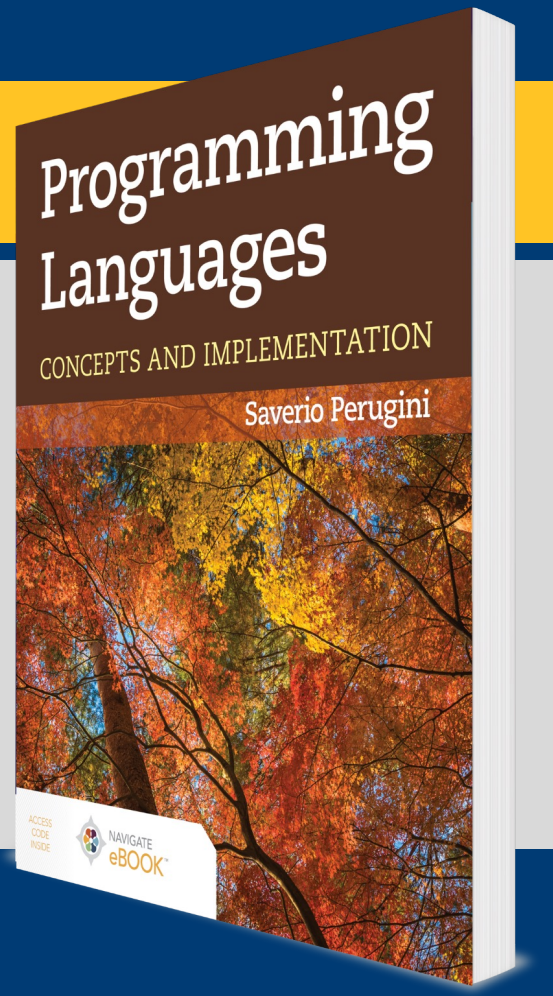
CHAPTER 7

Type Systems



Chapter 7: Type Systems

Clumsy type systems drive people to dynamically typed languages.

—Robert Griesemer

We study programming language concepts related to types—particularly, *type systems* and *type inference*—in this chapter.

Outline

- **7.1 Chapter Objectives**
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.1 Chapter Objectives

- Compare the two varieties of *type systems* for *type checking* in programming languages: *statically typed* and *dynamically typed*.
- Describe *type conversions* (e.g., *type coercion* and *type casting*), *parametric polymorphism*, and *type inference*.
- Differentiate between *parametric polymorphism* and *function overloading*.
- Differentiate between *function overloading* and *function overriding*.

Outline

- 7.1 Chapter Objectives
- **7.2 Introduction**
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.2 Introduction (1 of 2)

- The *type system* in a programming language broadly refers to the language's approach to type checking.
- In a *static type system*, types are checked and almost all type errors are detected before run-time.
- In a *dynamic type system*, types are checked and most type errors are detected at run-time.
- Languages with static type systems are said to be *statically typed* or to use *static typing*.
- Languages with dynamic type systems are said to be *dynamically typed* or to use *dynamic typing*.
- Reliability, predictability, safety, and ease of debugging are advantages of a statically typed language.
- Flexibility and efficiency are benefits of using a dynamically typed language.

7.2 Introduction (2 of 2)

- There are a variety of methods for achieving a degree of flexibility within the confines of the type safety afforded by some statically typed languages:
  - *parametric polymorphism*,
  - *ad hoc polymorphism,* and
  - *type inference*.
- Why ML and Haskell?

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- **7.3 Type Checking**
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.3 Type Checking

- A *type* is a set of values (e.g., `int` in C = {-2^{15} .. 2^{15} - 1}) and the permissible operations on those values (e.g., `+` and `*`).
- *Type checking* verifies that the values of types and (new) operations on them—and the values they return—abide by these constraints.
- Languages that permit programmers to deliberately violate the integrity constraints of types (e.g., by granting them access to low-level machine primitives and operations) have *unsound* or *unsafe type systems*.
  - e.g., Fortran, C, and C++ are *weakly typed languages*.
- In contrast, other languages do not permit programmers to circumvent type constraints
  - e.g., Java, ML, and Haskell all have a *sound* or *safe type system* and are, thus, sometimes referred to as *strongly typed* or *type safe languages* (Table 7.1).

