Chapter 7 – Type Theory

Introduction to Programming Languages

First Edition, 2013

Author: Arvind Bansal

ISBN: 978-146-6565142

Topics Covered

Introduction

- Types and their role
  - Types are sets of objects with well defined properties and operations
  - A member of that set will follow the associated properties and operations
  - Type declarations provide better error correction, memory allocation, precision, and computational efficiency
- Types can be static types or dynamic types
  - Static types are declared at compile time. Explicit type information is lost after compilation in static types
  - Dynamic types can associate type with an identifier at runtime
  - Strongly typed languages do not alter a type of an identifier
- Type violation
  - When a type of object is treated as another type of object due to some programming language property or construct
- Polymorphism (to be explained later)
  - Allows a subset of operations on different types of objects

Advantages of Type Declaration I

- Error correction
  - Type mismatches are identified at compile time
- Optimized memory allocation
  - Different objects are allocated different amount of memory. Compile time declaration allows this optimized allocation at compile time
  - Optimized memory allocation also facilitates optimized computation as there is no need of extra memory processing
- Compile-time type conversion of operands
  - Integers and real numbers can be mixed in arithmetic operations
  - The operands are coerced to other type at compile time.
- Compile-time disambiguation of operators
  - Arithmetic operators are disambiguated when mixing integers and reals
- Code optimization
  - Knowledge of type declaration allows the effective use of registers and the removal of unnecessary code

Advantages of Type Declaration II

- Extra precision for numbers
  - Type declaration allows operations needing **large numbers or extra precision** that require more memory
- Software refinement and maintenance
  - User defined types allow easy incorporation and modification of data structures facilitating software maintenance
- Concurrent execution
  - Declaration of semaphores and monitors used for synchronization
- Use of the generic polymorphic procedures
  - Declaration of generic procedures and generic type allows for a function to be used for different types of data objects
  - Example of such operation is funding length of a list
- Disadvantages: 1) difficult to track in large programs; 2) not user-friendly due to large number of variables

