Traits in CSharp

Stefan Reichhart[*]

Software Composition Group, University of Bern, Switzerland

Abstract. Traits are a well-known simple, but powerful compositional model for reuse. Although traits already implemented in dynamically typed languages, they're not yet practically realized in statically typed languages. Typing traits and adapting the model to these languages is more complex to achieve. We report on our experience and practical research implementing traits in CSharp 2.0, concerning generics. We show the difficulties and possible solutions of typing and parameterizing traits in generally, possible enhancements for statically typed languages as well as adapting traits to CSharp regarding features like overriding and hiding.

[*] thanks to Nathanael Schaerli for support on traits and Smalltalk, and Hans Beck for help on CSharp parsers

1 Introduction

The main focus of this work is on identifying interesting and important aspects of introducing traits to CSharp. We also identify required and optional features for statically typed languages as well as conflict situations. The implementation presented in this paper is a simple prototype (a preprocessor) based on the trait flattening property [9]. It is meant to be a study case for a clean implementation.

The main problem concerning traits and statically typed languages like CSharp is about typing traits and to keep the ability to share code easily. Many approaches already exist in theory. This practical work contributes to the research about traits by presenting a simple prototype, showing the possibilities and difficulties in integrating traits in statically typed languages.

Although the focus of this work is on CSharp most of the results are directly applicable to other typed object-oriented languages. The implemented "trait flattening framework", kept mostly language independent, would also work for most other c-like languages (with only slight modification).

Section 2 shortly introduces traits. The following sections cover the basics about CSharp, give a short overview about a first dirty-prototype done in CSharp itself and contain some more extensive descriptions, results and practical research by doing the final implementation using Smalltalk.

2 Traits in a Nutshell

The following paragraphs are extracted and shortened from [11]. More about traits might be found in [12, 14, 13, 11, 5, 1]. [10, 8] cover the implementation in Squeak/Smalltalk.

Traits are essentially sets of methods (i.e. mappings from method names to method bodies) that serve as the behavioral building block of classes and the primitive units of code reuse. Classes (and composite traits) are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state. With this approach, classes retain their primary role as generators of instances, while traits are purely units of behavior and reuse.

Traits contain methods, but no state, so state conflicts are avoided, but method conflicts may exist. A class is specified by composing a superclass with a set of traits and some glue methods. Glue methods are defined in the class and they connect the traits together; i.e., they implement required trait methods (possibly by accessing state), they adapt provided trait methods, and they resolve method conflicts.

Trait composition respects the following three rules:

– Methods defined in the class take precedence over trait methods. This allows the glue methods defined in a class to override methods with the same name provided by the used traits.
– Flattening property. A non-overridden method in a trait has the same semantics as if it were implemented directly in the class using the trait.
– Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

A conflict arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Conflicts are resolved by implementing a glue method at the level of the class that overrides the conflicting methods, or by excluding a method from all but one trait. In addition traits allow method aliasing; this makes it possible for the programmer to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden.

Traits bear comparison to mixins [4, 3, 6], but we do not discuss them in this report.

3 Exploring CSharp...

At the first glance, CSharp seems to be very similar to other statically typed languages with a slightly different syntax only. But when we have a closer look we discover that it is — simply said — much “bigger” than others. CSharp is actually a composition/mixture of multiple languages, providing many features of C, C++ and VisualBasic (.Net in general). This makes it part-wise more expressive than other statically typed languages but also more complicated.

An example for this is the “syntactical inflation” caused by the huge amount of keywords and various kinds of modifiers. The “explicitly” of CSharp, containing the overriding/hiding mechanism as well as the memory allocation, also contribute to its complexity.

3.1 Generics

Despite other attempts to introduce generics (Java 1.5) where generics are built on top of the existing language, not touching the VM, CSharp does have a native support in IL and CLR[1]. Therefore not only *classes* support generics, but also *structs* and other type-like structures and even other CLR-compliant languages of .Net, e.g. VisualBasic .Net.

