Language Design & Implementation – 6CC509

Lecture 3: Values and Types

Ioannis Tsioulis

Language Design & Implementation – 6CC509

Lecture 3: Values and Types

Ioannis Tsioulis

slides acknowledgements:
D. Voorhis, U. of Derby
M. Scott, U. of Rochester
D. Watt, U. of Glasgow

Values and Types

Understand the Distinction: Dynamically vs Statically typed

- **First**… What are types?
- **Second**… What is type safety?

Understand the Distinction: Dynamically vs Statically typed

- For many Languages, this is true:
  - ***Type*** *= a (relatively) invariant set of values and zero or more associated operators.*
  - ***Operator*** *= a function or procedure that given one or more values, returns one or more values.*
  - ***Value*** *= an immutable instance of a given type, i.e., one of a set of values.*
  - ***Constant*** *= an invariant reference to a value.*
  - ***Variable/parameter*** *= a possibly time-varying reference to a value.*

http://wiki.c2.com/?TopsTypeDeterminatorChallenge

Understand the Distinction: Dynamically vs Statically typed

Handy rule of thumb:

- **Statically-typed languages** tend to associate types with variables and values (expressions). A value (expression) can only be assigned to a variable if their types are type compatible, and this can often be determined (at least to some degree) at compile-time.

- **Dynamically-typed languages** tend to associated types only with values (expressions). A value (expression) can be assigned to a variable unconditionally.

Understand the Distinction: Single vs Multiple-dispatch

- ***Single dispatch*** = conventional OO
  - Dynamically dispatch on the run-time type (as opposed to the declared or static type) of the (often implicit or invisible) first argument.
- ***Multiple dispatch*** = more powerful than conventional OO
  - Dynamically dispatch on the run-time type (as opposed to the declared or static type) of all arguments.
- Other forms of dispatch are possible…

Understand the Distinction: First vs Second-class

- Usually ...
  - ***First-class*** constructs can be defined by expressions and assigned to (or bound to) variables or parameters at run-time and returned by operators/functions.
  - ***Second-class*** constructs can’t.

Understand the Distinction: Mutable vs Immutable

- ***Mutable*** things can change. ***Immutable*** things can't.
  - Functional programming derives considerable benefit from avoiding mutability.
  - For example, if *variables can be assigned once and only once*, programs can become much easier to reason about.

Values and Types

- A **value** is any entity that can be manipulated by a program. Values can be evaluated, stored, passed as arguments, returned as function results, and so on.
- Different programming languages support different types of values:
  - C supports integers, real numbers, structures, arrays, unions, pointers to variables, and pointers to functions. (Integers, real numbers, and pointers are primitive values; structures, arrays, and unions are composite values.)
  - C++, which is a superset of C, supports all the above types of values plus objects. (Objects are composite values.)
  - JAVA supports booleans, integers, real numbers, arrays, and objects. (Booleans, integers, and real numbers are primitive values; arrays and objects are composite values.)
- Most programming languages group values into types.
- We define a **type** to be a set of values, equipped with one or more operations that can be applied uniformly to all these values.
  - Every programming language supports both **primitive types**, whose values are primitive, and **composite types**, whose values are composed from simpler values.

Primitive Types I

- A **primitive value** is one that cannot be decomposed into simpler values.
- A **primitive type** is one whose values are primitive.
- Every programming language provides built-in predefined primitive types. Some languages also allow programs to define new primitive types.
- **Boolean**, **Character**, **Integer**, and **Float** are the most common primitive types:
  - The Boolean type has exactly two values, false and true.
    - In some languages these two values are denoted by the literals **false** and **true**, in others by predefined identifiers **false** and **true**.
  - The Character type is a language-defined or implementation-defined set of characters.
    - The chosen character set is usually **ASCII** (128 characters), **ISOLATIN** (256 characters), or **UNICODE** (65 536 characters).
  - The Integer type is a language-defined or implementation-defined range of whole numbers. The range is influenced by the computer’s word size and integer arithmetic.
    - For instance, on a 32-bit computer with two’s complement arithmetic, **Integer** will be {−2 147 483 648, … ,+2 147 483 647}.
  - The Float type is a language-defined or implementation-defined subset of the (rational) real numbers.
    - The range and precision are determined by the computer’s word size and floating-point arithmetic.