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Example

```
program main                                                          %(1)
struct galaxy {                                                       %(2)
integer starCount;                                                    %(3)
double float distance; }                                              %(4)
{ integer x, y; % integer takes 4 bytes on a 32 bit machine           (5)
  float w, z;  % float takes 8 bytes on a 32 bit machine              (6)
  double a, b;  % takes 8 bytes on a 32 bit machine                   (7)
  string  c, d;                                                       %(8)
  galaxy neighbors[10];                                               %(9)
   x = 4; y = 6;                                                      %(10)
   w = x + y; % '+' is integer addition; the evaluation is coerced to float   (11)
    z = w + y; % '+' is a floating point addition; 'y' is coerced to float    (12)
   c = "Milky Way"                         %                          (13)
   d = z + c  % type mismatch error                                   (14)
   neighbors[1].starCount = 32567823418; % extra accuracy             (15)
   neighbors[1].distance = 4.5 E**12; }  %                            (16)
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Notion of Type

- Types are sets with well defined properties and operations
- Basic types
  - Mathematical types such as integers, floating point, Boolean, sets etc.
  - String processing types such as char, list
  - Computer organization information such as bit, byte, word, longword etc.
  - Synchronization primitives such as semaphore and monitor
- Declaring references to objects
- Structured types formed by joining multiple types
- Types can be
  - Passed as parameters as in parametric polymorphism
  - Declared as subtypes that follow the properties of original types, and is called inclusion type
  - An object can be transformed to an object of higher type without loss of information **– Coercion**
  - An operator or symbol may have multiple meanings - **overloading**

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Structured and Abstract Types

- Set operations generating new sets create new types
- Set operations are

Set Operations	Corresponding Types
Ordered sets	**Enumeration and subrange**
Cartesian product	**Record/struct/Tuple**
Finite mapping	**Arrays / Association list**
Disjoint union	**Variant record**
Power set	**Set of subsets**
Cartesian product + Disjoint union	**Recursive data types**

- Abstract data types impose additional properties and restrictions, and may have new operations on sets

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Cartesian Product ←→ Tuples

- Cartesian product produces a set of n-tuples of the form
  - $(a^1_{i,}\ a^2_{i,} \cdots, a^n_i)$ $(1 \le i \le n) \in S_1 \times S_2 \times \cdots \times S_n$
  - Size of the set is *size-of(S_1)* × *size-of(S_2)* × … × *size-of(S_n)*.
- composite data-entities are written using syntactic constructs 'struct' or 'record' , or tuples in different languages
  - Different fields correspond to different sets
- <u>Example</u>
  - Complex number: real × real
  - Rational number: integer × integer

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Finite Mapping ←→ Arrays

- Many-to-one mapping from Domain to Co-domain
- Corresponds to arrays, association lists, ordered sets
- Arrays are modeled as
  - Domain as natural numbers
  - Codomain as any data type
  - Example: integer a[3] has domain = {0, 1, 2}, codomain as set of integer values; and mapping as {0 → integer-value, 1 → integer value, 2 → integer value}
- Association list is modeled as
  - domain as enumeration type with each element as a key
- <u>Example</u>
  - Domain is {world-war-II, 1967, Earth }
  - Codomain is {1939, man-on-moon, water }
  - { world-war-II ↦ 1939; 1967 ↦ man-on-moon; and Earth ↦ water }

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Power Set ←→ Set Construct