Table 7.1 Features of Type Systems Used in Programming Languages

Concept	Definition	Example(s)
Static type system	Types are checked and almost all type errors are detected before run-time.	C/C++
Dynamic type system	Types are checked and most type errors are detected at run-time.	Python
Safe type system	Does not permit the integrity constraints of types to be deliberately violated.	C#, ML
Unsafe type system	Permits the integrity constraints of types to be deliberately violated.	C/C++
Explicit typing	Requires the type of each variable to be explicitly declared.	C/C++
Implicit typing	Does not require the type of each variable to be explicitly declared.	Python

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- **7.4 Type Conversion, Coercion, and Casting**
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.4 Type Conversion, Coercion, and Casting (1 of 2)

- 7.4.1 Type Coercion: Implicit Conversion
- 7.4.2 Type Casting: Explicit Conversion
- 7.4.3 Type Conversion Functions: Explicit Conversion

7.4 Type Conversion, Coercion, and Casting (2 of 2)

- There are a variety of methods for providing programmers with a degree of flexibility within the confines of the type safety afforded by some statically typed languages, thereby mitigating the rigidity enforced by a sound type system.

- These methods include *conversions* of various sorts, *parametric* and *ad hoc polymorphism*, and *type inference*.

7.4.1 Type Coercion: Implicit Conversion

- *Coercion* is an implicit conversion in which values can deviate from the type required by an operator or function without warning or error because the appropriate conversions are made automatically before or at run-time and are transparent to the programmer.
- See the C program `coercion.c`
- Notice also that coercion happens automatically without any intervention from the programmer.
- See the C program `storage.c`
- There are no guarantees with coercion.
- Java does not perform coercion (see `NoCoercion.java`), even between `float`s and `double`s (see `NoCoercion2.java`)

7.4.2 Type Casting: Explicit Conversion

- There are two forms of explicit type conversions:
  - type casts and
  - conversion functions.
- A `type cast` is an explicit conversion that entails *interpreting* the bit pattern used to represent a value of a particular type as another type.
- See the C program `cast.c`.

7.4.3 Type Conversion Functions: Explicit Conversion

- Some languages also support built-in or library functions to convert values from one data type to another.
- See the C program `conversion.c`.
- Since the statically typed language ML does not have coercion, it needs provisions for converting values between types.
- ML supports conversions of values between types through functions.
- Conversion functions are necessary in Haskell, even though types can be mixed in some Haskell expressions.

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- **7.5 Parametric Polymorphism**
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.5 Parametric Polymorphism: Monomorphic Function Types

- Both ML and Haskell assign a unique type to every value, expression, operator, and function.
- Recall that the type of an operator or function describes the types of its domain and range.
- Certain operators require values of a particular type.
  - e.g., the `div` (i.e., division) operator in ML requires two operands of type `int` and has type `fn : int * int -> int` ,
  - whereas the `/` (i.e., division) operator in ML requires two operands of type `real` and has type `fn : real * real -> real` .
- These operators are *monomorphic*, meaning they have only one type.

7.5 Parametric Polymorphism: Polymorphic Function Types (1 of 3)

- Other operators or functions are *polymorphic*, meaning they can accept arguments of different types.
  - For example, the type of the `(+)` (i.e., prefix addition) operator in Haskell is

```
(+) :: Num a => (a,a) -> a
```

indicating that if type `a` is in the type class `Num`, then the + operator has type

```
(a,a) -> a
```

- In other words, `(+)` is an operator that maps two values of the same type `a` to a value of the same type `a`.
- If the first operand to the `(+)` operator is of type `Int`, then `(+)` is an operator that maps two `Int`s to an `Int`.