Note: CSharp does provide generics at the type-level as well as at method-level. Both “scopes” are independent of each other. Therefore you might define a

[1] read more about IL/CLR and generics in .Net on http://msdn.microsoft.com/library

generic method `void foo<T>(T arg)` in a non-generic class or in a generic class with a different (or the same) generic type. [2]

4 CSharpT

As this project focused on CSharp it was only obvious to develop the traits extension in CSharp itself. The environment we were using was Mono[3] , the only freely available cross platform for the .Net frameworks.

This first developed prototype in CSharp, simply called CSharpT, is based on a very simple and heuristic low-level parser, extracting only the most basic information (but including generics) and representing the code as a simplified DOM tree. That allows to treat the code on a higher lever. The flattening logic respectively the preprocessor supports *Aliasing* `->` as the only feature.

The syntax for trait declarations are taken over by Smalltalk but adapted to fit CSharp. The trait declaration for using traits is defined as a preprocessor directive, but encapsulated in comments to keep the code permanently compilable. The following example shows this possible kind of declaration for traits.

```
class MyShape {
          // # trait TColor : color
          // # trait TCircle : color->colorCircle
}
```

This first prototype suffers many problems like relevant order of traits — which shouldn't be by the theory of traits. However it was mainly done to get a rough overview for a representative prototype as well as to develop a good and easy syntax for traits in statically typed languages – this one is quite different and more complex due types (Section 7).

5 Prerequisites: a good parser

The most important insight resulting from the CSharpT prototype is the need to have a good parser that allows code-snippets to be treated at a higher respectively object level. Without this prerequisite neither dynamic-linking nor flattening of traits would be reasonable or safe.

Therefore, the parser for the final implementation of traits in CSharp is based on a simplified version of CSharp's language grammar using the parser generator SmaCC [4]. The grammar rules, taken over by the provided Java parser example, have been adapted to CSharp and simplified to a minimum.

[2] CSharp Generics Indroduction at http://msdn.microsoft.com/library
[3] http://www.go-mono.com
[4] http://www.refactory.com/Software/SmaCC

This parser is able to identify all header information about structured objects like namespaces, types and type-members, but ignores the body of type-members. That means, all other code elements are identified either as simple statements or structured code and simply accepted without being checked. This behavior is similar to tolerant parsers as described in “Island Grammars” [7]. The following figure shows a sample block of parser rules based on that behavior, describing the method body’s elements.

```
chunkSequence   : <a sequence of tokens>
statement       : chunkSequence ";"
structuredChunk : chunkSequence "{" codeElements? "}"
codeElement     : statement | structuredChunk | ...
```

The advantages of this parser are that it is quite short and able to parse many other C-like languages as well. The drawback of this solution is that the parser might also accept invalid code as it does not parse method bodies and other elements on that level.

Section 8.3 shows that such a parser is not suitable to enable the full power of traits in CSharp or statically typed object oriented languages. Furthermore it is shown a complete grammar parser is required.

6 Introducing traits

6.1 Basics

The first step in introducing traits to CSharp or another statically typed object oriented language is to define a code-container for the trait elements. The simplest solution is to put each trait into a single file, declaring it similar to types, shown in the code example below.

```
traitCompilationUnit : "trait" "{" traitElements? "}"
traitElements        : traitElement | traitElements traitElement
```

where a traitElement is a simple method (everything else is not allowed, e.g. constructors, properties).

As traits and similar solutions are already well known it isn’t reasonable to introduce a completely new syntax for using traits as it would cause some extra effort to developers already familiar with traits, for example in Squeak/Smalltalk [13]. However slight adaptations have to be done to fit the syntax of the target language and because the syntax taken over by dynamically typed languages is not expressive enough due to missing types.