- A set of all subsets of the original set
  - Number of elements in the set = $2^{\text{size-of(original set)}}$
- A variable is bound to any subset of the enumerable set S
- Basis of set based programming
  - one can define all the set operations on these subsets.
- <u>Example</u>
  - type student = (tom, phil, jean) % declaration of enumerated set
  - var regular_students : set of students;
  - The type student represents a power set of $2^3 = 8$ elements
  - {{ }, {tom}, {phil}, {jean}, {tom, phil}, {tom, jean}, {phil, jean}, {tom, phil, jean}}

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Disjoint union ←→ Variant Record

- Two sets S_1 and S_2 are disjoint if $S_1 \cap S_2 = \emptyset$
- Sets are colored using Cartesian product and mixed
  - Disjoint set = $\{Color_1\} \times S_1 \cup \{Color_2\} \times S_2$
- <u>Example</u>
  - Set_1 = *{Mary, Nina, Ambika, Susan}* ; Set_2 = *{Tom, Rubin, Mark}*
  - $Color_1$ = *girl;* $Color_2$ = *boy*
  - $Set_1 \uplus Set_2$ = *{girl}* × *{Mary, Nina, Ambika, Susan}* ∪ *{boy}* × *{Tom, Rubin, Mark}*
- Variant records: two parts - fixed part and variant part
  - Variant part is modeled as disjoint union as fields are selected based upon a multiple valued variable or a Boolean flag
  - Set = Fixed-part × {true} × variant-set_1 ∪ {false} × variant-set_2
- Problems with disjoint union
  - Different types of object may overlap on the same memory space
  - Incorrect operations on memory space due to type violation

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

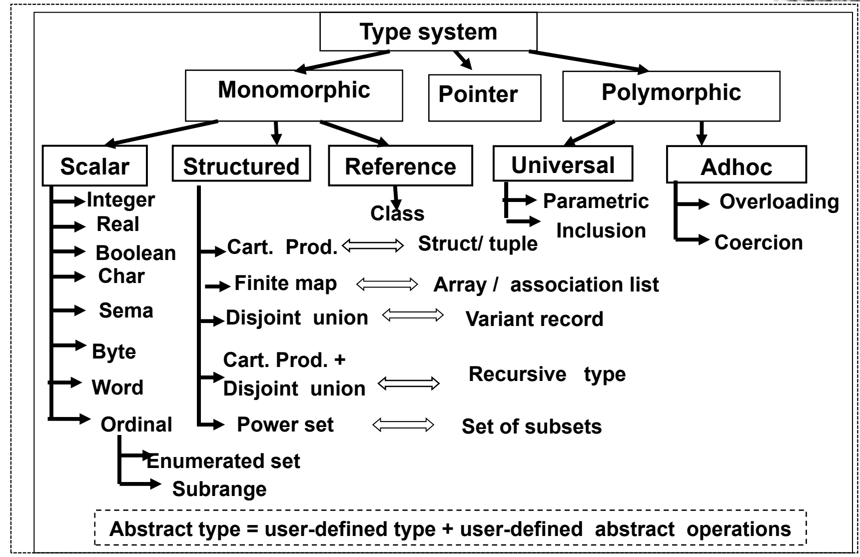
Recursive Data Types

- Modeled using *Cartesian product* and *disjoint union*
  - base case and recursive part connected through disjoint union
  - Concatenation in recursive part connected using Cartesian product
- List represents set of all lists of different sizes
  - Recursive definition gives the same set as the set operations involving Cartesian product and disjoint union
  - List = set of one element ⋃ set of two elements ⋃ …
  - List = {false} X {nil} ⋃ {true} X data-type X list (that can be expanded)
- Binary tree represents set of all trees of different depths
  - Recursive definition gives the same set as the set operations involving Cartesian product and disjoint union

Recursive definition	Set operation
<list> **::=** *<data> <list>* | **nil**	*<list>* **::=** *<data>* × *<list>* $\uplus$ *nil*
<bt> **::=** *<bt> <data> <bt>* | **nil**	*<bt>* **::=** *<bt>* × *<data>* × *<bt>* $\uplus$ nil

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Overall Type Structure



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Limitations of Type Theory

- Program properties that alter at runtime are not captured
  - Compiled code has no boundaries between data objects
- Example of runtime alterable properties
  - Array bound check: The index of an array element is computed at runtime, and can violate start and end marker
  - Substring of a string: start and length of the substring are computable at runtime, and can violate the overall size of a string.
  - Variant part of a variant record: Variant part's type interpretation is dependent upon the value of the flag that is altered at runtime.
  - Accessing elements in the data area using independent pointer that allow pointer arithmetic. Pointers can violate data-object boundary and program segment boundaries.
- Monomorphic types limit operations to one type of data objects

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Array Bound Check Problem

- **Program**
  - The variable j is multiplied by 2 every time for six times giving final value as 64
  - a[j] means nonexistent a[64]
  - Goes into memory space bound to some other variable and corrupts
- **Solution**
  - Perform array bound check before accessing any array element
- **Overhead**
  - Requires two additional operation for every array element access
  - Computationally very slow
- Vendors provide a compile time switch for executing programs with and without array-bound check

