Type system of Go/C#

Ralf Vogler[1]

[1] Technische Universität München
Boltzmannstraße 3, D-85748 Garching, Germany
voglerr@in.tum.de

Abstract. This paper discusses the type systems of the programming languages Go (Google) and C# (Microsoft). Type systems in general, static vs. dynamic, nominal vs. structural and duck typing are explained. The two type systems are compared to each other and to other languages in order to highlight the differences and specialties. Particular focus is placed on the dynamic aspect of the two statically typed languages.

Keywords: type system, Go, C#

1 Introduction

"Well-typed programs cannot go wrong" (Robin Milner, 1978)

Type systems are arguably the most important part of programming language design. Values in a program are assigned different types, which can then be used for type checking in order to prevent unwanted behavior induced by wrong usage of functions, operators and so on. The addition of an integer and a string could for example result in a type error. Type safety describes to what extent those errors can be prevented. If there are no type errors, a program is called well-typed.
In statically typed languages type checking is done at compile-time, which saves time and helps program verification. In dynamically typed languages most type checking is done at run-time which can result in errors that are hard to find (e.g. implicit conversions) and makes testing even more important.

The two presented languages are relatively new and popular (TIOBE June 2011: C# 4th, Go 31st). Go is developed by Google (mainly Robert Pike) and was officially announced in November 2009. C# is developed by Microsoft within the .NET initiative and first appeared in 2001. The most recent version 4.0 was released in 2010. It is the most widely used language in .NET.
Both Go and C# are statically typed, imperative, compiled, garbage collected and offer type inference [1][2]. Go has additional features that simplify threaded programming and concurrency. C# is multi-paradigm and also allows functional, object-oriented and generic programming.
While C# has added many different language features over time (more than e.g. Java), Go rejects many concepts known from other languages (e.g. generic types,

exceptions, assertions, pointer arithmetic, overloading of methods and operators) in order to keep a low complexity [1].
Although being statically typed both languages have dynamic characteristics. C# got a dynamic type in the latest version 4.0 and Go's novel handling of interfaces makes it feel very dynamic and also allows some form of duck typing[1].

2 Static vs. dynamic, nominative vs. structural, duck typing

"I spent a few weeks... trying to sort out the terminology of "strongly typed," "statically typed," "safe," etc., and found it amazingly difficult.... The usage of these terms is so various as to render them almost useless." (Benjamin C. Pierce, Types and Programming Languages) [3]

The most prominent distinction between type systems is whether they use static or dynamic typing[2]. In the following the term static typing is used for type checking done at compile-time and dynamic typing used for type checking done at run-time. Depending on how strongly a language is typed this definition holds true.

Example static and dynamic type checking in C#

```
// static
int a = "1";      // type error at compile-time

// dynamic
dynamic b = "1"; // b.GetType() = System.String
int c = b;       // type error at run-time
```

Both systems have their advantages and disadvantages [4]. With static typing, type checks can be done at compile-time, with dynamic typing, type checks are done when needed, e.g. on assignment of a variable to another. This affects performance. In most cases dynamic typing is slower because of the extra type checks executed at run-time.
Another important aspect is error detection. Static typing avoids type errors up front or even proofs that there are none; with dynamic typing, differences in type might or might not cause an error long after the allegedly wrong assignment.
Static typing allows code assist in the development environment, which isn't always possible with dynamic typing or just after adding type hints or annotations.
Statically typed languages require specifying types and therefore require more planning and experience instead of programming in the beginning. They are good for large scale projects with big programs and many programmers. Dynamically typed languages are normally easier to learn, easier to use and allow fast prototyping. It facilitates code reuse: ported code is more likely to work or requires less

[1] See end of chapter 2
[2] There are different views on categorization and one could even say that "Dynamic languages are static languages" (Robert Harper, 2011) which just have one type for all values.

modification. Reflection[3] and serialization[4] are easier, as well as dynamic generation and evaluation of code.
These are some reasons why dynamic languages like JavaScript are getting increasingly popular, even outside the browser.
Besides type checking there are other aspects which can be dynamic. There are also programming languages that are considered partly dynamic even though they are statically typed. Go and C# for instance.

In a nominative (or name-based) type system, type compatibility is determined by the type names used in the declaration of variables. That means if two variables that have types T1 and T2, with T1 being structurally equal to T2, they are still considered incompatible.
In a structural (or property-based) type system T1 and T2 would be considered to be equal and therefore compatible. This allows more flexibility, for example ad hoc creation of types and interfaces, and more importantly creation of supertypes without the need of modifying existing code.
Nominal typing is considered to be safer, since it prevents unintended type equivalence. E.g a function format(partition string) and a function format(text string) could implement the same interface in Go.