Traits could then be used defining a *use*-structure similar to the following one somewhere in the type-code, yet without any concern about conflicts and their resolution.

```
class MyCircle {
     uses {
         TColor; TCircle;
     }
 }
```

6.2 Aliasing and Exclusion

As proposed in the traits theory *aliasing* and *exclusion* have to be introduced to grant access to conflicting methods and avoid conflicts.

As the "@" symbol used in the Smalltalk prototype [10, 13] for aliasing is not common in statically typed languages, an other more appropriate solution should be developed to fit the language. The same holds for the "#" symbol which is already and often used for preprocessor directives in c-like languages. The example below shows a possible syntax, yet without introduced types for simplicity.

```
uses {
    TColor { foo -> fooAliased; };
    TCircle { ^fooExcluded; };
}
```

Each *aliasing* `->` and *exclusion* `^` is separated by a mandatory semicolon. Spaces are ignored. The aliasing is to be interpreted differently than in Smalltalk. So the arrow means "is-aliased-to".

6.3 Requirements

As traits not only provide, but also require a set of methods they need or depend on, it is necessary to enable this by introducing a declaration for requirements.

A syntax similar to the trait *use*-declaration might be chosen. That could look like in the following code example — again without types and omitted braces on methods.

```
trait TCircle {
    requires {
         resize;
     }
 }
```

7 Typing traits

7.1 Basics

As CSharp and most other statically typed object oriented languages offer *overloading*, it is required to type *Aliases*, *Exclusions* to distinguish overloaded methods. The following example shows how the trait declaration of Section 6.2 could be extended with types.

```
    uses {
        TColor { foo(int,double)->fooAliased; };
        TCircle { ^fooExcluded(ICollection); };
    }
```

Certainly, the type-declarations are case-sensitive and types have to be declared in the right order. If no arguments are used the "empty" braces might be omitted to reduce the effort of maintaining the declaration.

Return-types aren't required to be introduced as they're not relevant to distinguish overloaded methods. Therefore neither aliasing nor exclusion should define them.

A similar syntax strategy might be chosen for trait requirements. Although return-types aren't necessary for requirements either, it might make sense and be useful to declare them. This would allow a trait to more precisely specify its requirements towards a class or another trait using it. That helps preventing late compile-time errors on invalid/incompatible return types. The following code example shows a requirement on the method `radius()` using the simple return type `double`.

```
   trait TCircle {
        requires {
            void resize(double);
        }
   }
```

Another reasonable syntax for requirements would be the declaration of abstract methods, shown in the following code example.

```
   trait TCircle {
        public abstract void resize(double);
   }
```

7.2 Type Parameters

Although typed traits as introduced in Section 7.1 are already quite useful, their ability to share code and reduce code duplication is still limited. This disadvantage of reusability can be solved by introducing type parameters to traits. The syntax for this might be taken over by generics in CSharp or Java 1.5, shown in the following code example.

```
    trait TSequenceable <S> { ... }
```

The consequence of using parameterized traits is that code can be shared more flexible and easier. Two approaches of parameterized traits exist and are shortly mentioned.

Template-like traits look similar to templates in C++ but rather behave like generic traits and share their properties and implementation. They might be regarded as a lightweight approach to increase code sharing. Template-like traits aren't further discussed in this paper and not implemented either. However, to give an example, they might be enabled by simply substituting the type parameter in the *use*-declaration of the class (non-generic or generic), shown in the code below.

```
class MyNumberCollection {
    uses { TSequenceable<INumber>; }
}
```

(Full) Generic traits Instead of simply parameterizing the trait, it would be reasonable to directly apply generics to traits and use the advantages of generics. However, this is only possible in languages providing generics, as for example CSharp or Java 1.5. Generic traits are discussed in detail in Section 8.3.

```
class MyCollection <T> {
    uses { TSequenceable <T>; }
}
```

As CSharp provides generics on the class and method level (scope), both have been implemented in the prototype.

7.3 Return-types

When typing traits comes to return types, interesting questions and discussion topics like the following ones arise.

- What types should trait methods return?
- How much generic should or can return types be to guarantee code sharing?
- How can the type of a class be returned by a trait method without interfering the previous questions ?

