Information TypeSets in C#

A Proposal for C# 6

Jason Lind
EVP / Chief Strategy Officer
jason@lind-i.com / 414.788.2820

Abstract

Object oriented programming was a major evolution in computer programming, thinking in objects enables developers to code more efficiently and effectively. Modern OO languages, such as C# enable developers to use patterns that enable compile time type checking. This paper will be using C# 4.0 when discussing object oriented languages in general, while there certainly are OO languages whose features differ greatly from C# its implementation is exemplary of all major implementations OO languages.

Logic and data are organized into type constructs, such as classes and interfaces, which define what logic and information exist on a given type. The programmer can only access information defined on these types when writing their logic, if they attempt to perform an operation on a type that is not defined on that type their code will not compile. This innovation, in my opinion, was the key reason object oriented languages have become the de facto standard in software development.

The type system only goes so far, types are compile time defined and an instance of an object can only be of one type. This has not exactly been a major limitation, as a type can derive and compose other types so the developer can instantiate an object of type A, and if implements type B, they can treat the object as type B in the code, such as passing it to a method which takes a parameter of type B. The limitation is not in functionality, but in being able to produce better compile-time-safe code.

Information Safe Code

Modern type safe systems do a great job at enabling developers to enforce logic safe code, however they do a very poor job at enabling them to enforce information safety. Objects are much more than their type constructs and any non-trivial object has a myriad of metadata (synonymous with information in this paper) associated with it such as business rules and security. In order for a developer to place a type constraint that type must exist on that object, ergo while it is possible to express metadata through OO constructs such as interface the object must implement that interface, not just conform to it.

That an object must implement a given type leads to a significant impracticality of implementing logic that is information safe in modern OO languages. As type safe refers to type constraints, information safe refers to metadata constraints. Metadata constraints are known at compile time, a large part of programming is deciding how to treat an object by evaluating the object. The problem is to compile time enforce these constraints object definitions need to be designed and instantiated very carefully, and taking this step would generally require greater time and experience level than most projects have the benefit of.

It is the proposal of this paper to introduce information safe structures to C# through a concept we will call ‘typesets’. A typeset is a set of properties and constrains that extend a struct or map to a reference type, and are simply a pointer to that type.

The TypeSet

Email Example

```
namespace Lind.TypeSets.Example
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public #Email Email { get; set; }
    }
    public set Email : string
    {
        const string EmailRegex = @"\b[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b";
        public string DomainName
        {
            get { return value.Split('@')[1]; }
        }
        check
        {
            return Regex.IsMatch(value, EmailRegex);
        }
    }
    public set PublicEmail : #Email
    {
        check
        {
            return EmailData.IsPublic(value.DomainName);
        }
    }
    public set InternalEmail : #Email
    {
        check
        {
            return value.DomainName.ToUpper() == "LIND-I.COM";
        }
    }
}
```

It is important to differentiate between typesets and normal types, this can be accomplished by prepending the hash symbol to typeset names. Typesets can define a ‘check’ clause which assumes the value of the set as the input variable and returns a variable, indicating the validity of the set. The check clause is called by the compiler anytime a type is casted into a set, types that fail the check will either throw a runtime exception, if an explicit cast is used, or be marked as invalid if an optional cast is used. Passing an invalid set out of scope will result in a runtime exception.

The example above defines a Person entity with an Email typeset. The Email typeset is defined over string and uses a regular expression to perform the check. A PublicEmail typeset extends Email with the additional check of a database call to determine if the domain is public and an InternalEmail typeset extends Email with a check to see if the domain is corporate.

```
public class Example
{
    public static void EmailExample()
    {
        Person p = new Person();
        p.FirstName = "Jason";
        p.LastName = "Lind";
        p.Email = (#Email)"lind@yahoo.com"; //works
        Register(p.Email); //Can't register
        p.Email = (#Email)"jason@lind-i.com"; //works
        Register(p.Email); //Registers as a corporate user
        p.Email = (#Email)"j@j"; //throws exception
    }
    public static void Register(#Email email)
    {
        MailMessage message = new MailMessage();
        message.To.Add(email);
        if(email is #PublicEmail)
           message.Body = "We do not accept public emails.";
        else if(email is #InternalEmail)
           message.Body = "You have been registered as a corporate user.";
        else
           message.Body = "You have been registered as a standard user.";
        }
    }
}
```

The example above uses the Email typesets previously defined to register a Person.

Phone Example

This example uses a reference type which is mapped to a set.

```
namespace Lind.TypeSets.Example
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public #Email Email { get; set; }
        public #PhoneNumber PrimaryPhoneNumber { get; set; }
    }
    public class PhoneNumber
    {
        public string PhoneNumber { get; set; }
        public PhoneType Type { get; set; }
    }
    public enum PhoneType
    {
        Home,
        Work,
        Cell,
        Fax
    }
    public set PhoneNumber
    {
        public string PhoneNumber {get; set; }
        public PhoneType {get; private set; }
        public string AreaCode
        {
            get { return value.PhoneNumber.Split('-')[0]; }
        }
        check
        {
            return Regex.IsMatch(value.PhoneNumber, /*A Regex for XXX-XXX-XXXX*/);
        }
    }
    public map PhoneNumber to #PhoneNumber
    {
        PhoneNumber <-> PhoneNumber,
        Type -> PhoneType
    }
    public set HomePhoneNumber : #PhoneNumber
    {
        check { return value.PhoneType == PhoneType.Home; }
    }
    public set WorkPhoneNumber : #PhoneNumber
    {
        check { return value.PhoneType == PhoneType.Work; }
    }
    public set CellPhoneNumber : #PhoneNumber
    {
        check { return value.PhoneType == PhoneType.Cell; }
    }
    public set FaxNumber : #PhoneNumber
    {
        check {return value.PhoneType == PhoneType.Fax;
    }
    public set VoiceNumber : #PhoneNumber
    {
        check {return value.PhoneType != PhoneType.Fax;}}}
```

The above code adds the PhoneNumber set to person and defines a PhoneNumber class which maps to that set. SetMaps can be declared in any assembly that references both the reference type and the typeset and are declared outside the set definition. Any number of reference types can map to a set but only one map can exist for given reference/typeset pair.

The map definition introduces the <-> and -> keywords which indicate a two way and one way binding, properties on the typeset with a public get and set accessor must be defined as a two way binding and those with a private set accessor a one way binding.

The above example extends the PhoneNumber set by PhoneType for use in later business logic.

```
public class Example
{
    public static void AlertUser(string message)
    {
        Person p = GetCurrentPerson(); //Gets current person
        Alert(p.PrimaryPhoneNumber, message);
    }
    public static void Alert(#PhoneNumber number, string message)
    {
        int hour = DateTime.Now.Hour;
        if((number is #HomePhoneNumber && (hour < 9 || hour > 17))
            || number is #WorkPhoneNumber)
            SendVoiceMessage((#VoiceNumber)number, message);
        else if(number is #CellPhoneNumber)
            SendText((#CellPhoneNumber)number, message);
        else if(number is #FaxNumber)
            SendFax((#FaxNumber)number, message);
    }
    public static void SendFax(#FaxNumber fax, string message)
    {
        //TODO: Send Fax
    }
    public static void SendText(#CellPhoneNumber number, string message)
    {
        //TODO: Send Text
    }
    public static void SendVoiceMessage(#VoiceNumber number, string message)
    {
        //TODO: Send Voice Message
    }
}
```

The above example will alert a user's primary phone number based on phone number type and additional logic. In a real world scenario the Send methods would be implemented in a different layer.