CSci 555: Functional Programming
Type System Concepts

H. Conrad Cunningham

3 February 2019

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of February 2019 is a recent version of Firefox from Mozilla.

Type System Concepts

Introduction

The goal of these notes are to examine the general concepts of type systems.

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` and 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* [Liskov 1987] [Wikipedia 2018a].

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.

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.

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.

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 thus 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 an example of a primarily structurally typed language.

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 polymorphic operations 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 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 is 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?

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.

Exercises

TODO

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 [Cunningham 2018a].

In Summer 2018, I revised it to become Section 5.2 in the new Chapter 5 of the textbook *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [Cunningham 2018b]. I also moved the "Kinds of Polymorphism" discussion from the 2017 List Programming chapter of that book to the new subsection "Polymorphic Operations". (This section draws on various Wikipedia articles [Wikipedia 2018b] and other sources.)

In Fall 2018, I copied the general concepts from ELIFP and recombined it with the Python-specific content from the first part of [Cunningham 2018a] to form a chapter for a posible future book based on Python 3.

This chapter sought to be compatible with the concepts, terminology, and approach of the 2018 version of my textbook *Exploring Languages with Interpreters and Functional Programming* [Cunningham 2018b], in particular of Chapters 2, 3, and 5.

In Spring 2019, I extracted the general concepts discussion from the Python 3 chapter for use in a Scala-based course

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.

References

[Cunningham 2018a]: H. Conrad Cunningham. Basic Features Supporting Metaprogramming, Chapter 2, *Python 3 Reflexive Metaprogramming*, 2018.

[Cunningham 2018b]: H. Conrad Cunningham. *Exploring Languages with Interpreters and Functional Programming*, https://www.cs.olemiss.edu/~hcc/csci450/ELIFP/ExploringLanguages.html, draft 2018.

[Liskov 1987]: Barbara Liskov. Keynote Address—Data Abstraction and Hierarchy, In the Addendum to the *Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)*, Leigh Power and Zvi Weiss, Editors, ACM, 1987. [local]

[Wikipedia 2018a]: Wikipedia, Liskov Substitution Principle, accessed 30 August 2018.

[Wikipedia 2018b]: *Wikipedia* articles on "Polymorphism", "Ad Hoc Polymorphism", "Parametric Polymorphism", "Subtyping", and "Function Overloading", accessed 30 August 2018.

Terms and Concepts

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.