Example Nominal typing in C#

```
    struct A { public int i; }
    struct B { public int i; }
…
       A a; B b;
       b = a;  // compile-time error
```

C# has a nominative type system. Go uses structural typing for interfaces. Most statically-typed languages are nominally typed. However many statically-typed functional languages like Haskell, OCaml and so on are structurally typed. There are also hybrids: C++ is nominally typed but its template system uses structural typing.

Duck typing is a style of dynamic typing. Structural typing checks the entire structure at compile-time, whereas duck typing only checks the part relevant for the current task at run-time: “If it walks like a duck and quacks like a duck, it must be a duck.” (attributed to James Whitcomb Riley). Whether a method exists or not only gets checked if it is called. Go allows duck typing through interfaces, C# allows it for objects of type *dynamic*. Likewise Objective-C allows sending any message to an object of type *id*. If the object cannot respond to a message, an exception is raised.

Example Duck typing in C#

```
       dynamic s = "Hello";
       s.ToUpper(); // ok
       s.foo();     // run-time exception
```

[3] Observation or modification of a program’s structure at run-time
[4] Storing/loading of data structures at run-time

3 Type system of Go

“It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.” [1]

Go’s type system is quite unusual yet also very simple. There are some built-in types including structures, functions and interfaces. Everything implementing the methods of an interface, automatically implements the interface. Types can be implicitly inferred from expressions and don’t need to be explicitly specified.
The special handling of interfaces and the implicit typing makes Go feel very lightweight and dynamic.

3.1 Built-in types, references and values

Built-in types “Named instances of the boolean, numeric, and string types are predeclared. Composite types—array, struct, pointer, function, interface, slice, map, and channel types—may be constructed using type literals.” [1].
Since Go doesn’t have generics but wants to offer type-safe lists and maps, it has two special types for this: slices[5] and maps. Channels[6], slices and maps are the only reference types and have to be allocated and initialized by `make()`. Everything else is a value type and can be allocated by `new()`. Different from C, even strings and arrays are values. That means that assigning an array to another or passing an array into a function copies all elements. Furthermore the size of an array is part of its type. That means that e.g. `[10]int` and `[20]int` are distinct types.

3.2 Object orientation, methods and interfaces

Go allows an object-oriented style of programming but there is no type hierarchy. Methods can be defined for any named type that is not a pointer or an interface. A method is declared like a function, just with a preceding parameter identifying the type it should work on. This is done outside the type declaration, which makes extensions easier. Dynamically dispatched methods are only possible through an interface, otherwise methods are resolved statically. An interface specifies what methods a type must provide to be used in some context. Unlike many other languages, Go uses structural typing for interfaces so that it’s enough to implement the methods of an interface in order to implement the interface itself.
This approach makes it possible to introduce new interfaces without manipulation of existing types. “[It] promotes separation of concerns and improves code re-use, and makes it easier to build on patterns that emerge as the code develops.” [1].
Ambiguities among similar interfaces might make it necessary to add an extra empty method to an interface in order to set it apart.

---

[5] Slices wrap arrays in order to offer more a convenient interface to data by reference

[6] Channels (see Hoare's Communicating Sequential Processes) are used for concurrency and combine communication and synchronization

Go doesn't provide typical type-inheritance but it allows embedding interfaces. For example a new interface can be defined by listing other interfaces (which must have a disjoint set of methods).

Example Interfaces, functions, implicit & explicit typing

```
package main
import "fmt"

type Walker interface { walk() }
type Talker interface { talk() }
type Duck struct {}
type Human struct {}
func (x Duck) walk() { fmt.Printf("Duck walks\n") }
// Duck now automatically implements the Walker
interface
func (x Human) walk() { fmt.Printf("Human walks\n")
}
func (x Human) talk() { fmt.Printf("Human talks\n")
} // Human implements the Walker and Talker
interfaces now

func doTheWalk(x Walker) { x.walk() }

func main() {
      d := new(Duck) // type inference, d is of
type Duck
      h := new(Human) // h of type Human
      doTheWalk(d) // call over the interface
      doTheWalk(h)
      h.talk()    // call on the struct
      var d2 Walker = new(Duck) // explicit typing
      var h2 Walker = new(Human)
      d2 = h2 // ok because same interface
      d2.walk() // h2.talk() couldn't be accessed
because it's not in the Walker interface
}
```

4 Type system of C#

Many features are added with each new version of C#, even concerning the type system. In the newest version C# 4.0 a dynamic type has been introduced.