Although these and others have not been deeply researched in this work — neither implemented — possible existing approaches are quickly shown and described.

Interfaces Trait methods returning interfaces (see code example below) might help sharing code with some additional work for the developer defining interfaces. Still the return type might not be the appropriate one in any cases as it would only allow a common subset of methods in the referenced classes to be called on the return type. Besides, interfaces alone don't solve the problem of returning the type of the class.

```
class MyCollection {
    uses { TList; }
}
trait TList : IList {
    IList reverse() { ... }
    void concat(IList a, IList b) { ... }
}
```

Implicit return types A simple approach in returning the type of the class is either by introducing a type keyword `ThisType` or by returning the “type” of the trait `TList` (not really a type!), shown in the following code example.

```
trait TList {
    ThisType reverse() { ... }            // keyword
    void concat(TList a, TList b) { ... } // trait 'type'
}
```

When the trait is used or referenced by the class the keyword `ThisType` or the type `TList` will by substituted by the type of the class `MyCollection`. However this approach lacks control of the type and might not be appropriate in any case either.

Explicit type parameter Using an explicit type parameter[11] for the trait enables traits to be shared easily among many classes due to flexible substitution of the parameter by any other one.

```
class MyCollection {
    uses { TList<ICollection>; }
}
trait TList<S> {
    S reverse() { ... }
    void concat(S a, S b) { ... }
}
```

However this definition might serve two situations/problems. So the developer has to know if the type parameter either refers to the return type as above or another type used by the trait/class.

This fact becomes obvious when the language supports generics using generic traits (discussed in Section 8.3) as shown in the following example.

```
class MyCollection<T> {
    uses { TList<T>; }
}
```

In this case the trait `TList<S>` from above couldn't be used in `MyCollection<T>`, as the type parameter in `TList<S>` addresses the return type and not the generic type of the class. The same could apply to template-like traits in (non-)generic classes. Therefore an explicit solution like the one from above doesn't go well with generics and causes inconsistency.

Although many approaches to this problem exist, a simple and consistent solution has still to be worked out and thoroughly tested.

8 Enhancements for Statically Typed Languages

8.1 Libraries

Many languages, including CSharp provide the mechanism to include libraries into the code, so for traits in statically typed languages. Certain definitions of a trait might depend on functionality provided by external components.

```
using System.Graphics;
trait TShape { ... }
```

The language enabling traits should automatically propagate all library statements of a trait to the class using the trait (duplicates removed). This way the programmer does not need to know about any dependencies of used traits.

The inclusion of libraries is not a required feature to enable traits in statically typed languages but seems to be reasonable and convenient. This feature has been included in the prototype implementation and proved very useful in code examples.

8.2 Trait Interfaces

Interfaces are a good programming tool in typed languages, for example to capture similarities among classes without class relationships or to reveal the programming interface without revealing its class.

Traits might also implement interfaces, like classes do.

```
trait TCircle : IShape { ... }
```

Meaning, trait `TCircle` implements `IShape`. When a class uses such a trait, the declared interface in the trait will be propagated to the class, declaring the class to implement that interface, shown in Figure 1. This supports the developer by saving time and code.

Trait Interfaces can also be regarded as a convenient addition for statically typed languages - they're not mandatory to fulfill the trait properties.

Besides trait interface propagation might cause some irritation or conceptional conflict when they are used together with exclusion. This fact is shown in the code example below.