Primitive Types II

Although nearly all programming languages support the Boolean, Character, Integer, and Float types in one way or another, there are many complications:

- Not all languages have a distinct type corresponding to Boolean.
  - For example, C++ has a type named **bool**, but its values are just small integers; there is a convention that zero represents false and any other integer represents true. This convention originated in C.
- Not all languages have a distinct type corresponding to Character.
  - For example, C, C++, and JAVA all have a type **char**, but its values are just small integers; no distinction is made between a character and its internal representation.
- Some languages provide not one but several integer types.
  - For example, JAVA provides **byte** {−128, . . . ,+127}, **short** {−32 768, . . . ,+32 767}, **int** {−2 147 483 648, . . . ,+2 147 483 647}, and **long** {−9 223 372 036 854 775 808, . . . ,+9 223 372 036 854 775 807}. C and C++ also provide a variety of integer types, but they are implementation defined.
- Some languages provide not one but several floating-point types.
  - For example, C, C++, and JAVA provide both **float** and **double**, of which the latter provides greater range and precision.

Composite Types

- A **composite value** is a value that is composed from simpler values.
- A **composite type** (or **data structure**) is a type whose values are composite.
- Programming languages support a huge variety of composite values: tuples, structures, records, arrays, algebraic types, discriminated records, objects, unions, strings, lists, trees, sequential files, direct files, relations, etc.
- The variety might seem bewildering, but in fact nearly all these composite values can be understood in terms of a small number of structuring concepts, which are:
  - Cartesian products (tuples, records)
  - Mappings (arrays)
  - Unions (algebraic types, discriminated records, objects)
  - Recursive types (lists, trees).

Composite Types I

- **Records** (structures) were introduced by Cobol, and have been supported by most languages since the 1960s.
  - A record consists of collection of fields, each of which belongs to a (potentially different) simpler type.
  - Records are akin to mathematical tuples; a record type corresponds to the Cartesian product of the types of the fields.
- **Variants** (unions) differ from "normal" records in that only one of a variant record's fields (or collections of fields) is valid at any given time.
  - A variant record type is the union of its field types, rather than their Cartesian product.
- **Arrays** are the most commonly used composite types.
  - An array can be thought of as a function that maps members of an index type to members of a component type.
  - Arrays of characters are often referred to as strings, and are often supported by special-purpose operations not available for other arrays.

Composite Types II

- **Enumerations** were introduced by Wirth in the design of Pascal.
  - They facilitate the creation of readable programs, and allow the compiler to catch certain kinds of programming errors.
  - An enumeration type consists of a set of named elements.
  - In Pascal, one can write:
    ```
    type weekday = (sun, mon, tue, wed, thu, fri, sat);
    ```
  - In C, the difference between the two approaches is purely syntactic:
    ```
    enum weekday {sun, mon, tue, wed, thu, fri, sat};
    ```
- **Subranges** were first introduced in Pascal, and are found in many subsequent languages.
  - A subrange is a type whose values compose a contiguous subset of the values of a base type (i.e., the parent type).
  - Subranges in Pascal look like this:
    ```
    type test_score = 0..100;
    ```
- **Sets**, like enumerations and subranges, were introduced by Pascal.
  - A set type is the mathematical powerset of its base type, which must often be discrete.
  - A variable of a set type contains a collection of distinct elements of the base type.
  - Pascal supports sets of any discrete type, and provides union, intersection, and difference operations:
    ```
    var A, B, C : set of char;
    ```

Composite Types III

- **Pointers** are l-values.
  - A pointer value is a reference to an object of the pointer’s base type.
  - Pointers are often but not always implemented as addresses. They are most often used to implement recursive data types.
  - A type T is recursive if an object of type T may contain one or more references to other objects of type T.
- **Lists**, like arrays, contain a sequence of elements, but there is no notion of mapping or indexing.
  - A list is defined recursively as either an empty list or a pair consisting of a head element and a reference to a sub-list.
  - While the length of an array must be specified at elaboration time in most (though not all) languages, lists are always of variable length.
  - To find a given element of a list, a program must examine all previous elements, recursively or iteratively, starting at the head.
  - Because of their recursive definition, lists are fundamental to programming in most functional languages.
- **Files** are intended to represent data on mass-storage devices, outside memory.
  - Files can be conceptualized as a function that maps members of an index type to members of a component type.

