CENG 242 - Chapter 8: Type Systems

Burak Metehan Tunçel - May 2022

Older programming languages had very simple type systems that are not suitable for large-scale software development. Therefore, there are more powerful type systems, which were adopted by the modern programming languages.

In a programming language, there are choices for type systems. Design choices for types:

- *monomorphic* vs *polymorphic* type system.
- Is overloading allowed?
- Is coercion(auto type conversion) applied, how?
- Do type relations and subtypes exist?

Monomorphic and Polymorphic

Monomorphic types: Each value has a single specific type. Functions operate on a single type. `C` and most languages are monomorphic.

Polymorphism: A type system allowing different data types handled in a uniform interface:

- ***Ad-hoc polymorphism:*** Also called *overloading.* Functions that can be applied to different types and behave differently.
- ***Inclusion polymorphism:*** Polymorphism based on subtyping relation. Function applies to a type and all subtypes of the type (class and all subclasses).
- ***Parametric polymorphism:*** Functions that are general and can operate identically on different types.

1 Inclusion Polymorphism

Inclusion polymorphism is a type system in which a type may have subtypes, which inherit operations from that type. In particular, inclusion polymorphism is a key concept of object-oriented languages, in which a class may have subclasses, which inherit methods from that class.

1.1 Types and Subtypes

Recall that a type T is a set of values, equipped with some operations. A ***subtype*** of T is a subset of the values of T, equipped with the same operations as T. Every value of the subtype is also a value of type T, and therefore may be used in a context where a value of type T is expected.

It is better if the programming language allows us to declare the variable's *subtype*, and thus declare more accurately what values it might take. This makes the program easier to understand, and possibly more efficient.

If the programming language supports subtypes, we cannot uniquely state the subtype of a *value*. For example, the value 2 is not only in the type `int` but also in the subtypes `char` and `short` in `C`. However, the language should allow the subtype of each variable to be declared explicitly.

General properties of subtypes:

- A necessary condition for S to be a subtype of T is that every value of S is also a value of T, in other words $S \subseteq T$. A value known to be in subtype S can then safely be used wherever a value of type T is expected.
- The type T is equipped with operations that are applicable to all values of type T. Each of these operations will *also be applicable to values of the subtype S*. We say that S ***inherits*** these operations.

Suppose that a value known to be of type T_1 is computed in a context where a value of type T_2 is expected. In such a context we have insisted that T_1 must be equivalent to T_2. In the *presence of subtypes*, however, we can allow a looser compatibility between types T_1 and T_2.

T_1 is ***compatible*** with T_2 if and only if T_1 and T_2 have values in common. This implies that T_1 is a subtype of T_2, or T_2 is a subtype of T_1, or both T_1 and T_2 are subtypes of some other type.

If T_1 is indeed compatible with T_2, there are two possible cases of interest:

- T_1 is a subtype of T_2, so all values of type T_1 are values of type T_2. In this case, the value of type T_1 can be used safely in a context where a value of type T_2 is expected; no run-time check is necessary.
- T_1 is not a subtype of T_2, so some (but not all) values of type T_1 are values of type T_2. In this case, the value of type T_1 can be used only after a run-time check to determine whether it is also a value of type T_2. This is a kind of run-time type check, but much simpler and more efficient than the full run-time type checks needed by a dynamically typed language.

1.2 Classes and Subclasses

A class C is a *set of objects*, equipped with some operations (constructors and methods). These methods may be inherited by the *subclasses* of C. Any object of a subclass may be used in a context where an object of class C is expected.

Consider a class C and a subclass S. Each object of class C has one or more variable components, and is equipped with methods that access the variable components. Each object of class S inherits all the variable components of objects of class C, and may have additional variable components. Each object of class S potentially inherits all the methods of objects of class C, and may be equipped with additional methods that access the additional variable components.

Subclasses are *not exactly analogous to subtypes*. The objects of the subclass S may be used wherever objects of class C are expected, but the former objects have additional components, so the set of objects of subclass S is *not a subset* of the objects of C.

The methods of the superclass remain applicable to objects of the subclass, although they can be overridden (specialized) to the subclass if necessary.

2 Parametric Polymorphism

A ***monomorphic*** ("single-shaped") procedure can operate only on arguments of a fixed type. A ***polymorphic*** ("many-shaped") procedure can operate uniformly on arguments of a whole family of types.

Parametric polymorphism is a *type system* in which we can write polymorphic procedures. The functional language HASKELL are prominent examples of languages that exhibit parametric polymorphism.