4.1 The Common Type System (CTS) and .NET

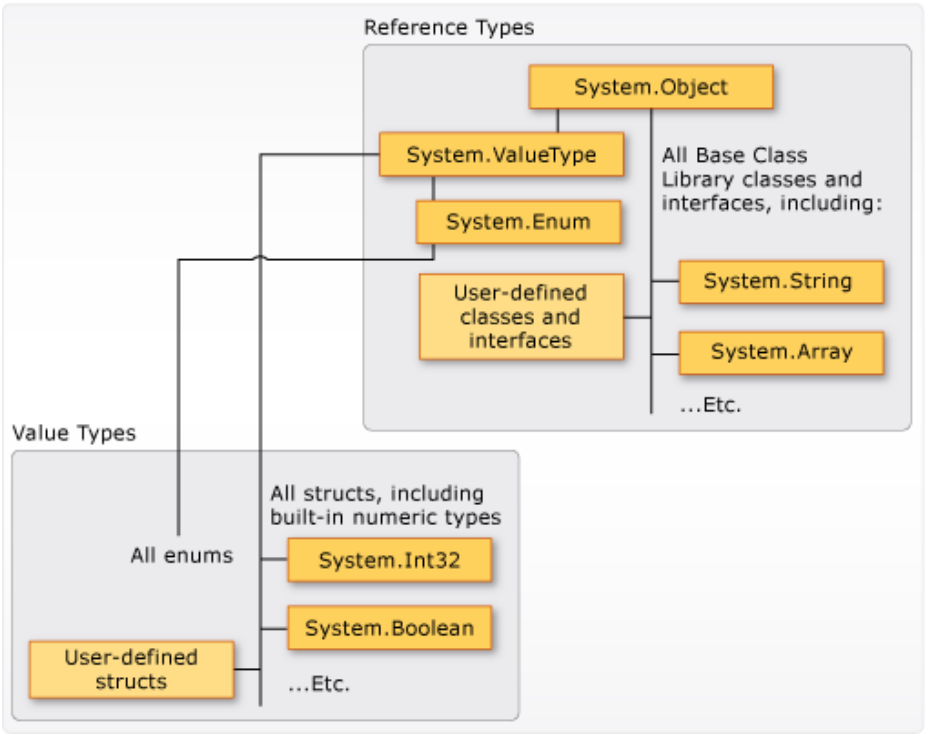

Fig. 1. The Common Type System of .NET [2]

The Common Language Infrastructure (CLI) which includes the Common Type System (CTS) is an open specification developed by Microsoft. Among others it is implemented by .NET and Mono.
Since C# is part of .NET it also uses the CTS which has two main categories of types: value types which contain their data and reference types which only contain a reference to the value's address in memory.

C# has a unified single-root type system [Fig1]. All types, including primitives, are subclasses of the System.Object class. That means that every type inherits the methods System.Object offers, e.g. ToString() and GetType(). For performance reasons all value types are allocated on the stack [2].

Converting value types to reference types is known as boxing. This can be done implicitly by the compiler or explicitly by casting. For example if a variable of a value type gets assigned to a variable of type object, it gets boxed to an object.
Unboxing converts reference types to value types. In contrast to boxing this requires an explicit cast [2].

The Common Language Runtime (CLR) provides garbage collection, just-in-time (JIT) compilation, sandboxing and other features.
The Dynamic Language Runtime (DLR) sits on top of the CLR and adds services that are needed for dynamic languages [Fig2] [5]. The DLR uses .NET's Object as the root of the type system which lets dynamic languages easily and naturally talk to each other [6].