Type Systems

- A **type system** consists of
  1. a mechanism to define types and associate them with certain language constructs, and
  2. a set of rules for type equivalence, type compatibility, and type inference.

- **Type equivalence** rules determine when the types of two values are the same.
- **Type compatibility** rules determine when a value of a given type can be used in a given context.
- **Type inference** rules define the type of an expression based on the types of its constituent parts or (sometimes) the surrounding context.

Type Checking

- **Type checking** is the process of ensuring that a program obeys the language's type compatibility rules.
- A language is said to be **strongly typed** if it prohibits the application of any operation to any object that is not intended to support that operation.
- A language is said to be **statically typed** if it is strongly typed and type checking can be performed at compile time.
  - In practice, the term is often applied to languages in which most type checking can be performed at compile time, and the rest can be performed at run time.
- In the strictest sense of the term, few languages are statically typed. A few examples:
  - Ada is strongly typed, and for the most part statically typed (certain type constraints must be checked at run time).
  - A Pascal implementation can also do most of its type checking at compile time, though the language is not quite strongly typed.
  - Implementations of C rarely check anything at run time.
- **Dynamic (run-time) type checking** is a form of late binding, and tends to be found in languages that delay other issues until run time as well.
  - Lisp and Smalltalk are dynamically (though strongly) typed.
  - Most scripting languages are also dynamically typed; some (e.g., Python and Ruby) are strongly typed.

Types and Subroutines

- **Subroutines** are considered to have types in some languages, but not in others.
- Subroutines need to have types if they are first- or second-class values (i.e., if they can be passed as parameters, returned by functions, or stored in variables).
  - In each of these cases there is a construct in the language whose value is a dynamically determined subroutine; type information allows the language to limit the set of acceptable values to those that provide a particular subroutine interface (i.e., particular numbers and types of parameters).
- In a statically scoped language that never creates references to subroutines dynamically (one in which subroutines are always third-class values), the compiler can always identify the subroutine to which a name refers, and can ensure that the routine is called correctly without necessarily employing a formal notion of subroutine types.

Function Classes

- A programming language is said to have **first-class functions** if it treats functions as first-class citizens. This means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.
  - First-class functions are a necessity for the functional programming style, in which the use of higher-order functions is a standard practice.
- A language construct is **second-class** if it can be passed as a parameter but not be returned from a subroutine, or assigned to a variable.
  - Procedures are second-class in most imperative languages.
- A language construct is **third-class** if it can't even be passed as a parameter.

https://en.wikipedia.org/wiki/First-class_function

Dynamic Typing & Polymorphism

- **Polymorphism** allows a single body of code to work with objects of multiple types.
  - It may or may not imply the need for run-time type checking.
- Because the types of objects can be thought of as implied (unspecified) parameters, dynamic typing is said to support implicit parametric polymorphism.
- With rare exceptions, the programmer need not specify the types of objects explicitly.
  - The task of the compiler is to determine whether there exists a consistent assignment of types to expressions that guarantees, statically, that no operation will ever be applied to a value of an inappropriate type at run time.
  - As implemented in Lisp, Smalltalk, and the various scripting languages, fully dynamic typing allows the programmer to apply arbitrary operations to arbitrary objects.
  - Only at run time does the language implementation check to see that the objects actually implement the requested operations.
- Unfortunately, while powerful and straightforward, dynamic typing incurs significant run-time cost, and delays the reporting of errors.

Polymorphism and Generics

- In object-oriented languages, **subtype polymorphism** allows a variable X of type T to refer to an object of any type derived from T.
- Given a straightforward model of inheritance, type checking for subtype polymorphism can be implemented entirely at compile time.
- Most languages that envision such an implementation, including C++, Eiffel, Java, and C#, also provide explicit parametric polymorphism (generics), which allow the programmer to define classes with type parameters.
- **Generics** are particularly useful for container (collection) classes: “list of T” (List<T>), “stack of T” (Stack<T>), and so on, where T is left unspecified.
- Like subtype polymorphism, generics can usually be type checked at compile time, though Java sometimes performs redundant checks at run time for the sake of interoperability with pre-existing non-generic code.
- Smalltalk, Python, and Ruby use a single mechanism for both parametric and subtype polymorphism, with checking delayed until run time.