2.1 Polymorphic Procedures

A ***type variable*** is an identifier that stands for any one of a family of types. In this section we shall write type variables as Greek letters $(\alpha, \beta, \gamma, \ldots)$, partly for historical reasons and partly because they are easily recognized. In HASKELL type variables are conventionally written as lowercase Roman

A ***polytype*** derives a family of similar types. Two examples of polytypes are $\sigma \times \tau$ and $\sigma \times \tau \to \tau$. A polytype always includes one or more type variables.

The family of types derived by a polytype is obtained by making all possible systematic substitutions of types for type variables. The family of types derived by $\sigma \times \tau \to \tau$ includes

- Integer × Boolean → Boolean
- String × String → String

It does not include

- Integer × Boolean → Integer
- Integer → Integer
- Integer × Integer × Integer → Integer

In other words, each type in the family derived by $\sigma \times \tau \to \tau$ is the type of a function that accepts a pair of values and returns a result of the same type as the second component of the pair.

In brief:

- Operations are same, but types are different.
- Types with type variables: *polytypes*.
- Most functional languages are polymorphic.
- Object oriented languages provide polymorphism through inheritance, run time binding and generics.

2.2 Parameterized Types

A ***parameterized type*** is a type that takes other type(s) as parameters. For instance, consider array types in C. We can think of τ[] as a parameterized type, which can be specialized to an ordinary type (such as `char[]`, or `float[]`, or `float[][]`) by substituting an actual type for the type variable τ.

All programming languages have built-in parameterized types. For instance, C and C++ have τ[] and τ^*. But only a few programming languages, notably HASKELL, allow programmers to define their own parameterized types.

Example 2.1

Consider the following HASKELL parameterized type definition:

```
type Pair τ = (τ, τ)
```

In this definition τ is a type parameter, denoting an unknown type.

Example 2.2

The following HASKELL (recursive) parameterized type definition defines homogeneous lists with elements of type τ:

```
data List τ = Nil | Cons (τ, List τ)
```

2.2.1 Polymorphism in C++ and Java

In such languages, inheritence provides subtyping polymorphism. C++ `virtual` methods, and all methods in Java implements late binding to improve polymorphism through inheritence.

Generic abstractions, C++ `template` and Java generics provide polymorphic classes and functions.

```
template <typename T>
void sort(T arr[], int n) {
  // ... your favorite sort algorithm here
}
```

```
class Test {
  //Java requires functions be in a class
  void <T> sort(T[] arr) {
    // ... your favorite sort algorithm here
  }
}
```

Code 1

C++ templates use compile time binding. Java generics binds at run time.

2.3 Type Inference

Type inference is a process by which the type of a declared entity is inferred, where it is not explicitly stated. Some functional programming languages such as HASKELL rely heavily on type inference, to the extent that we rarely need to state types explicitly.

Type inference sometimes yields a monotype, but only where the available clues are strong enough. However, the available clues are not always so strong: the function body might be written entirely in terms of polymorphic functions. Indeed, it is conceivable that the function body might provide no clues at all.

In these circumstances, type inference will yield a polytype and finds the *most general type satisfying the constraints.*

Excessive reliance on type inference, however, tends to make large programs difficult to understand.

- A reader might have to study the whole of a program in order to discover the types of its individual functions.
- Even the implementer of the program could have trouble understanding it: a slight programming error might cause the compiler to infer different types from the ones intended by the implementer, resulting in obscure error messages.

So explicitly declaring types, even if redundant, is good programming practice.

2.3.1 Inferring Type from Initializers

- C++11 `auto` type specifier *gets type from initializer or return expression.*
- C++11 `decltype(varexp)` *gets type same as the variables declared type.*

```
auto f(int a) {
  // double, function becomes double
  return a/3.0;
}
struct P { double x, y;} *pptr;

// double since pptr->x is double
decltype(pptr->x) xval;

// initializer is P typed
auto v = (P)({ 2.0, 4.0});
// f(3) returns double so t is double
auto t = f(3);
```

Code 2

- GCC has `typeof(expr)`, some other dialects have `__typeof__(expr)` macro having a similar mechanism in C.

3 Overloading

An identifier is said to be ***overloaded*** if it denotes two or more distinct procedures in the same scope. In other words, ***overloading*** is using same identifier for multiple places in same scope.

Such overloading is acceptable only if every procedure call is unambiguous, i.e., the compiler can uniquely identify the procedure to be called using only type information.

Polymorphic function: one function that can process multiple types.

C++ allows overloading of functions and operators.

