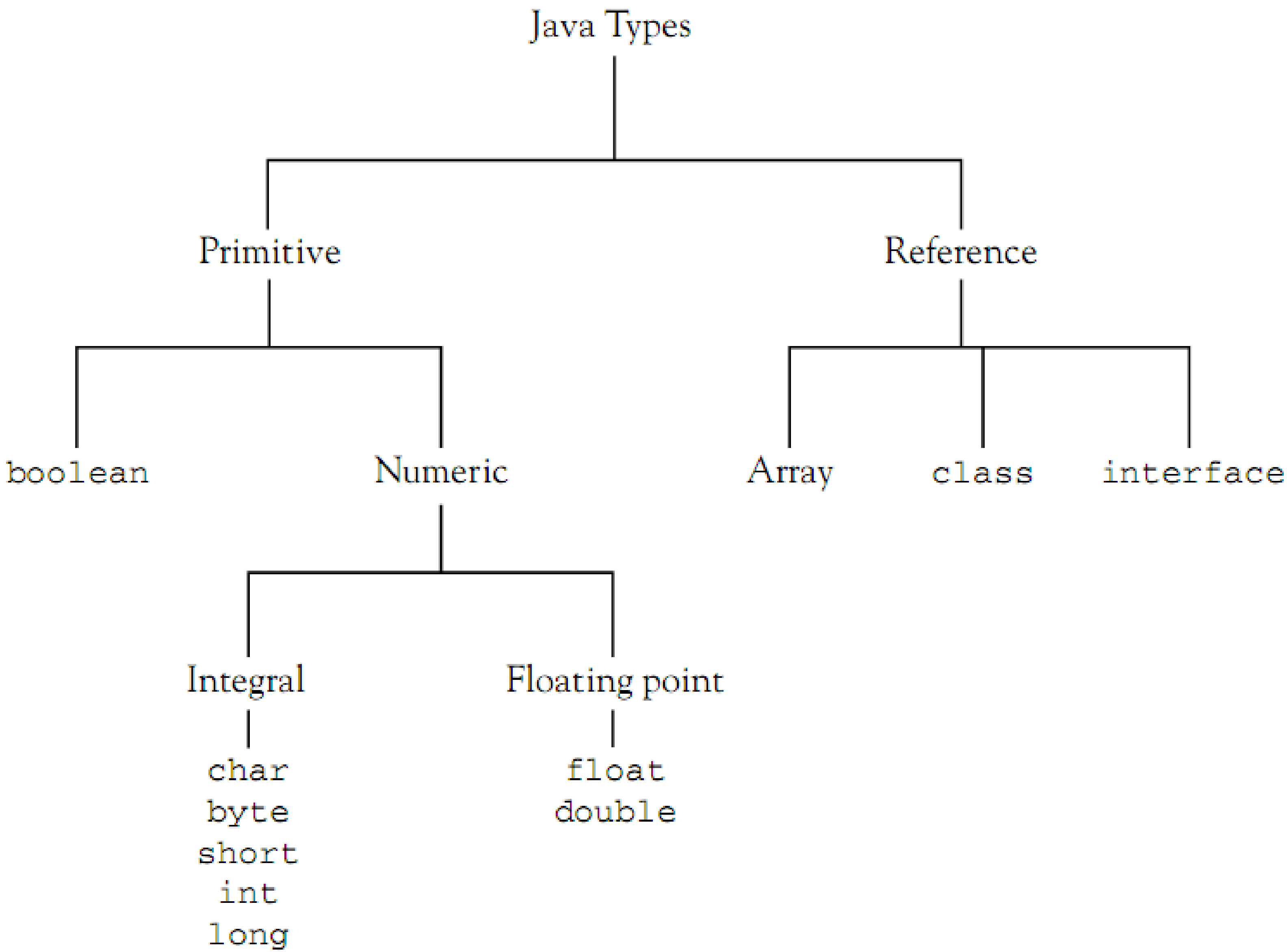


Figure 8.6 The type structure of Java

Ada

- Ada has a rich set of types
  - Simple types are called **scalar types**
  - Ordinal types are called **discrete types**
  - **Numeric types** include **real** and **integer** types
  - Pointer types are called `access` types
  - Array and record types are called **composite types**

Ada (cont'd.)