```
program main
integer i, j;
real a[50];
        ...
{ j = 1;
  for (i = 1; i <= 6; i++) j = 2 * j;  % j is 64
  a[j] = 120.2;  % A non-existent data element a[64] is being assigned a value.
  ...
}
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

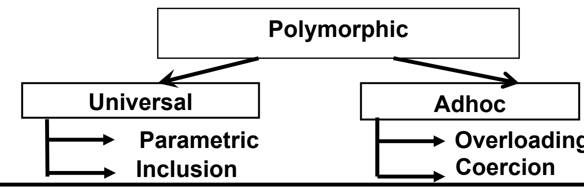
Substring of a String

- Program analysis
  - After the execution of for-loop, the value of j is 16
  - Length of the string is 6
  - Length of the substring is 4
  - Substring looks for substring of Arvind from position 16 of length 4
- Effect
  - Erroneous location after the substring allocation
- To correct this effect, string start and end needs to be carried and checked at runtime causing excessive overhead

```
program main
 { string my_name, short_name;
   integer i, j, k;
    my_name = "Arvind";  j = 1;
     for ( i = 0; i <= 3; I++) j = 2 * j;
          short_name =
            substring(my_name,  j,  4);
 }
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Polymorphism



- Polymorphism allows the same function or operation to work on multiple types of data objects.
  - Supports reusability of code
  - Universal polymorphism allows indefinite types of data objects
  - Adhoc polymorphism allows finite types of data objects
- Parametric polymorphism
  - Uses type variables, and passes concrete types as parameters.
- Inclusion polymorphism
  - Uses subset inclusion to inherit properties and operations of bigger set
- Coercion converts one type to another
- overloading allows multiple meaning to an operator / symbol

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Parametric Polymorphism

- Allows the use of generic functions on different types of objects
  - The operation is more associated with the structure of the data objects, rather than the property of individual data elements
  - Examples are adding list of integers; counting the elements in a list
  - Polymorphic type is written as input type → output type
- Mechanism of generic functions
  - Call subprogram is a generic function
  - Formal parameters of generic functions are expressed as type variables
  - Calling function passes the concrete type as parameters
  - Called function is specialized to specific type
- Examples
  - Polymorphic type of counting function: list(τ) → integer where list(τ) ::= $\tau \times$ list(τ) $\uplus$ nil
  - Polymorphic type of sum-of-a list: list(τ) → τ where $\tau \in$ { integer, real}
  - Polymorphic type of append is list(τ) x list(τ) → list(τ)

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Inclusion Polymorphism

- Any subset of an original type is a subtype
  - 2^N possibilities where N is the number of elements in the original set
  - For infinite size original set such as integer or real there are infinite possible subsets hence infinite number of subtypes
- Subtype inherits the properties and operations of the original type
  - No need to redefine the properties or operations for subtypes
  - Subclass inherits all properties and operations from parent class
- Limitations: closure property may be violated such as
  - Subtract(natural-number 1 – natural-number 2) is not necessary a natural-number despite natural-number being a subtype of integer
- <u>**Example**</u>
  - **subtype** Month is **INTEGER range** 1..12
  - **subtype** age is **INTEGER range** 0..150