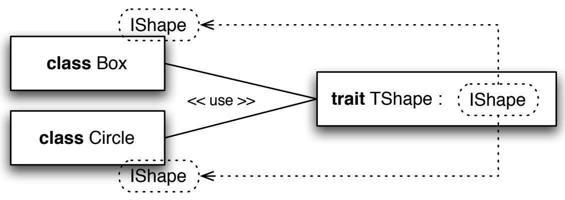

Fig. 1. interface propagation from the trait to the class

```
class MyCircle  {
    uses { TCircle { ^draw; }
}
trait TCircle : IShape { ... }
interface IShape {
    public void draw();
}
```

However the example shows a situation in which traits are “abused” for a selective code exclusion as only one class and trait is used. The trait exclusion mechanism is originally meant to prevent conflicts that may arise when two or more traits, used by a class, define the same method (explicit conflict resolution). Therefore the exclusion will normally not break the interface as at least one trait will provide the method excluded by the other traits. However this situation should be covered by implementing additional test routines.

The prototype implementation proposed in this paper supports trait interfaces for non-generic as well as for generic code. However, the combination of exclusions and interfaces is not handled and there aren’t any tests implemented to check the correct use or existence of the interfaces.

8.3 Generic Traits

When introducing generics to traits as already mentioned in Section 7.2 it is also important to take care about the generic types and their parameters. Depending on the strategy of using traits, flattening or dynamic/run-time, an appropriate variable binding mechanism and variable replacement/substitution has to be implemented to make the use of generic traits reasonable, shown in the following example.

```
class MyColor<T> {
    uses { TColor<T>; }
}

trait TColor<S> { ... }
```

The following relation has to be fulfilled in any case to assure that referenced traits do not change the "state" of the class:

$$genericTypeParameters(C) >= genericTypeParameters(\sum_{i=1}^{n}(T_i)) \quad (1)$$

with C a class using traits T_i; meaning, the used traits cannot have more or different generic type parameters (Section 9.4) than the class itself. In particular, a non generic class cannot use generic traits. The opposite does not violate the trait properties. The following example visualizes the relation above.

```
class MyCircle<T> {
    uses { TCircle<T,S>; }
}

trait TCircle<A,B> { ... }
```

If the relation above is ignored the class would become `MyCircle<T,S>` as we cannot drop the parameter `B`. This probably breaks any interface in the system, but moreover contradicts the properties of traits.

So, concerning variable bindings there are no free variables when using generic traits and all generic type parameters of used traits have to be boxed.

As the parser used in this prototype does not parse the method-level it was neither reasonable nor useful to test or implement a variable renaming respectively binding feature as this would not be very accurate, but rather heuristic and leading to many errors. So, the implementation of this paper only accepts "strict-matchings" of trait declaration (including generic type parameters) and therefore does not violate the trait properties. Certainly, this solution is extremely restrictive and is unsuitable for practical application of traits in a language providing generics as the developer has to care about each identifier or character representing a generic type parameter.

According to the examples above and the prototype implementation, no trait would be flattened to the class as the definition of `TColor<T>` is not found. You would need to explicitly change `S` to `T` in the trait declaration and in the code of the trait `TColor`. `TCircle<T,S>` is invalid and would cause an exception as it violates the relation above.

9 Enhancements for CSharp only

CSharp provides a bunch of different or "new" features - compared to other statically typed languages like Java. Therefore it makes sense to have a look at and treat these specially when introducing traits. The following subsections cover the most obvious and important ones.

9.1 Hierarchy modifiers / Polymorphism

Trait methods may want or need to override or hide methods of "higher" levels. This include methods of other traits or classes. A conceptional example for overriding traits is shown in Figure 2.

As CSharp has an explicit[5] overriding and hiding mechanism using the modifiers `virtual`, `override` and `new`, trait methods must be declared with the correct modifiers as otherwise the code would not be usable or compilable.

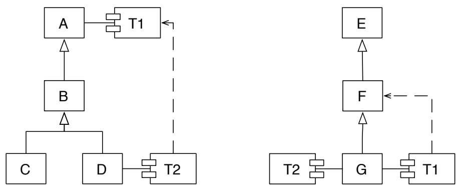

Fig. 2. Classes: A-G; Traits: T1, T2. Overriding is visualized by dashed lines. Trait T1 overrides behavior of Class F while it is not overriding anything when used by Class A. Many hierarchy modifier conflicts might exist in this constellation of classes and traits

Despite the advantage of this explicitly, it does also have a negative effect on traits. It prevents sharing trait methods easily among other classes or class hierarchies. This will lead either to code duplication or to fragile code as the developer has to know about all methods and their modifiers within the classes and traits.