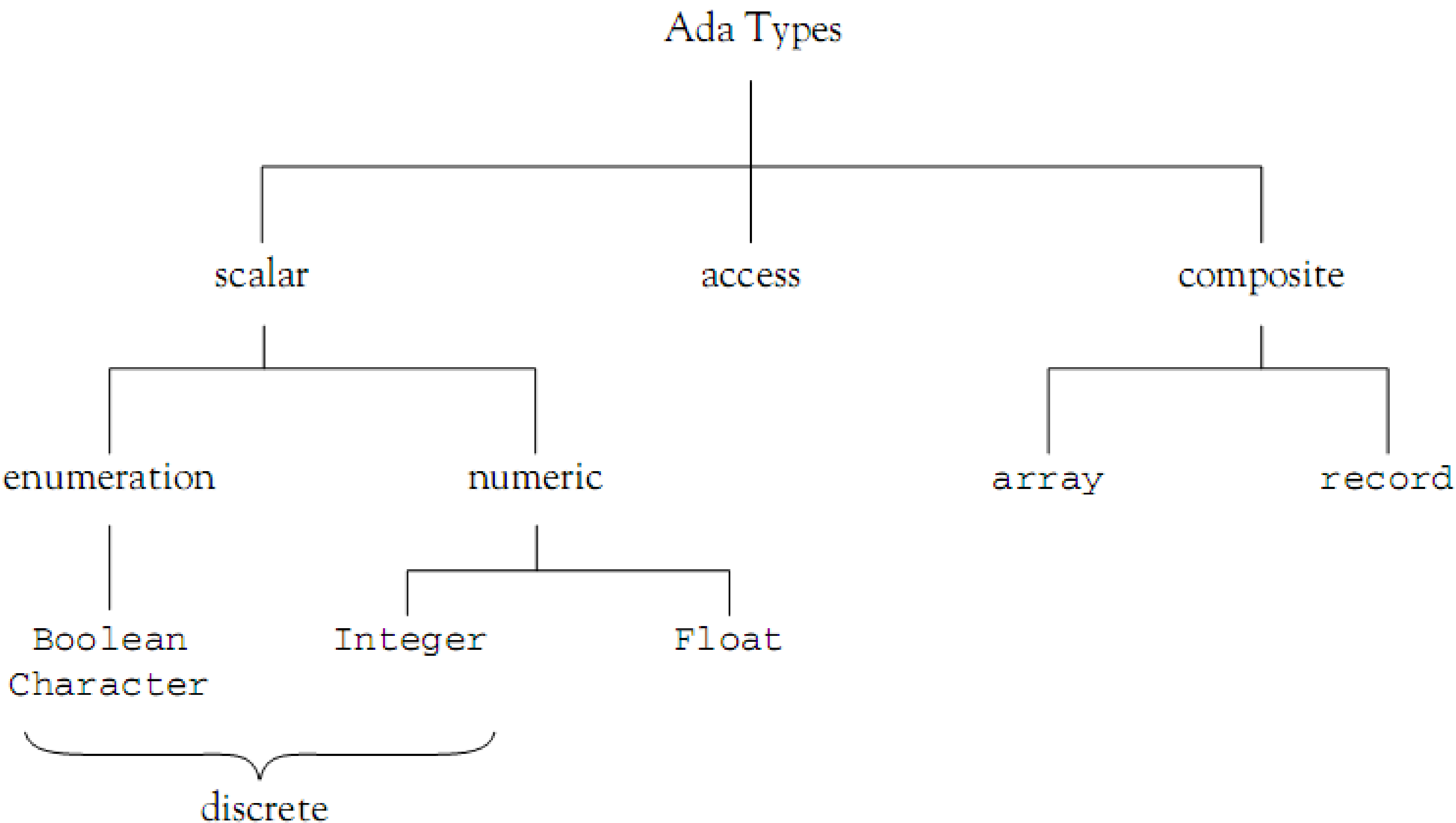


Figure 8.7 The type structure of Ada (somewhat simplified)

Type Equivalence

- **Type equivalence**: when are two types the same?
- Can compare the sets of values as sets
  - Are the same if they contain the same values
- **Structural equivalence**: two data types are the same if they have the same structure
  - Built in the same way using the same type constructors from the same simple types
  - This is one of the principal forms of type equivalence in programming languages

Type Equivalence (cont'd.)

- Example:
  - `Rec1` and `Rec2` are structurally equivalent
  - `Rec1` and `Rec3` are not structurally equivalent (`char` and `int` fields are reversed)

```
struct Rec1{
    char x;
    int y;
    char z[10];
};

struct Rec2{
    char x;
    int y;
    char z[10];
};

struct Rec3{
    int y;
    char x;
    char z[10];
};
```

Type Equivalence (cont'd.)

- Structural equivalence is relatively easy to implement (except for recursive types)
  - Provides all the information needed to perform error checking and storage allocation
- To check structural equivalence, a translator may represent types as trees and check equivalence recursively on subtrees
- Questions still arise over how much information is included in a type under the application of a type constructor

Type Equivalence (cont'd.)

- Example: are `A1` and `A2` structurally equivalent?

```
typedef int A1[10];
typedef int A2[20];
```

  – Yes, if size of the index set is not part of an array type
  – Otherwise, no
- Similar question arises regarding member names of structures

Type Equivalence (cont'd.)

- Example: Are these two structures structurally equivalent?

```
struct RecA{
    char x;
    int y;
};
```

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

  - If structures are considered to be just Cartesian products, then yes
  - They are typically not considered equivalent, because variables of different structures would have to use different names to access member data

Type Equivalence (cont'd.)

- **Type names** in declarations may or may not be given explicitly
  - In C, variable declarations can use **anonymous types**
  - Names can also be given right in `structs` and `unions`, or by using a `typedef`
- Structural equivalence when type names are present can be done by simply replacing each name by its associated type expression in its declaration (except for recursive types)

Type Equivalence (cont'd.)

- Example: in C code
  - Variable `a` has two names: `struct RecA` and `RecA` (given by the `typedef`)
  - Variable `b` has only the name `RecB` (the `struct` name was left blank)
  - Variable `c` has no type name at all (only an internal name not usable by the programmer)

```
struct RecA{
    char x;
    int y;
} a;

typedef struct RecA RecA;

typedef struct{
    char x;
    int y;
} RecB;

RecB b;

struct{
    char x;
    int y;
} c;
```

Type Equivalence (cont'd.)

- Structural equivalence by replacing names with types can lead to infinite loops in a type checker when applied to recursive types

```
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode{
   char data;
   CharList next;
};

struct CharListNode2{
   char data;
   CharList2 next;
};
```

Type Equivalence (cont'd.)

- **Name equivalence**: two types are the same only if they have the same name
  - Easier to implement than structural equivalence, as long as every type has an explicit name
  - Two types are equivalent only if they are the same name
  - Two variables are type equivalent only if their declarations use exactly the same type name

Type Equivalence (cont'd.)

- Example: in C code:
  - a, b, c, and d are structurally equivalent
  - a and c are name equivalent, and not name equivalent to b or d
  - b and d are not name equivalent to any other variable

```
struct RecA{
   char x;
   int y;
};

typedef struct RecA RecA;

struct RecA a;
RecA b;
struct RecA c;
struct{
   char x;
   int y;
} d;
```

Type Equivalence (cont'd.)

- Ada implements a very pure form of name equivalence
  - Requires type names in variable and function declarations in virtually all cases
- C uses a form of type equivalence that falls between name and structural equivalence:
  - Name equivalence for `structs` and `unions`
  - Structural equivalence for everything else
- Pascal is similar to C, except that almost all type constructors lead to new, inequivalent types

Type Equivalence (cont'd.)

- Java's approach is simple:
  - It has no `typedefs`
  - `class` and `interface` declarations implicitly create new type names, and name equivalence is used for these types
  - Arrays use structural equivalence, with special rules for establishing base type equivalence

Type Checking

- **Type checking**: the process by which a translator verifies that all constructs are consistent
  - Applies a type equivalence algorithm to expressions and statements
  - May vary the use of the type equivalence algorithm to suit the context
- Two types of type checking:
  - **Dynamic**: type information is maintained and checked at runtime
  - **Static**: types are determined from the text of the program and checked by the translator

Type Checking (cont'd.)

- In a strongly typed language, all type errors must be caught before runtime
  - These languages must be statically typed
  - Type errors are reported as compilation error messages that prevent execution
- A language definition may not specify whether dynamic or static typing is used

Type Checking (cont'd.)

- Example1:
  - C compilers apply static type checking during translation, but C is not strongly typed since many inconsistencies do not cause compilation errors
  - C++ adds strong type checking, but mainly in the form of compiler warnings rather than errors, which do not prevent execution

Type Checking (cont'd.)

- Example 2:
  - Scheme is a dynamically typed language, but types are rigorously checked
  - Type errors cause program termination
  - No types in declarations and no explicit type names
  - Variables have no predeclared types, but take on the type of the value they possess

Type Checking (cont'd.)

