Type System Concepts

H. Conrad Cunningham

05 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good of April 2022 is a recent version of Firefox from Mozilla.

1 Type System Concepts

1.1 Chapter Introduction

The goal of this chapter is to examine the general concepts of type systems.

1.2 Types and Subtypes

The term *type* tends to be used in many different ways in programming languages. What is a type?

Conceptually, a *type* is a set of values (i.e., possible states or objects) and a set of operations defined on the values in that set.

Similarly, a type `S` is (a behavioral) *subtype* of type `T` if the set of values of type `S` is a "subset" of the values in set `T` an set of operations of type `S` is a "superset" of the operations of type `T`. That is, we can safely *substitute* elements of subtype `S` for elements of type `T` because `S`'s operations behave the "same" as `T`'s operations.

This is known as the *Liskov Substitution Principle* [12,20].

Consider a type representing all furniture and a type representing all chairs. In general, we consider the set of chairs to be a subset of the set of furniture. A chair should have all the general characteristics of furniture, but it may have additional characteristics specific to chairs.

If we can perform an operation on furniture in general, we should be able to perform the same operation on a chair under the same circumstances and get the same result. Of course, there may be additional operations we can perform on chairs that are not applicable to furniture in general.

Thus the type of all chairs is a subtype of the type of all furniture according to the Liskov Substitution Principle.

1.3 Constants, Variables, and Expressions

Now consider the types of the basic program elements.

A *constant* has whatever types it is defined to have in the context in which it is used. For example, the constant symbol `1` might represent an integer, a real number, a complex number, a single bit, etc., depending upon the context.

A *variable* has whatever types its value has in a particular context and at a particular time during execution. The type may be constrained by a declaration of the variable.

An *expression* has whatever types its evaluation yields based on the types of the variables, constants, and operations from which it is constructed.

1.3.1 Static and dynamic

In a *statically typed language*, the types of a variable or expression can be determined from the program source code and checked at "compile time" (i.e., during the syntactic and semantic processing in the front-end of a language processor). Such languages may require at least some of the types of variables or expressions to be *declared* explicitly, while others may be *inferred* implicitly from the context.

Java, Scala, and Haskell are examples of statically typed languages.

In a *dynamically typed language*, the specific types of a variable or expression cannot be determined at "compile time" but can be checked at runtime.

Lisp, Python, JavaScript, and Lua are examples of dynamically typed languages.

Of course, most languages use a mixture of static and dynamic typing. For example, Java objects defined within an inheritance hierarchy must be bound dynamically to the appropriate operations at runtime. Also Java objects declared of type `Object` (the root class of all user-defined classes) often require explicit runtime checks or coercions.

1.3.2 Nominal and structural

In a language with *nominal typing*, the type of value is based on the type *name* assigned when the value is created. Two values have the same type if they have the same type name. A type `S` is a subtype of type `T` only if `S` is explicitly declared to be a subtype of `T`.

For example, Java is primarily a nominally typed language. It assigns types to an object based on the name of the class from which the object is instantiated and the superclasses extended and interfaces implemented by that class.

However, Java does not guarantee that subtypes satisfy the Liskov Substitution Principle. For example, a subclass might not implement an operation in a manner that is compatible with the superclass. (The behavior of subclass objects are this different from the behavior of superclass objects.) Ensuring that Java subclasses preserve the Substitution Principle is considered good programming practice in most circumstances.

In a language with *structural typing*, the type of a value is based on the *structure* of the value. Two values have the same type if they have the "same" structure; that is, they have the same *public* data attributes and operations and these are themselves of compatible types.

In structurally typed languages, a type `S` is a subtype of type `T` only if `S` has all the public data values and operations of type `T` and the data values and operations are themselves of compatible types. Subtype `S` may have additional data values and operations not in `T`.

Haskell is primarily a structurally typed language.

1.3.3 Polymorphic operations

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (or operator symbols) are associated with implementations of the functions (or operations).

In general, two primary kinds of polymorphism exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at *compile time* based on the declared types of the entities. They exhibit *early binding*.

      Consider the language Java. It overloads a few operator symbols, such as using the + symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

      Haskell's *type class* mechanism, which we examine in a later chapter, implements overloading polymorphism in Haskell. There are similar mechanisms in other languages such as Scala and Rust.

   b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

      The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*. This the polymorphism associated with the class structure in Java.

      Haskell does not support subtyping. Its type classes do support *class extension*, which enables one class to inherit the properties of another. However, Haskell's classes are not types.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

   The object-oriented programming (e.g., Java) community often calls this type of polymorphism *generics* or *generic programming*.

   The functional programming (e.g., Haskell) community often calls this simply *polymorphism*.

TODO: Bring "row polymorphism" into the above discussion?