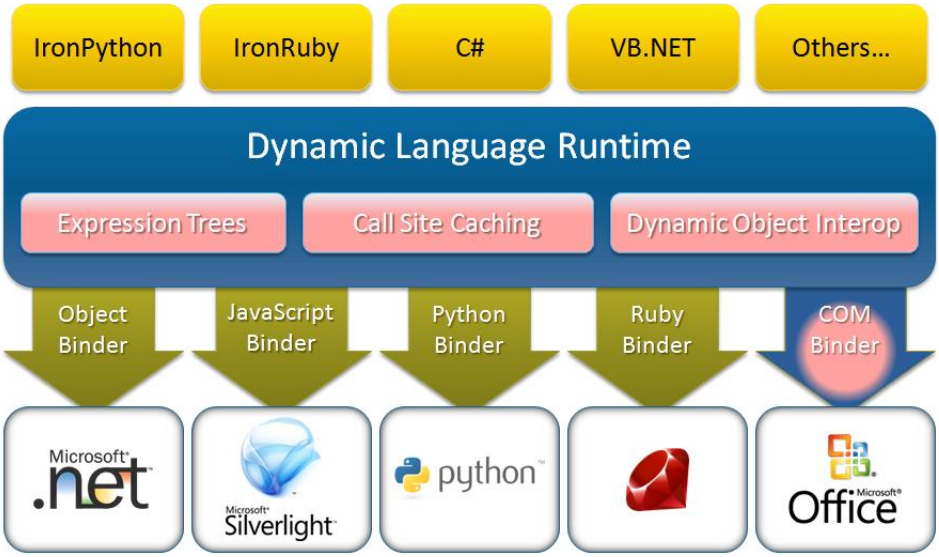

Fig. 2. The Dynamic Language Runtime [6]

4.2 Value types

Instances of value types are kept on the stack. Memory gets allocated directly after declaration and gets freed once the variable is out of scope.

Structures, also known as structs, are user-defined types that contain other types. All pre-defined primitive data types are structs. System.String not a struct, but a class and therefore not a primitive type.

Enumerations, also known as enums, are named values representing integers. Variables of type enum can be handled like integers in some ways and can also be cast to integers.

4.3 Reference types

Instances of reference types are typed managed references. After calling the constructor, the object is stored on the heap and a reference is assigned to the variable. When the variable gets out of scope the reference is broken and the object will be marked as garbage once there are no more references left. After some time it will be destroyed by the garbage collector and the memory will be freed.

Reference types include arrays, classes, interfaces, delegates, events, nullable types, pointers and the dynamic type.

4.4 Generics

Generics, also known as parameterized types, allow parametric polymorphism, i.e. code that is reusable across different types. To do this, type parameters are used instead of concrete types. Those parameters are then substituted by a concrete type once the generic is used. Parameterized types are instantiated at runtime and can be used for classes, interfaces, delegates and methods [7]. Unlike many other languages C# even allows specifying constraints for the type parameters [Table1].

Table 1. Constraints on Type Parameters [2]

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified.
where T: class	The type argument must be a reference type, including any class, interface, delegate, or array type.
where T: new()	The type argument must have a public parameterless constructor. When used in conjunction with other constraints, the new() constraint must be specified last.
where T: <base class name>	The type argument must be or derive from the specified base class.
where T: <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T: U	The type argument supplied for T must be or derive from the argument supplied for U. This is called a naked type constraint.

4.5 LINQ and its by-products

Language Integrated Query (LINQ) is a component for querying databases, XML-files and other sources. It had a lot of impact on new features added to the language and also on the type system. Since C# 3.0 queries can be done in code instead of using

strings. As an advantage all the queries can be checked for errors by the compiler. Internally it works by replacing certain SQL-like keywords with simple method calls. Lambda expressions[7] and implicitly typed variables via type inference have also been added in C# 3.0. By using the var-Keyword the type gets computed through local type inference (except for array initializers, see example).

Example LINQ query with implicitly typed variables and lambda expression

```
        string[] names = { "Delta", "Charlie", "Bravo", "Alfa" };
        var query = from s in names // type IEnumerable<string>
                    where s.Length == 5
                    orderby s
                    select s.ToUpper();
        foreach (var item in query.Select(x => x+1)) // additional
Select() with lambda expression, LINQ does the same
                Console.WriteLine(item); // BRAVO1, DELTA1
```

Anonymous types were introduced in order to ease the handling of query results. They allow the creation of nameless classes automatically generated by the compiler. The types are identified by their signature[8]. Two anonymous type declarations with the same signature also get the same class from the compiler.

Example Anonymous types

```
        var book1 = new { Title = "A", Pages = 123 };
        var book2 = new { Title = "B", Pages = 321 };
        var book3 = new { Title = "C", Pages = 321, Author = "D" };
        Console.WriteLine(book1); // { Title = A, Pages = 123 }
        book1 = book2; // ok because same type
        book2 = book3; // compile error because implicit conversion
not possible
```

Nullable types have been introduced in order to deal with database values that can be null. By adding a `?` after the type `T`, it becomes a nullable type `T?` which has two properties `HasValue` of type `bool` and `Value` of type `T` [7].

Example Nullable types

```
        int? n = 3; // same as using Nullable<int> n = 3
        Console.WriteLine(n); // 3, same as n.Value
        n = null;
        Console.WriteLine(n.HasValue); // False
```

Other things influenced by or at least supporting LINQ are anonymous methods through anonymous delegates or lambda expressions, extension methods, optional parameters, object/collection initializers and auto-implemented properties.