Dynamic Dispatch

- **Dynamic dispatch** occurs in a method call where the version of the method to be called cannot be determined at compile-time:
  - the target object could be either an object of a class that is equipped with the method, or
  - an object of a subclass that overrides the method.
- In most object-oriented systems, the concrete function that is called from a function call in the code depends on the dynamic type of a single object and therefore they are known as **single dispatch** calls, or simply **virtual** function calls.
- **Double dispatch** is a special form of **multiple dispatch**, and a mechanism that dispatches a function call to different concrete functions depending on the runtime types of two objects involved in the call.

https://en.wikipedia.org/wiki/Double_dispatch

Dynamic vs Static Typing

The growing popularity of scripting languages has led a number of prominent software developers to *publicly question the value of static typing*.

They ask: given that we can't check everything at compile time, how much pain is it worth to check the things we can?

As a general rule, it is easier to write type-correct code than to prove that we have done so, and static typing requires such proofs.

As type systems become more complex (due to object orientation, generics, etc.), the complexity of static typing increases correspondingly.

Anyone who has written extensively in Ada or C++ on the one hand, and in Python or Scheme on the other, cannot help but be struck at how much easier it is to write code, at least for modest-sized programs, without complex type declarations.

Dynamic checking incurs some run-time overhead, of course, and may delay the discovery of bugs, but this is increasingly seen as insignificant in comparison to the potential increase in human productivity.

The choice between static and dynamic typing promises to provide one of the most interesting language debates of the coming decade.

M. Scott (2009), Programming Language Pragmatics, 3d Ed., Morgan Kaufmann

Understand the Distinction: Dynamically vs Statically typed

Language Category S (Static)

Every variable is directly associated with, or has, a type. In other words, every variable has a "type" property.

The type of a variable can be determined by explicit declaration of its type (e.g., in C, C++, C#, Java): `int x;`
Or by type inference (e.g., in C#): `var x = 3;`

Every value is associated with, or has, a type. E.g, integer, float, string, or some user-defined type.

The type of a value may be inferred or explicitly specified.

The following, in most languages, will be inferred to be an integer: `23498`

The following, in most C-like languages, explicitly specifies that the value is of type "double": `(double)23498`

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Language Category S (Static)

(continued…)

Actions

1. **Assignment to a variable**: The variable's type is used to ensure that it can only be assigned values whose type is compatible. A language implementation does this by throwing an exception or generating an error if an attempt is made to assign a value whose type is not compatible with the variable's type. In some languages, "compatible" means the value's type must strictly be the same as the variable's type. In other languages, "compatible" means the value's type must be the same as or a subtype of the variable's type. Depending on the language, other definitions of "compatible" are possible. Compatible assignments of values replace the variable's current value.

2. **Invocation of operators**: Value types are used by the language to select the compatible specific operator when operators are polymorphic, and to ensure that argument types are compatible with operators' parameter types. See the "Operator Invocation" subsection below for more detail.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Language Category D1 (Dynamic)

Variables are not directly associated with a type. In other words, they do not have a "type" property.

Every value has a type, such as integer, float, string, or some user-defined type.
The type of a value may be inferred or explicitly specified.

PHP is a language in this category.

Actions

1. **Assignment to a variable**: Variables may be assigned any value of any type at any time.

2. **Invocation of operators**: Values' types are used by the language to ensure that the compatible specific operator -- if from a set of polymorphic operators -- is selected when invoked, and to ensure that values are compatible with operator parameters. Notably, many Category D1 languages do not expose this functionality to the language user, but it is used internally to (for example) dispatch the correct built-in "+" operator: the one implementing numeric addition if the operands are numeric, or the one implementing string concatenation if the operands are not numeric. See the "Operator Invocation" subsection below for more detail.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Language Category D2 (Dynamic)

Variables are not directly associated with a type. In other words, they do not have a "type" property.

Every value is represented by a string of characters.

ColdFusion and Perl are languages in this category.

Actions

1. **Assignment to a variable**: Variables may be assigned any value at any time.

2. **Invocation of operators**: Operators perform parsing as needed to determine whether each argument value (which is a string of characters) represents an integer, number, date, etc. See the "Operator Invocation" subsection below for more detail.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Notes on the Preceding Slides

In all three language categories, the only observable run-time state change is via variable assignment.