- Example 3:
  - Ada is a strongly typed language
  - All type errors cause compilation error messages
  - Certain errors, like range errors in array subscripting, cannot be caught prior to execution
  - Such errors cause exceptions that will cause program termination if not handled by the program

Type Checking (cont'd.)

- **Type inference**: types of expressions are inferred from the types of their subexpressions
  - Is an essential part of type checking
- Type-checking rules and type inference rules are often intermingled
  - They also have a close interaction with the type equivalence algorithm
- Type inference and correctness rules are one of the most complex parts of the semantics of a language

Type Compatibility

- Two different types that may be considered correct when combined in certain ways are called **compatible**
  - In Ada, any two subranges of the same base type are compatible
  - In C and Java, all numeric types are compatible (and conversions are performed)
- **Assignment compatibility**: the left and right sides of an assignment statement are compatible when they are the same type
  - Ignores that the left side must be an **l-value** and the right side must be an **r-value**

Type Compatibility (cont'd.)

- Assignment compatibility can include cases where both sides do not have the same type
- In Java, `x=e` is legal when `e` is a numeric type whose value can be converted to the type of `x` without loss of information

Implicit Types

- **Implicit types**: types that are not explicitly given in a declaration
  - The type must be inferred by the translator, either from context information or from standard rules
- In C, variables are implicitly integers if no type is given, and functions implicitly return an integer value if no return type is given
- In Pascal, named constants are implicitly typed by the literals they represent
- Literals are the major example of implicitly typed entities

Overlapping Types and Multiply-Typed Values

- Two types may overlap, with values in common
- Although preferable for types to be disjoint, this would eliminate the ability to create subtypes through inheritance in object-oriented languages
- In C, types like `unsigned int` and `int` overlap
- In C, the literal `0` is a value for every integral type, a value of every pointer type, and represents the null pointer
- In Java, the literal value `null` is a value of every reference type

Shared Operations

- Each type is associated, usually implicitly, with a set of operations
- Operations may be shared among several types or have the same name as other operations that may be different
- Example: + operator can be real addition, integer addition, or set union
- **Overloaded operation**: the same name is used for different operations
  - Translator must decide which operation is meant based on the types of the operands

Type Conversion

- **Type conversion**: converting from one type to another
  - Can be built into the type system to happen automatically
- **Implicit conversion** (or **coercion**): inserted by the translator
- **Widening conversion**: target data type can hold all of the converted data without loss of data
- **Narrowing conversion**: conversion may involve a loss of data

Type Conversion (cont'd.)

- Implicit conversion:
  - Can weaken type checking so that errors may not be caught
  - Can cause unexpected behavior if the conversion is done in a different way than the programmer expects
- **Explicit conversion** (or **cast**): conversion directives are written into the code
  - Conversions are documented in the code
  - Less likelihood of unexpected behavior
  - Makes it easier for the translator to resolve overloading

Type Conversion (cont'd.)

- Example In C++:

```
double max (int, double);
double max (double,int);
max(2,3)
```

  - Ambiguous, because of the possible implicit conversions from `int` to `double` on either first or second parameter
- Java only permits widening implicit conversions for arithmetic types
- C++ emits warning messages for narrowing

Type Conversion (cont'd.)

- Explicit casts need to be somewhat restricted
  - Often to simple types, or just arithmetic types
- If casts are permitted for structured types, they must have identical sizes in memory
  - Allows translation to **reinterpret** the memory as a different type
- Example: in C, `malloc` and `free` functions are declared using a generic pointer or **anonymous pointer type** `void*`
- Object-oriented languages allow conversions from subtypes to supertypes and back in some cases

Type Conversion (cont'd.)

- Alternative to casts is to use predefined or library functions to perform conversions
  - Ada uses **attribute functions** to allow conversions
  - Java contains functions like `toString` to convert from `int` to `String` and `parseInt` to convert from `String` to `int`
- Undiscriminated unions can hold values of different types
  - With no discriminant or tag, a translator cannot distinguish values of one type from another

Polymorphic Type Checking

- Most statically typed languages required that explicit type information be given for all names in declarations
- It is possible to determine types of names without explicit declaration:
  - Can collect information on the uses of a name and infer the type from the set of all uses
  - Can declare a type error because some of the uses are incompatible with others
- This type inference and type checking is called **Hindley-Milner type checking**

Polymorphic Type Checking (cont'd.)

- Example in C code: `a[i] + i`
  - `a` must be declared as an array of integers, and `i` as an integer, giving an integer result
- Type checker starts out with this tree:

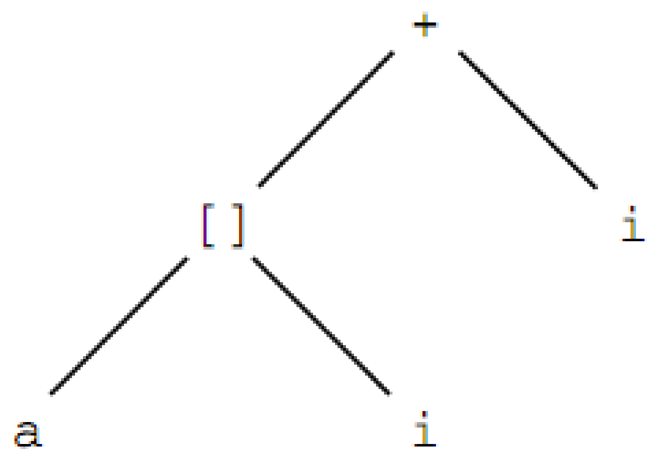


Figure 8.8 Syntax tree for a[i] + i

Polymorphic Type Checking (cont'd.)

- Types of the names (leaf nodes) are filled in from declarations

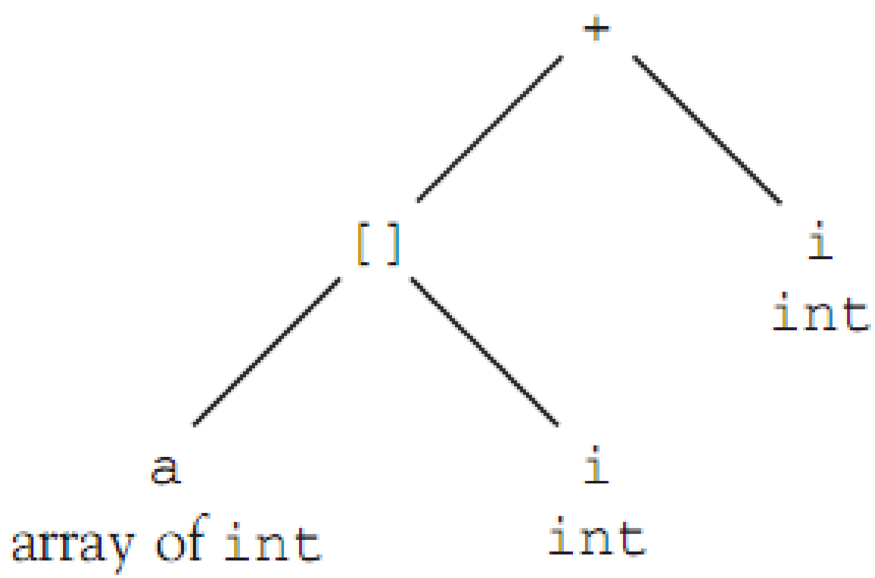


Figure 8.9 Types of the names filled in from the declarations

Polymorphic Type Checking (cont'd.)