7.5 Parametric Polymorphism: Polymorphic Function Types (2 of 3)

- With this type of polymorphism, referred to as *parametric polymorphism*, a function or data type can be defined generically so that it can handle arguments in an identical manner, no matter what their type.
- The types themselves in the type signature are parameterized.

7.5 Parametric Polymorphism: Polymorphic Function Types (3 of 3)

- A polymorphic function type in ML or Haskell specifies that the type of any function with that polymorphic type is one of multiple monomorphic types.
- Recall that a polymorphic function type is a type expression containing type variables.
- The polymorphic type `reverse :: [a] -> [a]` in Haskell is a shorthand for a collection of the following (non-exhaustive) list of types:

```
reverse :: [Int] -> [Int],   reverse :: [String] -> [String] ,
```

- and so on.
- Unlike in languages with unsafe type systems (e.g., C or C++), in ML, the programmer is not permitted—because a program doing so will not run—to deviate at all from the required types when invoking an operator or function.

7.5 Parametric Polymorphism

- In Haskell, as in ML, the programmer is not permitted to deviate at all from the required types when invoking an operator or function.
- However, unlike ML, Haskell has a hierarchy of type classes, where a class is a collection of types, which provides flexibility in function definition and application.
- Haskell's type class system comprises a hierarchy of interoperable type, where a value of a type (e.g., `Integral`) is also considered a value of one of the supertypes of that type in the hierarchy (e.g., `Num`).

Contrast a `square` Function in Haskell with That in ML

Haskell

```
Prelude> square(n) = n*n

Prelude> :type square
square :: Num a => a -> a
```

ML

```
- fun square(n) = n*n;
val square = fn : int -> int
```

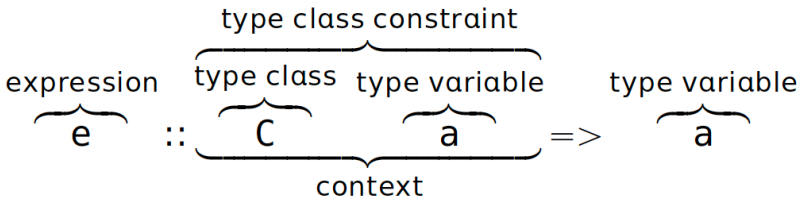
In Haskell, the type of `square` is called a *qualified type* and `Num` is a *type class.*

Qualified or Constrained Types in Haskell

- The `fromInteger` function is implicitly (i.e., automatically and transparently to the programmer) applied to every literal number without a decimal point:

```
Prelude> :type 1
1 :: Num p => p
```

- This response indicates that if type `p` is in the type class `Num`, then 1 has the type `p`.
- In other words, 1 is of some type in the `Num` class.
- Such a type is called a *qualified type* or *constrained type* (Table 7.2).



- A *type class* is a collection of types that are guaranteed to have definitions for a set of functions—like a Java `interface`.

Table 7.2 The General Form of a *Qualified Type* or *Constrained Type* and an Example

General: `e :: C a => a` means "If type `a` is in type class `C`, then `e` has type `a`."
Example: `3 :: Num a => a` means "If type `a` is in type class `Num`, then `3` has type `a`."

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- **7.6 Operator/Function Overloading**
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.6 Operator/Function Overloading (1 of 3)

- *Operator/function overloading* refers to using the same function name for multiple function definitions, where the type signature of each definition involves
  - a different return type,
  - different types of parameters,
  - and/or a different number of parameters.
- When an overloaded function is invoked, the applicable function definition to bind to the function call (obtained from a collection of definitions with the same name) is determined based on
  - the number and/or
  - the types of arguments used in the invocation.
- Function/operator overloading is also called *ad hoc polymorphism*.

7.6 Operator/Function Overloading (2 of 3)

- See `overloading.hs`
- In C, functions cannot be overloaded (see `nooverloading.c`)
- ML, Haskell, and C do not support function overloading.
- C++ and Java do support function overloading:
- See `overloading.cpp`, `overloading2.cpp`, and `Overloading.java`

7.6 Operator/Function Overloading (3 of 3)

- The Haskell type class system supports the definition of what seem to be overloaded functions like `square`.
- The flexibility fostered by a type or class hierarchy in the definition of functions is similar to ad hoc polymorphism (i.e., overloading), but is called *interface polymorphism*.
- ML and Haskell programs are thoroughly type-checked before run-time.
- Almost no ML or Haskell program that can run will ever have a type error.

