C#

Information Safety Enhancements Specification

Draft – March 2011

1. Introduction to Information Saftey

C#, like most modern object-oriented languages, is generally strongly typed; and although dynamic typing was introduced with version 4.0 in order to offer developers more flexibility the strong typed nature of C# remains the core of the language paradigm. Strongly typed languages were introduced to eliminate a large set of code bugs, such as misspelled members or type declarations, by throwing a compile-time error to the developer as opposed to a run-time error to the end-user. Upon a successful compile the developer can be assured that bugs surrounding types and their members simply do not exist.

Information Safety is formed around a similar paradigm: business validation logic flaws being caught at run-time as opposed to compile-time. A common theme in modern application development is that users enter free form text which must conform to a certain schema, and then the application makes decisions based on the information that text represents. Generally this information is stored as String, which is simply an array of characters, which in turn is parsed and processed by the application.

The parsing of a string maybe to process it into a schema defined object representing that string which is then passed around instead of a naked string, but even then decisions often need to be made on the content of that schema. At the end of the day there is no built in, object-oriented, way to perform validation consistently across all tiers of an application without passing around heavy weight business objects, and even then there is no way to place compile-time constraints on the contents of an object.

To achieve information safe programming this document proposes several enhancements to C#:

- A “Set” type which defines an immutable, read-only representation of data that can be transformed into/from other types
- Set theory based operators for constraint evaluation and set manipulation
- An XML based language for defining types whose transformation and validation logic is entirely serializable

2. The “Set” Type

An information set, or set, is an immutable read-only object representing data and defining it’s transformation into/from other types, including constraints on if a type can be transformed into a given set. A set can be transformed into/from any numbers of types, and can be defined using the partial keyword in a similar fashion as other C# type definitions.

2.1 Set Semantics

Since sets have vastly different semantics than other C# types, we propose that they always be denoted with a hash in front of the set name so that it is clear to the developer they are working with a set. This is demonstrated in the type definitions bellow which will be used throughout these examples.

```
public enum PhoneNumberContactTypes
{
   Home,Work,Mobile,Fax
}
public class PhoneNumber
{
   public #PhoneNumber Phone{get;set;} //Denotes the PhoneNumber set
   public PhoneNumberTypes PhoneType{get;set}
}
```

2.2 Set Definition

Set definitions consist of (all optionally):

- Private variables and functions used by the transformation logic
- Public functions which have readonly access to the members
- Members, which can only be set during transformation (for/into)
- For type declaration, defining the validity and transformation into/from a type
  - Valid declaration defining the validity of a type transformation, withholding this implies always valid
  - Into declaration defining the transformation of a type into a set
  - From declaration definition the transformation of set into a type
- Union statements specifying a union between the set definition and another set definition
  - Valid declaration defining the validity of a union, withholding implies direct cast/union will fail

This particular example is a set definition for a PhoneNumber, with transformation on a string:

```
public set PhoneNumber
{
   member #AreaCode AreaCode;
   member string PhoneNumber;
   for string
   {
      //assume real regex
      const string PhoneNumberRegex = "<areacode><phonenumber>";
      valid{return Regex.IsMatch(value, PhoneNumberRegex);}
      into
      {
         PhoneNumber = value;
         AreaCode = (#AreaCode)Regex.Match(PhoneNumberRegex).Matches["areacode"];
      }
      from{return PhoneNumber;}
   }
}
```

2.2.1 Set Members

Set members are immutable, readonly values and therefore can only be of a value type or a set themselves. Further more the member name must be unique throughout the set and therefore sets being unified into a new type cannot have conflicting types for members of the same name. Also a unification will fail if the shared members of the underlying sets have conflicting values.

2.2.2 Type Transformation

Types are transformed into/from sets through a syntax similar to that of casting and type checking as demonstrated bellow:

```
if(methods.Contains(DeliveryMethod.Voice) && number is #VoicePhoneNumber)
   SendVoiceMail((#VoicePhoneNumber)number, message);
```

Comparing an object to is will only call the validation logic for the object's type and not the transformation logic, whereas a cast will throw an exception on a false validation and run the transformation logic to return the set.

The set definition can define validation, transformation from a type into a set and transformation from a set into a type. Withholding a validation block indicates that all values of a given type are valid, similarly withholding the into block indicates that the set cannot be transformed from that type and withholding the from block indicates that the set cannot be transformed into that type.

A set cannot declare a type transformation for a type already declared in one of its unified underlying sets.

- valid Func<T, bool>: The block defining the validation logic for the given type
- into Action<T>: The block that populates the members based on the type, assumes type is valid for transformation
- from Func<T>: The block that returns a T given the members of the set, set must be valid to even get here

Sets can define themselves from a set which is a superset of its definition as demonstrated with the AreaCode set bellow:

```
public set AreaCode
{
   member string Code;
   for string
   {
      const string AreaNumberRegex = "<areacode>";
      valid{return Regex.IsMatch(value, AreaNumberRegex);}
      into{Code = value;}
      from{return Code;}
   }
   for PhoneNumber
   {
      valid{return value.Phone != null;}
      into{this = value.Phone.AreaCode;} //AreaCode is of #AreaCode
   }
}
```

2.2.3 Set Unification

Set declerations can declare additional sets which they unify, in essence declaring themselves a subset of that set. It is important to note this is fundamentally different from inheritance as a set can logically be a subset of many different sets and does not nessecarly have parent.

Declared set unifications can optionally declare validation logic which enables direct cast evaluation of the underlying subsets as shown with the area code example.

```
public set USAreaCode
{
   union #AreaCode
   {
      valid{/*verify US area code*/}
   }
}
public set LocalAreaCode
{
   union #USAreaCode
   {
      valid{/*verify Local area code*/}
   }
}
```

However it is not always possible to allow direct casts from the subset as shown in SMSPhoneNumber, which while it is a superset of PhoneNumber, what makes it a superset is additional information not found on the PhoneNumber set. As demonstrated bellow, the into block allows union statements between the set and another. In fact the into statement must union all subsets for it to compile.

```
public set SMSPhoneNumber
{
   union #PhoneNumber;
   for PhoneNumber
   {
      valid{return value.PhoneType == PhoneNumberContactTypes.Mobile;}
      into{this union value.Phone;}
      from
      {
         return new PhoneNumber(){Phone = this,
            PhoneType = PhoneNumberContactTypes.Mobile};
      }
   }
}
```

3. Set Operators

3.1 Boolean Operators

Boolean operaters compare sets and return a Boolean value.

```
#A a;
a == #A; //a is #A
a != #B; //a is not #B
a @= #B; //a is a superset of B (incl ==)
a @@= #B; //a is a proper superset of B (no ==)
a =@ #B; //a is a subset of B (incl ==)
a =@@ #B; //a is a proper subset of B (incl ==)
```