- Type checker now checks the subscript node (labeled `[]`)
  - Left operand must be an array
  - Right operand must be an `int`
  - Inferred type of the subscript node is the component type of the array - `int`

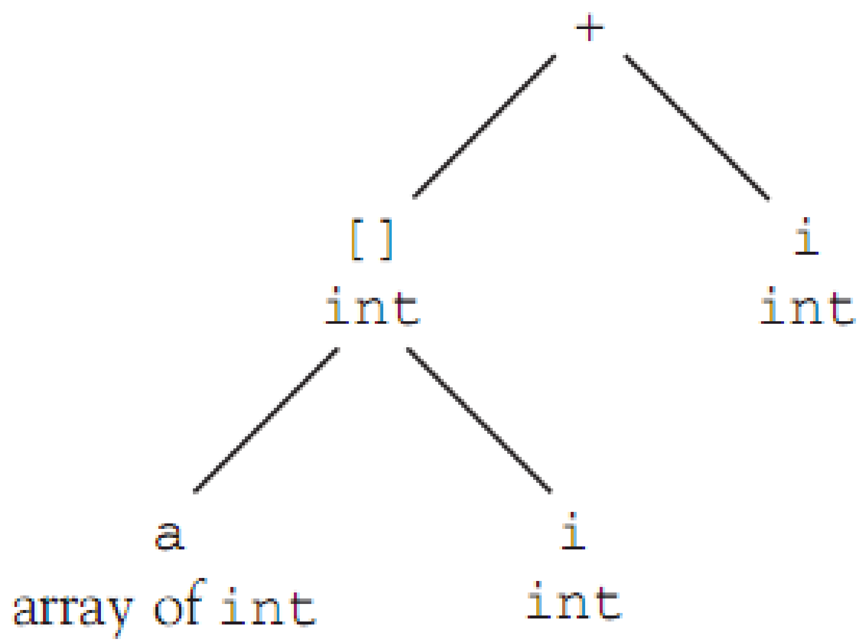


Figure 8.10 Inferring the type of the subscript node

Polymorphic Type Checking (cont'd.)

- + node type is checked
  - Both operands must have the same type
  - This type must have a + operation
  - Result is the type of the operands - `int`

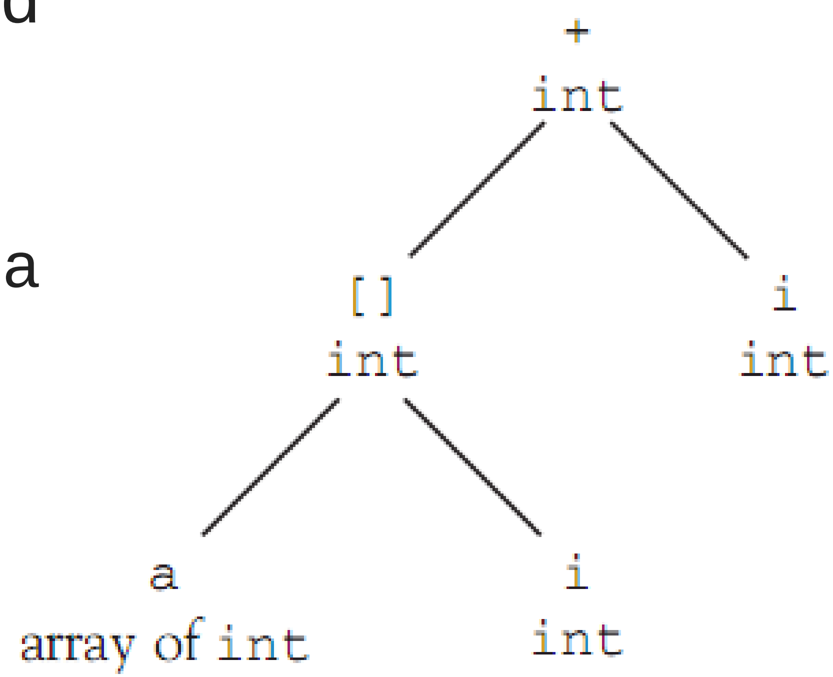


Figure 8.11 Inferring the type of the + node

Polymorphic Type Checking (cont'd.)

- Example: in C code:

```
a[i] + i
```

  – What if the declarations of a and `i` were missing?
- Type checker would first assign type variables to all names that do not yet have types

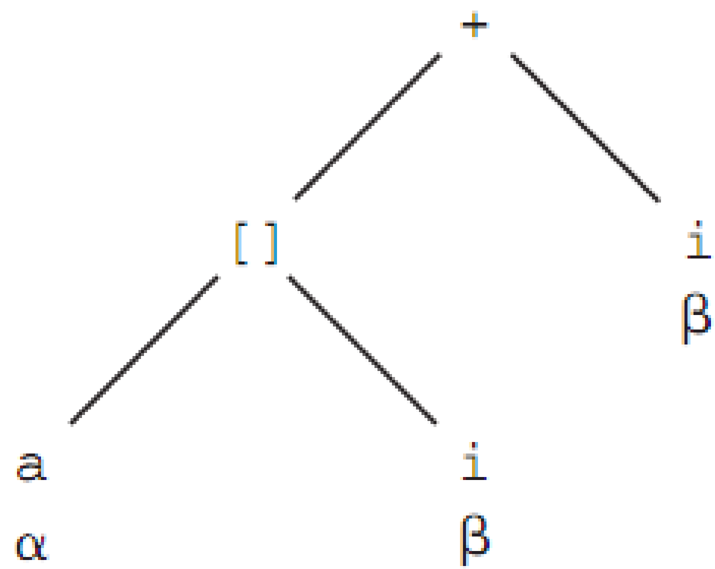


Figure 8.12 Assigning type variable to names without types

Polymorphic Type Checking (cont'd.)