Table 7.3 Parametric Polymorphism Vis-à-Vis Function Overloading

Type Concept	Function Definitions	Number of Parameters	Types of Parameters	Example Type Signature(s)
Parametric Polymorphism	single	same	parameterized	`[a] -> [a]`
Function Overloading	multiple	varies	instantiated	`int -> int`
(Ad Hoc Polymorphism)				`int * bool -> float`
				`int * float * char -> bool`

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- **7.7 Function Overriding**
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.7 Function Overriding (1 of 3)

Function overriding (also called *function hiding*) occurs when multiple function definitions share the same function name, but only one of those function definitions is visible at any point in the program due to the presence of scope holes.

7.7 Function Overriding (2 of 3)

```
1  (define overriding
2    (lambda ()
3      (let ((f (lambda ()
4                 (let ((g (lambda ()
5                            (let ((f (lambda () (+ 1 2))))
6                              ;; call to inner f
7                              (f)))))
8                   (g)))))
9        ;; call to outer f
10       (f))))
```

7.7 Function Overriding (3 of 3)

- Here, the call to function f on line 10 binds to the outermost definition of `f` (starting on line 3) because the innermost definition of `f` (line 5) is not visible on line 10—it is defined in a nested block.
- The call to function f on line 7 binds to the innermost definition of `f` (line 5) because on line 7 where `f` is called, the innermost definition of `f` (line 5) shadows the outermost definition of `f`.
- In other words, the outermost definition of `f` is not visible on line 7.

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- **7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing**
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing

- The concepts of *static/dynamic typing* and *explicit/implicit* typing are sometimes confused and used interchangeably.
- The modifiers “static” or “dynamic” on “typing” (or “checking”) indicate the time at which types and type errors are checked.
- However, the types of those variables can be declared explicitly (e.g., `int x = 1;` in Java) or implicitly (e.g., `x = 1` in Python).
- Languages that require the type of each variable to be explicitly declared use *explicit typing.*
- Languages that do not require the type of each variable to be explicitly declared use *implicit typing*, which is also referred to as *manifest typing* (Table 7.1).
- Statically typed languages can use either explicit (e.g., Java) or implicit (e.g., ML and Haskell) typing.
- Dynamically typed languages typically use implicit typing (e.g., Python, JavaScript, Ruby).

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- **7.9 Type Inference**
- 7.10 Variable-Length Argument Lists in Scheme
- 7.11 Thematic Takeaways

7.9 Type Inference (1 of 3)

- Explicit type declarations of values and variables help inform a static type system.
- See also `declaring.hs`
- In some languages with first-class functions, especially statically typed languages, functions have types.
- Instead of ascribing a type to each individual parameter and the return type of a function, we can declare the type of the entire function.
- Explicitly declaring types requires effort on the part of the programmer and can be perceived as requiring more effort than necessary to justify the benefits of a static type system.

7.9 Type Inference (2 of 3)

- Type inference is a concept of programming languages that represents a compromise and attempts to provide the best of both worlds.
- Type inference refers to the automatic deduction of the type of a value or variable without an explicit type declaration.
- ML and Haskell use type inference, so the programmer is not required to declare the type of any variable.
- ML and Haskell include a built-in type inference engine (i.e., *Hindley–Milner algorithm)* to deduce the type of a value based on context.