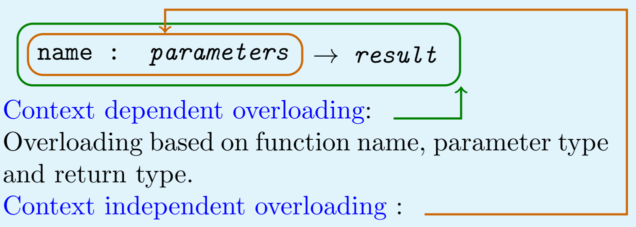
```
typedef struct Comp { double x, y; } Complex;
double mult(double a, double b) { return a * b; }
Complex mult(Complex s, Complex u) {
  Complex t;
  t.x = s.x*u.x - s.y*u.y;
  t.y = s.x*u.y + s.y*u.x;
  return t;
}
Complex a,b; double x,y;
a=mult(a,b) ; x=mult(y,2.1);
```

Code 3

We can characterize overloading in terms of the types of the overloaded functions. Suppose that an identifier F denotes both a function f_1 of type $S_1 \to T_1$ and a function f_2 of type $S_2 \to T_2$. (Recall that this covers functions with multiple arguments, since S_1 or S_2 could be a Cartesian product.) The overloading may be either *context-independent* or *context-dependent.*

In brief, binding is more complicated. Binding is made not only according to name but according to name and type.

Function type:



Context dependent overloading:
Overloading based on function name, parameter type and return type.
Context independent overloading :
Overloading based on function name and parameter type. No return type!

3.1 Context-dependent Overloading

Context-dependent overloading requires only that S_1 and S_2 are non-equivalent or that T_1 and T_2 are non-equivalent. If S_1 and S_2 are non-equivalent, the function to be called can be identified as below (context-independent overloading). If S_1 and S_2 are equivalent but T_1 and T_2 are non-equivalent, context must be taken into account to identify the function to be called. Consider the function call "$F(E)$", where E is of type S_1 (equivalent to S_2). If the function call occurs in a context where an expression of type T_1 is expected, then F must denote f_1; if the function call occurs in a context where an expression of type T_2 is expected, then F must denote f_2.

For example: Which type does the expression calling the function expects (context) ?

```
int f(double a) { ...① }
int f(int a) { ...② }
double f(int a) { ...③ }
double x,y;
int a,b;
```

Code 4

`a=f(x);` ① (x double)

`a=f(x);` ① (x double)

`a=f(a);` ② (a int, assign int)

`x=f(a);` ③ (a int, assign double)

`x=2.4+f(a);` ③ (a int, mult double)

`a=f(f(x));` ②(①) (x double, f(x):int, assign int)

`a=f(f(a));` ②(②) or ①(③) ???

Problem gets more complicated. (even forget about coercion)

Context dependent overloading is more expensive, complex and confusing. Most overloading languages are context independent.

3.2 Context-independent Overloading

Context-independent overloading requires that S_1 and S_2 are non-equivalent. Consider the function call "$F(E)$". If the actual parameter E is of type S_1, then F here denotes f_1 and the result is of type T_1. If E is of type S_2, then F here denotes f_2 and the result is of type T_2. With context-independent overloading, the function to be called is always uniquely identified by the type of the actual parameter.

Careful

Overloading is useful only for functions doing same operations. Two functions with different purposes should not be given same names. Confuses programmer and causes errors.

We must be careful not to confuse the distinct concepts of overloading (which is sometimes called *ad hoc polymorphism*) and *parametric polymorphism*. Overloading means that a small number of separately-defined procedures happen to have the same identifier; these procedures do not necessarily have related types, nor do they necessarily perform similar operations on their arguments. Polymorphism is a property of a single procedure that accepts arguments of a large family of related types; the parametric procedure is defined once and operates uniformly on its arguments, whatever their type.

4 Type Conversion

A ***type conversion*** is a mapping from the values of one type to corresponding values of a different type.

Programming languages vary, not only in which type conversions they define, but also in whether these type conversions are explicit or implicit.

A ***cast*** is an *explicit type conversion*. In `C`, `C++`, and `JAVA`, a cast has the form "$(T)E$". If the subexpression E is of type S (not equivalent to T), and if the programming language defines a type conversion from S to T, then the cast maps the value of E to the corresponding value of type T.

A ***coercion*** is an implicit type conversion, and is performed automatically wherever the syntactic context demands it. Consider an expression E in a context where a value of type T is expected. If E is of type S (not equivalent to T), and if the programming language allows a coercion from S to T in this context, then the coercion maps the value of E to the corresponding value of type T.

Some programming languages are very permissive in respect of coercions. However, the general trend in modern programming languages is to minimize or even eliminate coercions altogether, while retaining casts. At first sight this might appear to be a retrograde (back) step. However, coercions fit badly with parametric polymorphism and overloading, concepts that are certainly more useful than coercions. Casts fit well with any type system, and anyway are more general than coercions.