- Type checker now checks the subscript node
  - Infers that a must be an array
  - Infers that `I` must an `int`
  - Replaces β with `int` in the entire tree

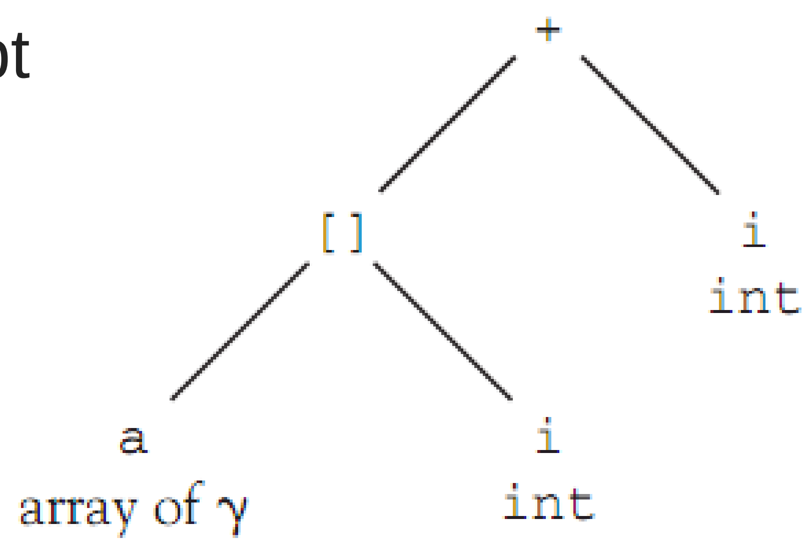


Figure 8.13 Inferring the type of the variable a

Polymorphic Type Checking (cont'd.)

- Type checker now concludes that the subscript node is type correct and has the type γ

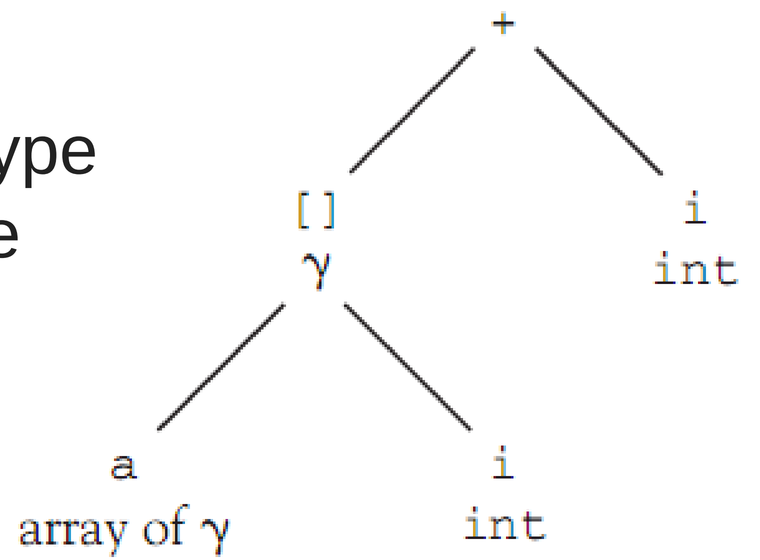


Figure 8.14 Inferring the type of the subscript node

Polymorphic Type Checking (cont'd.)

- + node type is checked
  - Concludes that γ must be type `int`
  - Replaces γ everywhere by `int`
- This is the basic form of operation of Hindley-Milner type checking

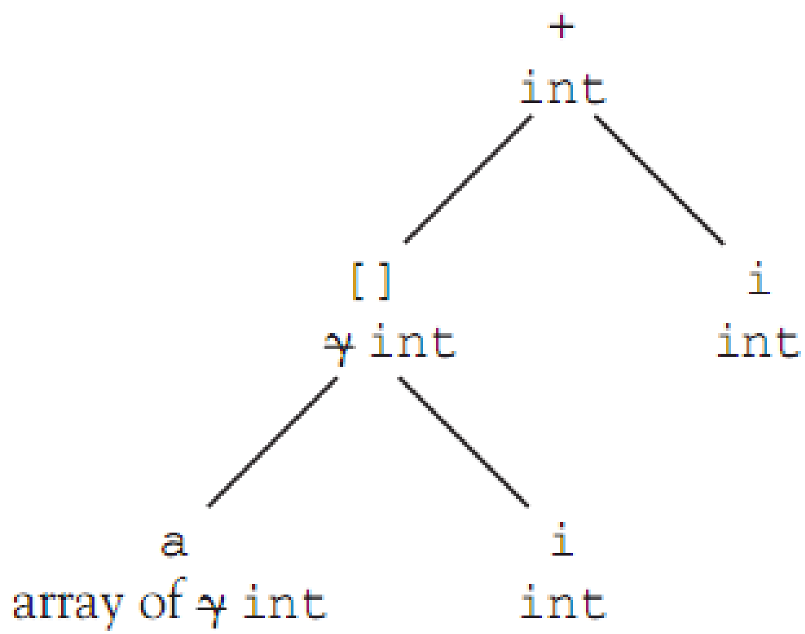


Figure 8.15 Substituting int for the type variable

Polymorphic Type Checking (cont'd.)

- Once a type variable is replaced by an actual type, all instances of that variable name must be updated with the new type
  - Called **instantiation** of type variables
- **Unification**: when type expressions for variables can change for type checking to succeed
  - Example array of α and array of β: we need to have $\alpha == \beta$, so β must be changed to α everywhere it occurs
  - Is a kind of pattern matching

Polymorphic Type Checking (cont'd.)

- Unification involves three cases:
  - Any type variable unifies with any type expression (and is instantiated to that expression)
  - Any two type constants unify only if they are the same type
  - Any two type constructions (such as array or struct) unify only if they are applications of the same type constructor and all of their component types also recursively unify

Polymorphic Type Checking (cont'd.)

- Hindley-Milner type checking advantages:
  - Simplifies the amount of type information the programmer must write
  - Allows types to remain as general as possible while still being strongly checked for consistency
- Hindley-Milner type checking implicitly implements polymorphic type checking
- Array of α is a set of infinitely many types, called **parametric polymorphism**
  - Hindley-Milner uses **implicit parametric polymorphism**

Polymorphic Type Checking (cont'd.)

- Sometimes called **ad hoc polymorphism** to distinguish it from overloading
- **Pure polymorphism** (or **subtype polymorphism**): when objects that share a common ancestor also either share or redefine operators that exist for the ancestor
- **Monomorphic**: describes a language that exhibits no polymorphism

Polymorphic Type Checking (cont'd.)

- Polymorphic functions are real goal of parametric polymorphism and Hindley-Milner type checking
- Example:

```
int max (int x, int y){
   return x > y ? x : y;
}
```

  - Body is the same if `int` is replaced by any other arithmetic type
  - Could add a new parameter representing the >

```
int max (int x, int y, int (*gt)(int a,int b) ) {
   return gt(x,y) ? x : y;
}
```

Polymorphic Type Checking (cont'd.)