1.3.4 Polymorphic variables

A *polymorphic variable* is a variable that can "hold" values of different types during program execution.

For example, a variable in a dynamically typed language (e.g., Python) is polymorphic. It can potentially "hold" any value. The variable takes on the type of whatever value it "holds" at a particular point during execution.

Also, a variable in a nominally and statically typed, object-oriented language (e.g., Java) is polymorphic. It can "hold" a value its declared type or of any of the subtypes of that type. The variable is declared with a static type; its value has a dynamic type.

A variable that is a parameter of a (parametrically) polymorphic function is polymorphic. It may be bound to different types on different calls of the function.

1.4 What Next?

TODO

1.5 Exercises

TODO

1.6 Acknowledgements

In Spring 2018, I wrote the general Type System Concepts section as a part of a chapter that discusses the type system of Python 3 [4] to support my use of Python in graduate CSci 658 (Software Language Engineering) course.

In Summer 2018, I revised the section to become Section 5.2 in Chapter 5 of the evolving textbook *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [7]. I also moved the "Kinds of Polymorphism" discussion from the 2017 List Programming chapter to the new subsection "Polymorphic

Operations". This textbook draft supported my Haskell-based offering of the core course CSci 450 (Organization of Programming Languages).

In Fall 2018, I copied the general concepts section from ELIFP and recombined it with the Python-specific content [4] to support my Python-based offering of the elective course CSci 556 (Multiparadigm Programming) and for a posible future book [5]. This chapter sought to remain compatible with the concepts, terminology, and approach of the 2018 version of my textbook *Exploring Languages with Interpreters and Functional Programming* [7], in particular Chapters 2, 3, 5, 6, 7, 11, and 21.

In Spring 2019, I extracted the general concepts discussion [5] to create this chapter [6] for use in my Scala-based offering of CSci 555 (Functional Programming).

The type concepts discussion draws ideas from various sources:

- my general study of a variety of programming, programming language, and software engineering over three decades [1–3,8–19].
- the *Wikipedia* articles on the Liskov Substitution Principle [20], Polymorphism [21], Ad Hoc Polymorphism [23], Parametric Polymorphism [24], Subtyping [25], and Function Overloading [22]

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

1.7 Terms and Concepts

TODO

Object, object characteristics (state, operations, identity, encapsulation, independent lifecycle), immutable vs. mutable, type, subtype, Liskov Substitution Principle, types of constants, variables, and expressions, static vs. dynamic types, declared and inferred types, nominal vs. structural types, polymorphic operations (ad hoc, overloading, subtyping, parametric/generic), early vs. late binding, compile time vs. runtime, polymorphic variables.

1.8 References

[1] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[3] Timothy Budd. 2000. *Understanding object-oriented programming with Java* (Updated ed.). Addison-Wesley, Boston, Massachusetts, USA.

[4] H. Conrad Cunningham. 2018. Python 3 reflexive metaprogramming. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci658/notes/PythonMetaprogramming/Py3RefMeta.html

[5] H. Conrad Cunningham. 2018. Multiparadigm programming with Python 3. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci556/Py3MPP/Ch05/05_Python_Types.html

[6] H. Conrad Cunningham. 2019. *Type system concepts.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci555/notes/TypeConcepts/TypeSystemConcepts.html

[7] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[8] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++.* Wiley, Indianapolis, Indiana, USA.

[9] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

[10] Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21, 3 (1989), 359–411.

[11] Roberto Ierusalimschy. 2013. *Programming in Lua* (Third ed.). Lua.org, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil.

[12] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[13] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[14] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (First ed.). Artima, Inc., Walnut Creek, California, USA.

[15] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[16] David L. Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976), 1–9.

[17] Michael L. Scott. 2015. *Programming language pragmatics* (Third ed.). Morgan Kaufmann, Waltham, Massachusetts, USA.

[18] Robert W. Sebesta. 1993. *Concepts of programming languages* (Second ed.). Benjamin/Cummings, Boston, Massachusetts, USA.

[19] Simon Thompson. 1996. *Haskell: The craft of programming* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[20] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_principle

[21] Wikpedia: The Free Encyclopedia. 2022. Polymorphism (computer science). Retrieved from https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

[22] Wikpedia: The Free Encyclopedia. 2022. Function overloading. Retrieved from https://en.wikipedia.org/wiki/Function_overloading

[23] Wikpedia: The Free Encyclopedia. 2022. Ad hoc polymorphism. Retrieved from https://en.wikipedia.org/wiki/Ad_hoc_polymorphism

[24] Wikpedia: The Free Encyclopedia. 2022. Parametric polymophism. Retrieved from https://en.wikipedia.org/wiki/Parametric_polymorphism

[25] Wikpedia: The Free Encyclopedia. 2022. Subtyping. Retrieved from https://en.wikipedia.org/wiki/Subtyping