There is **no** other explicit state change.

In all three language categories, operators may interpret their operands as they see fit, including recognising values of various types that may be encoded within their operand values. For example, in PHP, the "is_numeric()" operator may be used to test whether or not its operand is numeric, which can include both operands of numeric type and numeric strings. (E.g., 123 is of numeric type, "123" is a numeric string.)

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Operator Invocation (Part 1)

In most languages, a given operator name can have multiple operator definitions -- one operator definition for each combination of operand types. Each operator definition has its own operator signature, which consists of its name, operand types, and possibly its return type. For example:

+(integer, integer) returns integer
+(string, integer) returns string
+(integer, string) returns string
+(string, string) returns string

The interpreter (or compiler) uses the operand types to determine which definition to invoke. For example, an expression like 34 + "34" can be written as +(34, "34"). Replacing the operand values with their types, we get +(integer, string). That matches the third signature above, and so we invoke its corresponding operator definition.

The definitions determine the language behaviour. For example, in the above, only the first definition performs addition. The other three perform string concatenation. Thus, the language would return 68 given 34 + 34 and "3434" given 34 + "34" or "34" + 34.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Operator Invocation (Part 2)

Given the following:

+(integer, integer) returns integer
+(string, integer) returns integer
+(integer, string) returns integer
+(string, string) returns string

The first three definitions attempt addition, the second and third parsing strings to determine if they represent numeric values. The last definition performs string concatenation. Thus, the language would return 68 given 34 + 34, 68 given 34 + "34", "3434" given "34" + "34", and an error given 34 + "splat" because "splat" doesn't represent an integer.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Operator Invocation (Part 3)

Given the following:

+(integer, integer) returns integer
+(string, string) returns string

The first definition performs addition, the second performs concatenation. Thus, the language would return 68 given 34 + 34 and "3434" given "34" + "34"; 34 + "34" is an error because there's no definition for +(integer, string).

Some languages do not distinguish operand types outside of operators and treat all values as strings, so the only signature (for "+") is effectively:
+(string, string) returns string

In such languages, when "+" is invoked it internally attempts to convert its operands to numeric values. If successful, the operator performs addition and returns a string containing only digits. If the conversion to numeric values is unsuccessful, the operator performs string concatenation on the operands and returns the result.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Operator Invocation (Part 4)

Other combinations are possible.

For example, given the following:
+(integer, integer) returns integer
+(string, integer) returns integer
+(string, string) returns string

The first two definitions attempt addition; the second parsing the string to determine if it represents a numeric value. Thus, the language would return 68 given 34 + 34, 68 given "34" + 34, "3434" given "34" + "34", an error given "splat" + 34 because parsing the string fails, and an error given 34 + "splat" because there is no operator definition for +(integer, string).

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Operator Invocation (Part 4)

Other combinations are possible.

For example, given the following:
+(integer, integer) returns integer
+(string, integer) returns integer
+(string, string) returns string

The first two definitions attempt addition; the second parsing the string to determine if it represents a numeric value. Thus, the language would return 68 given 34 + 34, 68 given "34" + 34, "3434" given "34" + "34", an error given "splat" + 34 because parsing the string fails, and an error given 34 + "splat" because there is no operator definition for +(integer, string).

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

Understand the Distinction: Dynamically vs Statically typed

Operator Invocation and Static Type Inference.

The return type of operators can be used to infer expression types prior to execution. For example, given operator signatures:

+(integer, integer) returns integer
+(string, integer) returns string
+(integer, string) returns string
+(string, string) returns string

And given an expression like: 34 + 38 + "splat" + 55

It can be represented as operator invocations: +(+(+(34, 38), "splat"), 55)

Now replace the operand values with their types: +(+(+(integer, integer), string), integer)

By replacing each operator invocation with its corresponding return value until we can't simplify further, i.e., +(+(+(integer, integer), string), integer) = +(+(integer, string), integer) = +(string, integer) = string, we determine that the expression is of type "string".

Note that dynamic dispatch of various sorts – single dispatch, multiple dispatch, etc. – may complicate this.

http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages

MEDITERRANEAN
COLLEGE

Thank You!