- In C-like syntax:

```
max (x, y, gt) {
   return gt(x,y) ? x : y;
}
```

- In ML legal syntax, this becomes:

```
fun max (x, y, gt) = if gt(x,y) then x else y;
```

Polymorphic Type Checking (cont'd.)

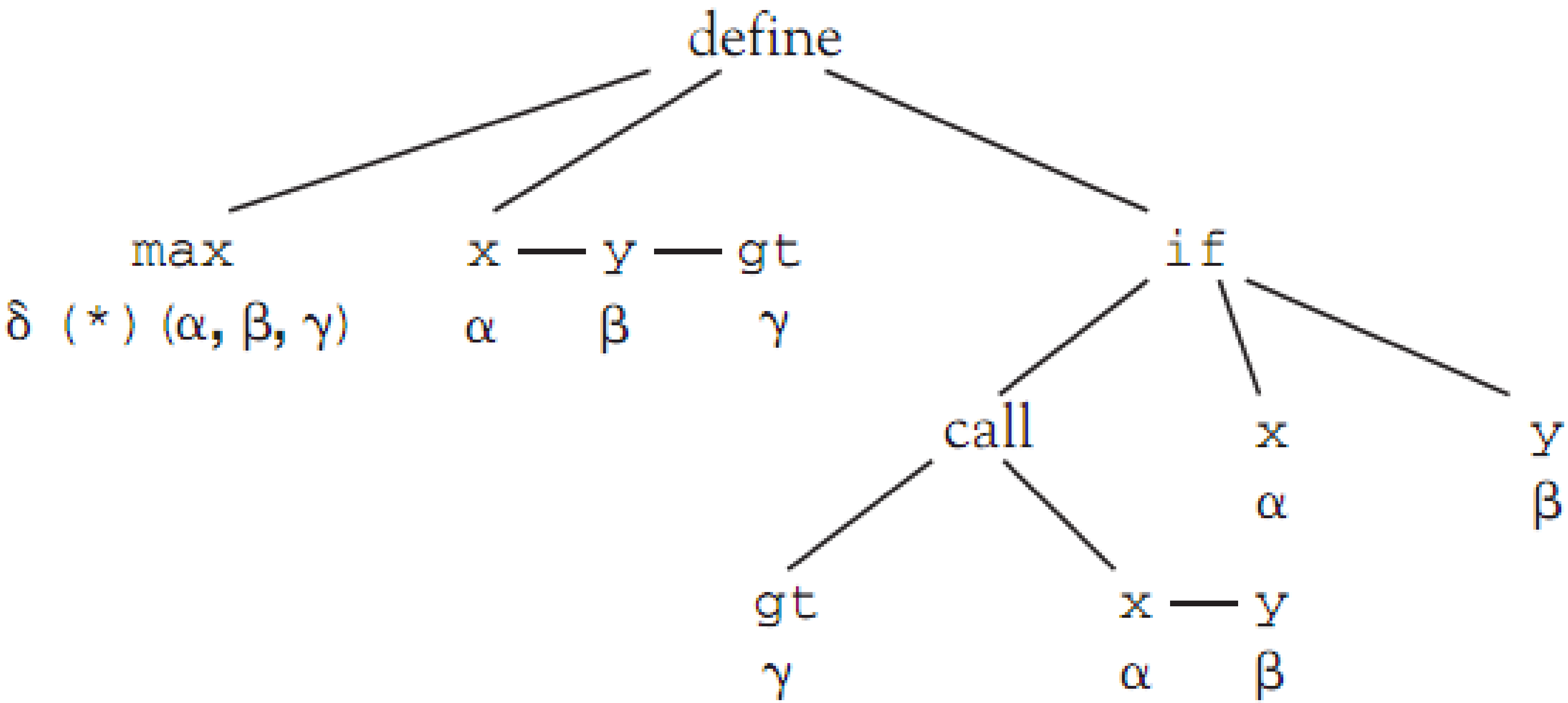


Figure 8.16 A syntax tree for the function max

Polymorphic Type Checking (cont'd.)