  - **type** Weekday is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
  - **subtype** Workingdays is Weekday **range** Mon..Fri

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Coercion

- Automatic conversion to another type to support mixed types
- Conversion preserves information
- Transitive and antisymmetric
- Does not alter the original object
- Only consumer occurrences support coercion in statically typed languages

- Mechanism
  - Create a temporary location for the converted object
  - Perform the operation
- Integer → float → double float
- Integer → long integer → quad integer

Example

integer m, n;

float x, y;

double d1, d2;

{m = 4;   n = 6; x = 3.4; y = m + x; d1 = n + y; d2 = d1 + 5;}

- Explanation
  - M coerced in y = m + x from integer to float
  - N and y coerced to double float in d1 = n + y
  - 5 coerced to double float equivalent in d2 = d1 + 5

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Overloading

- Arithmetic operators such as '+', '*', '/', '-' have different meanings based upon operands
- Adhoc polymorphism

- Example
  - '+' can be integer addition, floating point addition, complex number addition, insertion of an element in a set etc.

- Disambiguation of operators
  - At compile-time In statically typed language
  - At runtime in dynamically typed language

Example

integer x, y;

float a, b;

…

x = 3; a = 5.3;

y = x + 6;

b = a + 7.4;

- **Explanation**
  - '+' in y = x + 6 is integer addition since both operands are integers
  - '+' is b = a + 7.4 is floating point addition since 7.4 is a floating point number

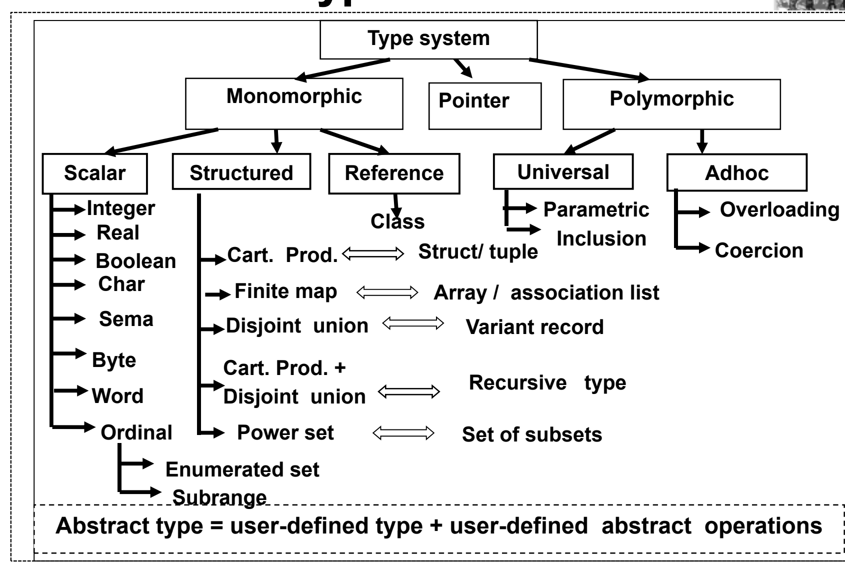
Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Type System in Modern Languages

- Support for both monomorphic and polymorphic type
  - Polymorphism includes universal and adhoc polymorphism
  - Monomorphic type supports scalar, structure and reference
- Further classification
  - Scalar types can be integer, float, Boolean, char, semaphores, byte, word, ordinal types, extra precision in integer and float, complex number
  - Structured types involve set operations: Cartesian products for tuple, ordered sets for sequences; finite mapping for arrays and association lists; disjoint types for unions / variant records; combination of Cartesian product and disjoint union for recursive data types
  - Reference types are used for objects and classes
  - Strings are sequences of characters. Have been treated as class in object oriented languages
- Pointers are treated differently in languages
  - Some languages do not support independent pointers for safety
  - Pointer arithmetic makes pointers unsafe

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Overall Type Structure



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142
Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

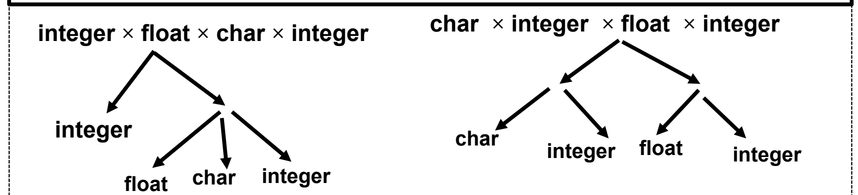
Universal Reference Type

- Object oriented languages support reference type to access objects stored in the heap
  - Reference is an internal representation that does not support pointer arithmetic or independent status like pointers
- Different languages name reference type differently
  - Java calls it object type; C++ and C# call it void *; CLU calls it any; Modula 3 calls it refany
- Object referred by universal reference type are altered dynamically