7.9 Type Inference (3 of 3)

- Strong typing provides safety, but requires a type to be associated with every name.
- The use of type inference in a statically typed language obviates the need to associate a type with each identifier:

Static, Safe Type System + Type Inference Obviates the Need to Declare Types

Static, Safe Type System + Type Inference ⇝ Reliability/Safety + Manifest Typing

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- **7.10 Variable-Length Argument Lists in Scheme**
- 7.11 Thematic Takeaways

7.10 Variable-Length Argument Lists in Scheme (1 of 2)

- Every function in Scheme is defined to accept only one list argument.
- Arguments to any Scheme function are always received collectively as one list, not as individual arguments.
- It is up to the programmer to decompose that argument list and group individual arguments in the formal parameter specification of the function definition using dot notation.
- Moreover, Scheme, like ML and Haskell, does pattern matching from this single list of arguments to the specification of the parameter list in the function definition.

`eval` / `apply` in Scheme

- `eval` and `apply` are the heart of any interpreter.
- `eval` is the Scheme primitive analog of `evaluate_expr` function in our coming Camille interpreter.
- `eval` accepts an expression and an environment as arguments.

```
> (define f (lambda (x) (cons x '())))
> (g 5)
> (5)
> (define f2 (list 'lambda '(x) (list cons 'x '())))
> f2
'(lambda (x) (cons x '()))
> (eval f)
#<procedure:f>
> ((eval f) 5)
'(5)
```

`apply` in Scheme

- `apply` is the Scheme primitive analog of the family of `apply_*` functions in our coming Camille interpreter (e.g., `apply_primitive`, `apply_closure`, and so on).
- `apply` accepts a function and its arguments as arguments:

```
>(apply + '(1 2 3))

> 6
```

7.10 Variable-Length Argument Lists in Scheme (2 of 2)

```
(define f (lambda (x) x))
(f 1)
(f '(1 2 3))
;;; x is just the list (1 2 3)
(define f (lambda x x))
(f 1 2 3)
(f 1)
;;; uses pattern matching like ML and Haskell
;;; g and h take a variable number of arguments
(define g (lambda (x . y) x))
(define h (lambda (x . xs) xs))
```

```
;;; continued from previous column
;;; only 1 argument passed
(g '(1 2 3))
(h '(1 2 3))
(write "a")
(newline)
;;; now 2 arguments passed
(g 1 '(2 3))
(h 1 '(2 3))
;;; now 3 arguments passed
(g 1 2 3)
(h 1 2 3)
```

Table 7.4 Scheme Vis-à-Vis ML and Haskell for Fixed- and Variable-Sized Argument Lists

	Natively		Through Simulation	
	fixed-size	variable-size	fixed-size	variable-size
Scheme	√ (only one argument)	×	√ (use .)	√ (use .)
ML	√ (one or more arguments)	×	N/A	×
Haskell	√ (one or more arguments)	×	N/A	×

Table 7.5 Scheme Vis-à-Vis ML and Haskell for Reception and Decomposition of Argument(s)

	Parameter(s) Reception	Single List Arg Decomposition	Example
Scheme	as a list	×	N/A
ML	as a tuple	√ (use `::`)	`x::xs`
Haskell	as a tuple	√ (use `:`)	`x:xs`

Outline

- 7.1 Chapter Objectives
- 7.2 Introduction
- 7.3 Type Checking
- 7.4 Type Conversion, Coercion, and Casting
- 7.5 Parametric Polymorphism
- 7.6 Operator/Function Overloading
- 7.7 Function Overriding
- 7.8 Static/Dynamic Typing Vis-à-Vis Explicit/Implicit Typing
- 7.9 Type Inference
- 7.10 Variable-Length Argument Lists in Scheme
- **7.11 Thematic Takeaways**

7.11 Thematic Takeaways

- Languages using *static type checking* detect nearly all type errors before run-time; languages using *dynamic type checking* delay the detection of most type errors until run-time.
- The use of automatic *type inference* allows a statically typed language to achieve reliability and safety without the burden of having to declare the type of every value or variable:

$$\text{Static, Safe Type System} + \text{Type Inference} \rightsquigarrow \text{Reliability/Safety} + \text{Manifest Typing}$$

- There are practical trade-offs between *statically* and *dynamically typed* languages—such as other issues in the design and use of programming languages.

Figure 7.1 Hierarchy of Concepts to Which the Study of Typing Leads