There was no satisfying and simple solution found in the practical research using the prototype addressing this problem. However the following can be determined by experience:

Simply forbidding traits to hide or override would solve this problem quickly. However it has been proved in the refactoring [2, 11] of the Smalltalk collection hierarchy (using traits) that overriding trait methods are extremely powerful and help sharing code as well as avoiding and even eliminating code duplication. This is the same for traits in CSharp or other statically typed object oriented languages.

Enhancing the trait's use and requirement declaration with an explicit mechanism to override/hide the implemented modifier in the trait method isn't really useful either. This would blow up the declaration, making it hard to read and understand. Besides, the overriding and hiding problems might still exist, especially when introducing new subclasses and traits. This would lead to a very

[5] "CSharp Programmers Reference" at http://msdn.microsoft.com/library

complex maintenance of the declarations.

There is nothing concrete implemented in the prototype to address this problem in CSharp. However, as the flattening process in this prototype is done before the actual compilation (preprocessor) it is useful to catch most errors/exceptions as soon as possible. Therefore simple pre-compiler tests have been developed on the modifiers mentioned above. These tests check if modifiers are correctly applied on classes and traits. Furthermore they check if there are any conflicts between a trait used by multiple classes requiring different overriding or hiding modifiers from the trait methods (shown in Figure 2). This way the prototype is much more convenient to use.

9.2 Accessibility modifiers

The modifiers `public`, `private`, `protected` and `internal` do not actually cause any conflicts, but in addition to polymorphism in statically typed object oriented languages or hierarchy modifiers in CSharp it is useful to introduce tests to check if these modifiers are correctly applied when using overriding or hiding.

9.3 Other modifiers and keywords

Beside hierarchy and accessibility modifiers, CSharp comes with a heap of other modifiers and keywords. These have not been further considered in this practical research and implementation, but might be of interest and importance for a future implementation.

9.4 Generic bounds

In contrast to other statically typed languages having generics, CSharp provides a mechanism to bind the generic type (introduced for type safety), called a constraint [6].

It seems reasonable to support constraints for generic traits in CSharp, too. The applied syntax is identical to the one used for generic type definitions.

```
trait TNumberCollection<T> where T : INumber { ... }
```

As a trait is a composable unit of behavior it is not allowed to change the "state" of the object respectively the class. Therefore the following relation must hold to preserve the trait/flattening property:

$$constraints(C) >= constraints(\sum_{i=1}^{n}(T_i)) \tag{2}$$

[6] "CSharp Generics Indroduction" at http://msdn.microsoft.com/library

with C a class using traits T_i; meaning, the used traits cannot have different, more or more restrictive constraints on the generic type parameter than the class itself. The opposite does not violate the trait property. The following figure shows a code example with contradicting constraints.

```
class MyMath<T> where T : INaturalNumber {
    uses { TNumber<T>; }
}

trait TNumber<A> where T : IFloatingPoint { ... }
```

Generic constraints for traits are not really necessary for having traits in CSharp, but they might help developing code using traits. Many compile- and run-time errors can already be caught during the flattening or generally processing of traits. On the other hand constraints might conflict with the idea of traits preventing the ability to share behavior among classes because traits would become too specialized as a consequence of constraints.

10 Summary

Implementing traits in a typed object oriented language is reasonable and not difficult to achieve as shown with this simple flattening-based prototype in CSharp. Especially traits based on the flattening property using a preprocessor are easy and fast to realize as neither the language nor any compiler has to be changed. An iterative development/prototyping suits well for integrating traits into the target language as some of its properties and features appear only during implementation and testing. Moreover, not all trait features/extensions suit well for all typed object oriented languages.