- Compiling universal reference type is unsafe
  - It can be associated with uncompatible type of objects at runtime
- Approaches to handle type compatibility at runtime
  - Casting – transforming one type of object to another. Two types of casting: **upward casting** and **downward casting**
  - Dynamic type tags – each data object keeps a type tag that is checked at runtime for compatibility before performing operation

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Type Equivalence

- Two types carrying same information should be equivalent
- Problem of equivalence is difficult because
  - Cartesian product is commutative
  - Same information may be grouped at different nesting level in a struct
  - Many fields may have the same type but different information
  - Difficult to align flexible base index in languages that support
  - Same basic type may represent incompatible information
- **Example: Same information with two different tree**
  - typedef struct { integer age; string name; float assignment_score;} student1;
  - typedef struct { string name; integer age;} person;
    typedef struct { person individual; float assignment_score;} student2;



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Structure vs. Name Equivalence

Structure Equivalence

- Based upon structural matching
- Structures are equivalent if they carry the same information
- Problems
  - Ambiguity by multiple fields of the same basic type
  - Commutativity of Cartesian product
  - Same type but different entity
  - Same information different nesting
- **Languages:** Modula 3, C and ML
  - Conservative approach
  - Two fields with same name and type are equivalent
  - Disallow permutation in the arrangement of fields, or
  - Permutation allowed if name and basic type matches

Name Equivalence

- Same name in addition to carrying the same information
- Restrictive but protects programmer's intention
- Easy to implement
  - the ease of type matching during compilation
- **Languages** : Ada, Pascal, Java, C# support name equivalence
- **Example**
  - **Type** Coordinate = **Record** x, y **: INTEGER**;
  - **Type** Complex = **Record** x, y **: INTEGER**;

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Implementation of Types

- Type information and various attributes are carried in symbol table
- In statically typed language, the attribute information is lost after compilation
  - The information is inherently compiled into operations in code area
- Implementation parts: Type descriptor and memory allocation
- Type descriptor during compilation
  - Name of the type
  - Type classification such as array, record etc.
  - Number of elements and bytes held by each element
- Tuples carry the information about
  - (tuple-name, number of fields , information about each specific field, number of bytes in each field, offset of each field).
- Arrays carry the information about
  - (array-name; number of elements; domain type, lower index, upper index; codomain type, bytes in each element, range of allowed values)

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Example of Type Descriptor

- **Student:** tuple of the form ((name: string, age: integer, major: string)
  - Type descriptor is of the form: (entity-type, name, bytes, total-size, individual field information)