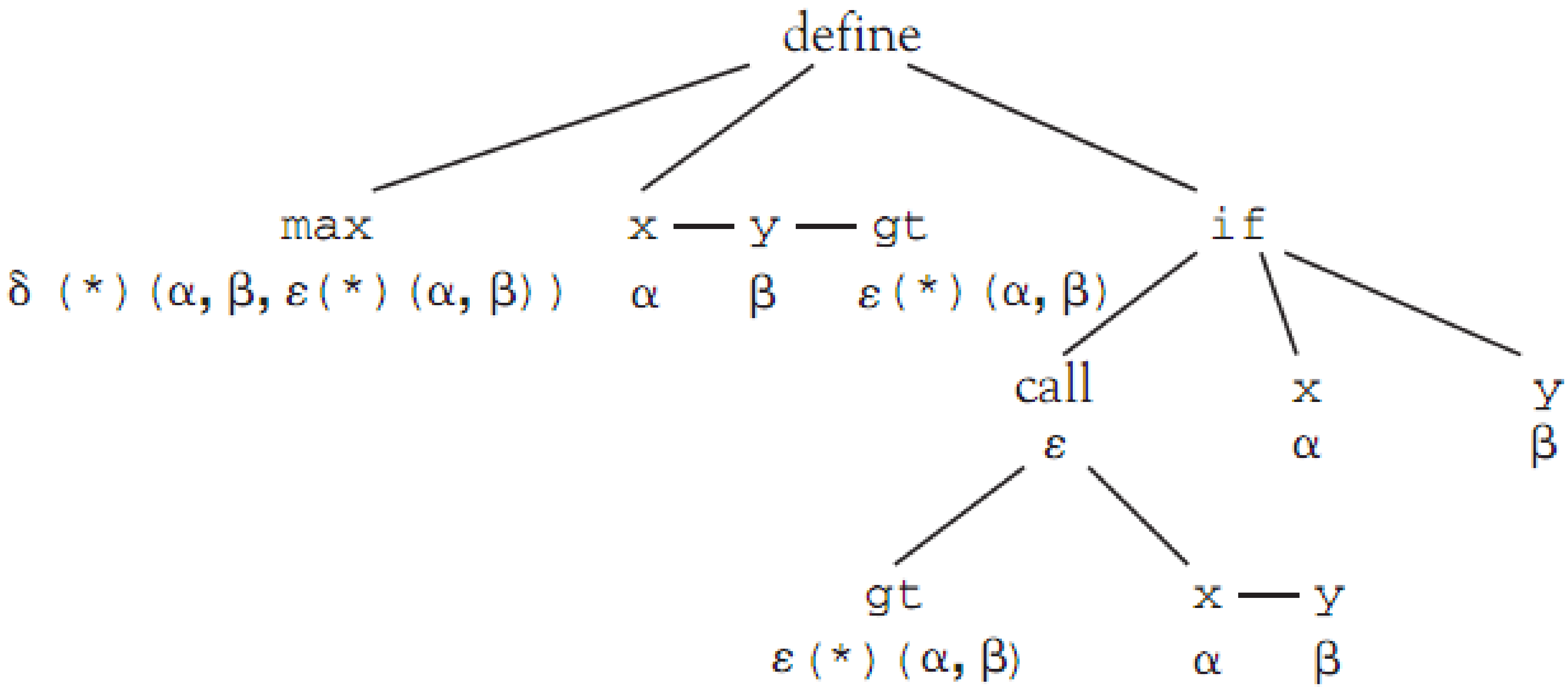


Figure 8.17 Substituting for a type variable

Polymorphic Type Checking (cont'd.)

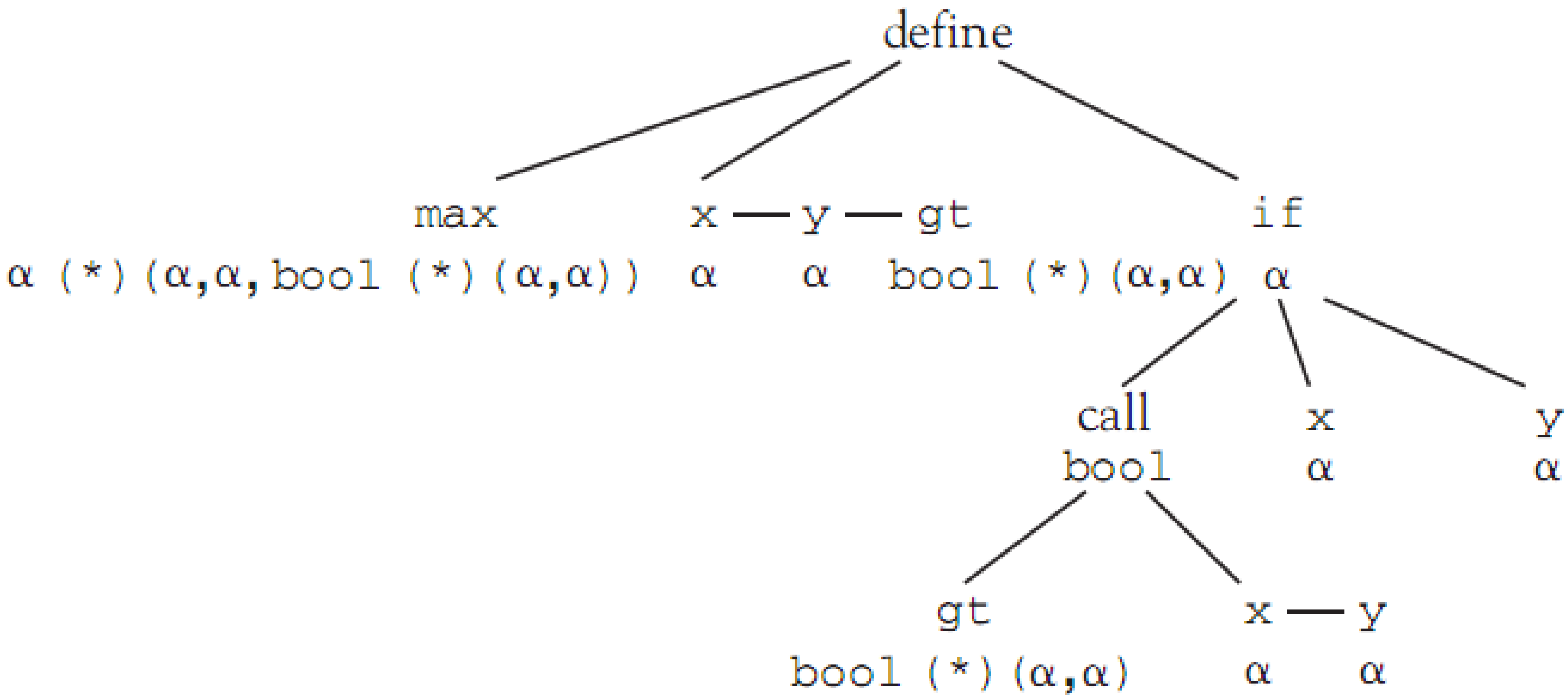


Figure 8.18 Further substitutions

Polymorphic Type Checking (cont'd.)

- Can now use max in any situation where the actual types unify
- If we provide these definitions in ML:

```
fun gti (x:int,y) = x > y;
fun gtr (x:real,y) = x > y;
fun gtp ((x,y),(z,w)) = gti (x,z);
```

  – We can call `max` function as follows:

```
max(3,2,gti); (* returns 3 *)
max(2.1,3.2,gtr); (* returns 3.2 *)
max((2,"hello"),(1,"hi"),gtp); (* returns (2,"hello") *)
```

Polymorphic Type Checking (cont'd.)

- Most general type possible for `max` function, called its **principal type**, is: `α (*)(α, α, bool (*)(α,α))`
- Each call to `max` **specializes** this principle type to a monomorphic type
  - May also implicitly specialize the types of the parameters
- Any polymorphically typed object passed into a function as a parameter must have a fixed specialization for the duration of the function
  - This restriction is called **let-bound polymorphism**

Polymorphic Type Checking (cont'd.)

- Two problems complicate Hindley-Milner type checking:
  - Let-bound polymorphism
  - The occur-check problem
- Polymorphic types also have translation issues
  - Copying values of arbitrary type without knowing the type means the translator cannot determine the size of the values
  - May cause **code bloat**

Explicit Polymorphism

- **Explicit parametric polymorphism**: to define a polymorphic data type, the type variable must be written explicitly
- Example: stack declaration in ML code

```
datatype 'a Stack = EmptyStack
                  | Stack of 'a * ('a Stack);
```

  – Values of type `Stack` can be written as:

```
val empty = EmptyStack; (* empty has type 'a Stack *)
val x = Stack(3, EmptyStack); (*x has type int Stack *)
```

Explicit Polymorphism (cont'd.)

- Explicitly parameterized polymorphic data types are nothing more than a mechanism for creating **user-defined type constructors**
  - A type constructor is a function from types to types
- Construction can be expressed directly in C as a `typedef`
- In ML, this is done with the `type` construct
- C++ is a language with explicit parametric polymorphism, but without the associated implicit Hindley-Milner type checking
  - Uses the **template** mechanism

Explicit Polymorphism (cont'd.)