Independent of doing a flattening or other approach to traits is chosen it is recommended to have mostly-complete and powerful high-level parser tools that allow to handle code in a rather object oriented manner. This way traits and most of the trait extensions for typed object oriented languages can be handled more easily and reliably. Moreover some extensions as shown in Section 8.3 may even require a complete parser.

Before implementing traits it is necessary to choose respectively adapt a good syntax (Section 6) to handle traits in the target language. This should be as “small” as possible, easy and quickly understandable. That means you should avoid introducing many new keywords or special syntax rules, keeping everything “self-explaining”. Moreover the syntax should fit the language regarding single tokens as well as structures. All this helps other developers using traits immediately without reading papers or going through many tutorials. Still the syntax should be flexible and extendable as later extensions/adaptions might follow.

The core of traits in typed object oriented languages using overloading is mostly formed by types (Section 7). They determine the key part of the trait syntax. Aliases and exclusions must enable argument types to distinguish overloaded methods. Requirement-declarations have to include them as well as they tighten the requirement and thus help preventing late run-time errors. Return types might also be required or just convenient to be included. They can also tighten the requirement and prevent compile-time errors.

If generic types are available in the target language, it is obviously to realize generic traits. However the generic parameters must be carefully treated as they require a well-designed variable binding mechanism (Section 8.3). Besides, other features like constraints in CSharp (Section 9.4) may exist and be integrated into traits.

As typed object oriented languages often use keywords and modifiers, it is also important to analyze them — in particular keywords for inheritance, hiding and accessibility. As shown in CSharp (explicitly of polymorphism) some may negatively affectSection 9.1 the use of traits, especially in more complex code or class hierarchies which are sensitive to conflicts and errors. Reasonably complex solutions to the modifier problem have to be worked out to use the full power of traits. Furthermore it is necessary to implement additional test routines on keywords and modifiers considering traits.

Beside syntax, types, generics and keywords there are many possible and more or less reasonable extensions for traits in typed object oriented languages. One could be the introduction of interfaces in trait definitions (Section 8.2) - if the language supports interfaces. Another one could be the automated inclusion of referenced libraries or modules (Section 8.1), e.t.c. Depending on the target language similar or other extensions might be necessary or only convenient for the developer.

Finally traits offer many possibilities of "features" and extensions for statically typed object oriented languages . However types do introduce an additional complexity in syntax and usage, and must be handled very carefully.

APPENDIX

A Implemented Prototype (summarized)

A.1 Basics

- A modular/extendable CSharp-parser framework (simplified CodeDom) understanding generics and other c-like languages with very slight adaptations necessary
- Detailed object extraction by the parser (attributes, parameters, etc)
- Convenient/easy use of preprocessor (automatic read/writeback to image/disc, detailed error report, etc)

A.2 Trait features

- Extendable and mostly language-independant trait logic, based on the flattening property
- Any CSharp-type provides support for traits
- Generic and non-generic traits
- Automatic library propagation
- Pre-compiler checks for common modifiers (simplified)
- Tests for generic constraints (strict matching only)
- Tests for generic types/parameters (strict matching only)
- Trait Interfaces

A.3 Missing features

- Return-type problem not handled
- Parsing of method-body (required for generic type binding/renaming)
- Template-like traits
- No existence check for trait interfaces (user responsibility)
- CSharp specific element-lookup respecting namespaces

A.4 Enhancements for the future

- Full-language-featured parser(-framework), accepting only valid code.
- Variable binding/Renaming for generic types for a more flexible usage of traits
- Implementing/evaluating various approaches to the return-type, keyword and modifier problem
- Combination of template-like and generic traits
- Constraints for requirement definitions (being even more restrictive)
- Dynamic/Runtime traits (no preprocessor) and/or clean implementation in the CSharp-language (or even in Rotor/CLI) itself

B Using the prototype

B.1 Download and installation

All sources are available in a pre-installed image on the SCG's Traits research pages or as a bundle ("CSharp") for VisualWorks 7.x at the SCG-Store.

```
db.iam.unibe.ch:5432_scgStore / CSharp
```

Before loading the bundle you have to make sure all of the following prerequisites are completely loaded.