```
(record, student, 3,  514,
        (string, name, 256, 0),  % information of field 1
        (integer, age, 2, 256), % information of field 2
        (string, major, 256, 258) % information of field 3
).
```

- **Class:** array [1..30] of student
  - Type descriptor is of the form: (entity-type, name, size, total bytes, domain information, codomain information)

```
(array, class, 30, 514,
        (integer, 1, 30),  % domain information
        (student, reference(student-descriptor))  % range information
)
```

- Code generator takes this information from type descriptor and embed it in the code area and the frame information

Type Inference and Checking

- Inference of implicit polymorphism is called type inference.
  - Languages like Prolog and Lisp have implicit polymorphism.
- Given explicit type, validating inferred types and declared types is called type checking.
- In statically typed language such as Scala type inference is used for inferring undeclared types.
- Polymorphic type components
  - Type variables declared as alpha, beta, gamma etc.
  - Concrete types such as integer, real, Boolean
  - Union of types, disjoint union of types; Cartesian product; mapping;
- Polymorphic type declaration
  - Input parameter type → output type
  - Multiple parameters are connected using Cartesian product $\alpha_1 \times \ldots \times \alpha_N$
  - Function composition $f \bullet g$ where input to g is α, output of g is β, and output of f is γ, then the polymorphic type of $f \bullet g$ is $(\alpha \rightarrow \beta) \rightarrow \gamma$
  - Values are converted to concrete type during inference

Polymorphism Example

Polymorphic type			
Function	**Type**	**Function**	**Type**
first	**list(α) → α**	**length**	**list(α) → integer**
rest	**list(α) → list(α)**	**append**	**list(α) × list(α) → list(α)**
cons	**α × list(α) → list(α)**	**insert**	**α × list(α) → list(α)**
null	**list(α) → Boolean**	**apply_all**	**(α → β) × list(α) → list(β)**

- **Example**

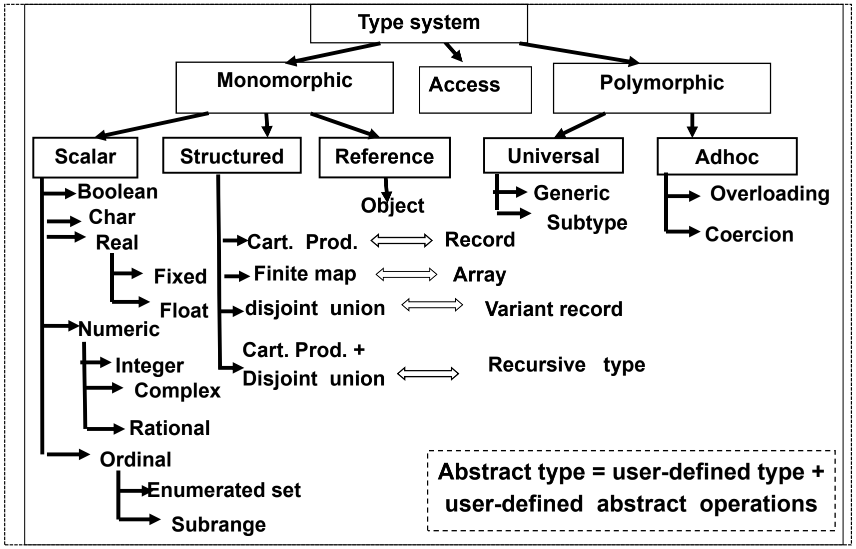
```
(defun  my_sum(Data)
    (if    (null   Data)  0  (+  (first    Data)  (my_sum  (rest    Data))))
)
```

  - Type inference starts with $\alpha \rightarrow \beta$
  - + operator limits the output of the function to integer or real
  - The value 0 further limits the output to the type integer
  - The function first sets input to list(integer)
  - **The final polymorphic type is list(integer) → integer**

Implementing Polymorphic Type

- Implementing polymorphism requires runtime specialization
  - Type variable requires runtime binding and specialization
- Source code can be translated differently to machine code
  - **Uniform polymorphism**: source code and machine code both exhibit polymorphism; same memory allocation scheme for different types of objects; memory allocation not optimized; extra effort and wastage to separate pointers from data
  - **Textual polymorphism**: polymorphism only is at source code level. Multiple specialized code at machine level. Excessive memory overhead for code area
  - **Tagged polymorphism**: data is represented differently for different types of objects; machine code uses uniform code; different dynamic sequence of code executed for different types of objects using type check and branch statements; used for operator overloading

Type System in Ada 2012



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Type System in Other Languages

- Type system in C++
  - Strongly typed language
  - Supports basic types such as integer, float, char, Boolean and string
  - Structured types such as struct, arrays, union, and recursive data types; reference type (class); and pointers
  - Supports different types of polymorphisms: parametric, inclusion, overloading, and coercion
  - Supports string as class library
- Type system in Modula – 3
  - Strongly statically typed language
  - Supports structural equivalence instead of name equivalence
  - Supports all structured types including set based constructs
  - Pointers as independent type
  - Supports objects as reference type
  - Supports procedure type, and procedures can be passed as parameters

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Summary I

- Type system is a classification system based upon well defined properties and operations.
- Types are sets, and user defined structural types are generated by operations and structures based upon set operations
  - The major set operations are*: ordered sets, ordered bags, Cartesian product, finite mapping, disjoint union, and power set*.
  - Cartesian Product ←→ tuple; Finite-mapping ←→ arrays; Power-set ←→ set declarations; ordered set ←→ enumeration type and range; disjoint union ←→ variant record; disjoint union and Cartesian product ←→ recursive data type
- **Reference types are used for object representation**
- Polymorphic types allow same operation or functions on multiple types of objects
  - Universal polymorphism: parametric and inclusion
  - Adhoc polymorphism: overloading and coercion

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Summary II

- The advantages of types are in
  - Identifying mismatch error, optimized memory allocation, extra precision, compile time overloading and coercion, user defined types for better software maintenance etc.
- The disadvantage of types is
  - Extra effort by the programmer to mentally keep track of variables, lack of reusability of variables for different types of objects.
- Statically typed languages loose type information after compilation
- Type descriptor is used to
  - Detect type mismatch, memory allocation, implicitly embed type information in code area by compiling to corresponding operations
  - Type descriptor includes information about attributes, number and size of the fields, size of the data elements etc.
- Universal reference type is used to implement objects in the heap
- Implementation of polymorphic types can be
  - Uniform, tagged, or textual

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142