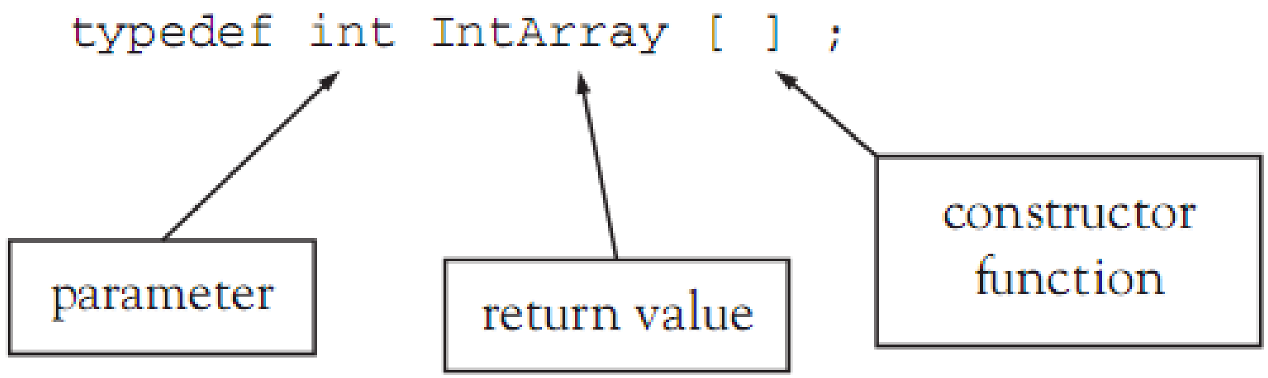


Figure 8.19 The components of a type definition in C

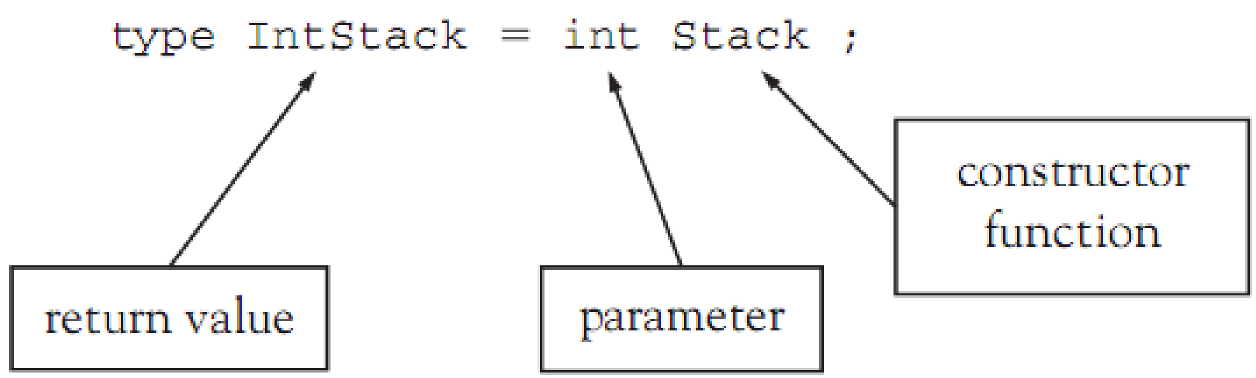


Figure 8.20: The components of a type definition in ML

Explicit Polymorphism (cont'd.)

- **Implicitly constrained parametric polymorphism**: implicitly applies a constraint to the type parameter
- **Explicitly constrained parametric polymorphism**: makes explicit what types of parameters are required

Case Study: Type Checking in TinyAda

- Goals:
  - Check identifiers to ensure that they are declared before they are used
  - Check that identifiers are not declared more than once in the same block
  - Record the role of an identifier as a constant, variable, procedure, or type name

Type Compatibility, Type Equivalence, and Type Descriptors

- TinyAda parser must:
  - Check that the type of an operand is appropriate for the operation being performed
  - Check that the name on the left side of an assignment statement is type-compatible with the expression on the right side
  - Restrict the types of certain elements of declarations, such as the index types of an array

Type Compatibility, Type Equivalence, and Type Descriptors (cont'd.)

- TinyAda uses a loose form of name equivalence to determine type compatibility
  - For arrays and enumerations, two identifiers are type-compatible if and only if they were declared using the same type name in their declarations
  - For built-in types `INTEGER`, `CHAR`, and `BOOLEAN` and their programmer-defined subrange types, two identifiers are type-compatible if and only if their supertypes are name-equivalent

Type Compatibility, Type Equivalence, and Type Descriptors (cont'd.)

- **Type descriptor**: primary data structure used to represent type attributes
- Type descriptor is entered into the symbol table when the type name is introduced
  - At startup for built-in type names `INTEGER`, `CHAR`, and `BOOLEAN`
  - Whenever new type declarations are encountered

The Design and Use of Type Descriptor Classes

- Type descriptor is like a variant record, containing different attributes depending on the category of the data type being described
- Each descriptor includes a type form field, with possible values of `ARRAY`, `ENUM`, `SUBRANGE`, and `NONE`, to identify the category of the data type
- Array type descriptor includes attributes for index types and element types (these attributes are also type descriptors)
- Enumeration type descriptor includes a list of symbol entries for the enumerate constant names

The Design and Use of Type Descriptor Classes (cont'd.)

- Type descriptors for subrange types (including `INTEGER`, `CHAR`, and `BOOLEAN`) include values of lower and upper bound and a type descriptor for the supertype
- There is no variant record structure in Java
  - Can model it with a `TypeDescriptor` class and three subclasses: `ArrayDescriptor`, `SubrangeDescriptor`, and `EnumDescriptor`

Entering Type Information in Declarations

- Type information must be entered wherever identifiers are declared in a source program
- Type information comes from type identifiers or from a type definition
  - Type identifiers: type descriptor is available in the identifier's symbol entry
  - Type definition: a new type might be created

Checking Types in Operands in Expressions

- The rules for TinyAda expressions give hints as to how their types should be checked
- The type of every operand must be checked, and the correct type descriptor must be returned

Processing Names: Indexed Component References and Procedure Calls

- Syntax for TinyAda indexed component references and procedure calls is the same if the procedure expects at least one parameter
  - Must distinguish between these two types of phrases, based on the role of the leading identifier

Completing Static Semantic Analysis

- Two other types of semantic restrictions can be imposed during parsing:
  - Checking of parameter modes
  - Check that only static expressions are used in number declarations and range type definitions
- Tanya has three parameter modes:
  - Input only: with the keyword `in`
  - Output only: with the keyword `out`
  - Input/output: with the keywords `in out`