---

[7] Compiler generated methods, passed via delegates
[8] See structural typing in chapter 2

4.6 The dynamic type

The pseudo-type `dynamic` has been added in C# 4.0 and defers type checking from compile- to run-time. Identifiers of type dynamic are skipped during static type checking, type errors result in runtime-exceptions. This is called dynamic member lookup or duck typing (normally used when talking about methods). The ExpandoObject class allows to create objects with members that can be dynamically altered at run-time.

Example dynamic method dispatch, member lookup and object creation

```
        Action<dynamic> Print = x => Console.WriteLine(x);
        Func<dynamic,int> GetLength = x => x.Length;

        Print(123);     // calls WriteLine(int)
        Print("abc");   // calls WriteLine(string)

        Print(GetLength("Hello, world")); // a string has a Length
property
        Print(GetLength(new int[] { 1, 2, 3 })); // an array too
        Print(GetLength(42)); // but not an integer -> run-time
exception

        dynamic foo = new ExpandoObject();
        foo.att1 = "I'm dynamic";
        foo.meth1 = Print;
        foo.meth1(foo.att1);    // prints "I'm dynamic"
```

The main reason for the new type was interoperability, first of all with COM-technology (Component Object Model) which is used for interprocess communication and dynamic object creation. The dynamic type simplifies code using COM drastically. Reflection can be done much easier. Dynamic types and especially duck typing also make the adaption of code easier and faster.
All the work is done by the compiler. There is no dynamic type in the CTS or .NET. Dynamic objects are of the type System.Object. The compiler is adding the necessary code for reflection and dynamic member lookup. If using the ExpandoObject, all the members are stored in a dictionary in the object and can therefore also be modified during run-time. The dynamic type could be done in C# 3.0 by implementing the interface IDynamicMetaObjectProvider which tells the DLR that the object knows how to dispatch operations by itself [8].
Since it's optimized, the dynamic type is faster than using reflection and only little slower than normal static types[9].

[9] See Appendix: C# static vs. dynamic benchmark

5 Comparison

In order to summarize, table 2 compares the type systems of Java, C, Go, C#, PHP and JavaScript.

Table 2. Comparison of type systems

Language	Type checking	Type safety	Type strength	Expression of types
Java	static	safe	strong	explicit
C	static	unsafe	strong	explicit
Go	static	safe	strong	implicit, optionally explicit
C#	static with dynamic	safe but permits unsafe blocks	strong	explicit, implicit
PHP	dynamic	-	weak	implicit
JavaScript	dynamic	-	weak	implicit

Both Go and C# use static typing, support duck typing and type inference. Go has structural typing, the others use nominal typing. Except for C, all allow object oriented programming.

References

1. The Go Programming Language, http://golang.org/
2. The C# Language, http://msdn.microsoft.com/en-us/vcsharp/aa336809
3. B.C. Pierce: Types and programming languages. The MIT Press, Massachusetts (2002)
4. Pike, R.: Another Go at Language Design, Stanford (2010), http://www.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf
5. Albahari, J., Albahari, B.: C# 4.0 in a Nutshell: The Definitive Reference, pp. 735—749. O'Reilly, Sebastopol (2010)
6. Chiles, B., Turner, A.: Dynamic Language Runtime.
7. A. Heijlsberg, S. Wiltamuth, P. Golde: Die C#-Programmiersprache: Die vollständige Referenz, pp. 517—578. Addison-Wesley, München (2007)
8. C. Burrows Blog on MSDN, http://blogs.msdn.com/b/cburrows/archive/tags/dynamic/

Appendix

C# static vs. dynamic benchmark:

```
        private static void dynamicBenchmark(bool dynamic)
        {
            DateTime t1, t2;
            t1 = DateTime.Now;
            int a;
            double b;
            string c;
            dynamic d;
            for (int i = 0; i < 1E8; i++)
            {
                if (dynamic)
                {
                    d = i;
                    d = i * Math.PI;
                    d = i.ToString();
                }
                else
                {
                    a = i;
                    b = i * Math.PI;
                    c = i.ToString();
                }
            }
            t2 = DateTime.Now;
            TimeSpan t = t2 - t1;
            Console.WriteLine((dynamic ? "dynamic:" : "static:") +
t.TotalSeconds);
        }
```

- ➔ static: 25,5414609
- ➔ dynamic: 27,7925897