- Smacc Tools
- SUnit, RBExtensions

The bundle contains several packages, each's name starting with "CSharp". Besides each package has a sibling package containing tests. All packages have an entry in "comments" that describes the content and how to use them. It is not recommended to load the packages separately!

- CSharpNamespace, CSharpFileTools : preprocessor and filetools for external appliance
- CSharpParser, CSharpCodeElements, CodeObjects : parser framework
- CSharpTraits, CSharpTraitsConflicts : trait logic
- CSharpTestCode : CSharp code-snippets/examples for testing

In some cases it might be necessary to recompile the parsers. To test if this is necessary simple run the tests within the parser package. If all tests fail, recompile one or all of the following parser/scanner definitions:

- SimplifiedStructure
- TypeSignature, ClassMemberSignature
- TraitDeclaration, TraitRequirement

Before you can use the prototype to flatten anything or run the trait tests you should execute the following line to setup the (image-internal) code-environment.

```
Environment uniqueInstance reinitialize.
```

B.2 Basic flattening

The package "CSharpTraitsTests" provides a class named "TestTraitsDemo" containing several simple examples (like the following one) showing the basic functionality of the prototype.

```
cu:=CSharpCompilationUnit from: (TestDemo readwritestream).
cu flattened inspect
```

This parses some code, creates, flattens and inspects a compilation-unit. "TestDemo" is a class placed in the package/code-container "CSharpTestCode".

B.3 Trait processing / flattening

The package “CSharpNamespace” provides a class “CSharpTraitsPreprocessor” that allows to transparently process an “environment” or “directory”. This process includes the tasks reading, parsing, updating the environment (linking compilation units), flattening, writing and error report.

```
CSharpTraitsPreprocessor >>>
     flattenDirectory: aDirectory
     flattenEnvironment: anEnvironment
```

A directory might contain one or more files and directories. The preprocessor recursively processes the given directory and ignores all non-Charp code files. If no errors occur during the process, you will find the flattened CSharp code in the directory given to the preprocessor as well as a copy of the originals (unmodified). You can process multiple directories at once.

```
Directory >>>
     usingDialog
     onFilename: aString
```

An environment is either internal (code stored in the image and accessible via class and selectors) or external (based on directories on the disk). When loading the image you already have a default global environment named “Environment” that contains all code of the package “CSharpTestCode”.

```
Environment >>>
     uniqueInstance
     reinitialize
```

More help is available in the bundle.

References

1. A. P. Black and N. Schärli. Traits: Tools and methodology. In *Proceedings ICSE 2004*, pages 676–686, May 2004.
2. A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. Technical Report IAM-02-007, Institut für Informatik, Universität Bern, Switzerland, Nov. 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.
3. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, Dept. of Computer Science, University of Utah, Mar. 1992.
4. G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
5. S. Ducasse, N. Schärli, O. Nierstrasz, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, 2005. under revision.
6. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
7. M. Kobel. Parsing by example. Diploma thesis, University of Bern, Apr. 2005.
8. A. Lienhard. Bootstrapping Traits. Master's thesis, University of Bern, Nov. 2004.
9. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *Journal of Object Technology*, 5(3):0–0, May 2006. To appear.
10. N. Schärli. Traits—composable units of behavior, Sept. 2003. http://www.iam.unibe.ch/∼scg/Research/Traits.
11. N. Schärli. *Traits — Composing Classes from Behavioral Building Blocks.* PhD thesis, University of Berne, Feb. 2005.
12. N. Schärli, S. Ducasse, and O. Nierstrasz. Classes = traits + states + glue (beyond mixins and multiple inheritance). In *Proceedings of the International Workshop on Inheritance*, 2002.
13. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.
14. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, Nov. 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.