Using FxCop with F#

[Introduction 2](#_Toc190026786)

[1 Design Goals 3](#_Toc190026787)

[2 Recommended Exemptions for All F# code 4](#_Toc190026788)

[2.1 AvoidUncalledPrivateCode 4](#_Toc190026789)

[2.2 AvoidUnusedPrivateFields 4](#_Toc190026790)

[2.3 DoNotCastUnnecessarily 5](#_Toc190026791)

[2.4 DoNotDeclareStaticMembersOnGenericTypes 5](#_Toc190026792)

[2.5 DoNotInitializeUnnecessarily 5](#_Toc190026793)

[3 Optional Exemptions for F#-to-F# DLLs 7](#_Toc190026794)

[3.1 DoNotNestGenericTypesInMemberSignatures (\*) 7](#_Toc190026795)

[3.2 GenericMethodsShouldProvideTypeParameter (\*) 7](#_Toc190026796)

[3.3 IdentifiersShouldBeCasedCorrectly (\*) 8](#_Toc190026797)

[3.4 IdentifiersShouldBeSpelledCorrectly (\*) 8](#_Toc190026798)

[3.5 IdentifiersShouldNotContainUnderscores (\*) 9](#_Toc190026799)

[3.6 NestedTypesShouldNotBeVisible 9](#_Toc190026800)

[3.7 OverloadOperatorEqualsOnOverloadingAddAndSubtract,OverloadOperatorEqualsOnOverridingValueTypeEqual, OperatorOverloadsHaveNamedAlternates, OverrideEqualsAndOperatorEqualsOnValueTypes, OverrideMethodsOnComparableTypes 10](#_Toc190026801)

[4 Exemptions currently required but being be actioned by F# Team 12](#_Toc190026802)

[4.1 DoNotDeclareVisibleInstanceFields (FSharp 1.0 #1385) 12](#_Toc190026803)

[4.2 AvoidUnsealedAttributes (FSharp 1.0 #1502) 12](#_Toc190026804)

[4.3 DoNotRaiseReservedExceptionTypes (FShap 1.0 #975) 13](#_Toc190026805)

[4.4 ImplementStandardExceptionConstructors (FSharp 1.0 1504) 13](#_Toc190026806)

[4.5 InterfaceMethodsShouldBeCallableByChildTypes (FSharp 1.0 #1502) 13](#_Toc190026807)

[4.6 RemoveUnusedLocals (FSharp 1.0 #1506) 14](#_Toc190026808)

[5 Exemptions Used by the F# Library 15](#_Toc190026809)

[5.1 AbstractTypesShouldNotHaveConstructors 15](#_Toc190026810)

[5.2 AvoidExcessiveParametersOnGenericTypes 15](#_Toc190026811)

[5.3 AvoidNamespacesWithFewTypes 15](#_Toc190026812)

[5.4 DoNotExposeGenericLists, PreferJaggedArraysOverMultidimensional 15](#_Toc190026813)

[5.5 OperationsShouldNotOverflow 16](#_Toc190026814)

# Introduction

F# and FxCop, etc. etc.

# Design Goals

F# and FxCop, in harmony.

# Recommended Exemptions for All F# code

This is based on FxCop 1.35

Here they are:

## AvoidUncalledPrivateCode

Resolution : "... appears to have no upstream public or protected callers."

Category : Microsoft.Performance (String)

CheckId : CA1811 (String)

RuleFile : Performance Rules (String)

Info : "There is uncalled code that is not externally visible

and is not a known runtime entry point. If this violation

fires in error, please send a problem report to the

Visual Studio Code Analysis team."

This stems from the fact that the F# compiler does inlining of methods internally to an assembly, which leave some internal property getters and other methods remaining on types. We guarantee to never optimize these away even internally in an assembly since they may be used by reflection. We won’t change or fix this behaviour. So this exemption must be applied to all F# code using any private OO members.

## AvoidUnusedPrivateFields

**Target** : **\_init** (IntrospectionTargetMember)

**Resolution** : **"It appears that field 'Array3.\_init' is never**

**used or is only ever assigned to. Use this field or**

**remove it."**

**Category** : Microsoft.Performance (String)

**CheckId** : CA1823 (String)

**RuleFile** : Performance Rules (String)

**Info** : "Private fields were detected that do not appear to

be accessed within the assembly. If this violation

fires in error, please send a problem report to the

Visual Studio Code Analysis team."

The F# compiler emits this as part of how we force the initialization class constructors of types to run. We emit a store into the field to force initialization (we should be doing this using RuntimeHelpers.RunClassConstructor but this is not supported on .NET CF, which makes it pretty useless as a “standard” way of forcing initialization. Hence we resort to writing into a field).

I think the real problem is that labelling the field as CompilerGenerated doesn’t help. Is this effectively a missing case in FxCop honouring CompilerGeneratedAttribute?

## DoNotCastUnnecessarily

Resolution : "..., a parameter, is cast to type ... multiple times in method ...

. Cache the result of the 'as' operator or direct cast in order to eliminate the redundant isinst instruction."

Category : Microsoft.Performance (String)

CheckId : CA1800 (String)

RuleFile : Performance Rules (String)

Info : "Avoid duplicate casts where possible, since there is

a cost associated with them."

These casts arise from the compilation of F# pattern matching. They could indeed normally be eliminated from F# code through further optimization, but it is not always trivial to do this. The F# team don’t think they can guarantee to make these eliminations 100% of the time, and may not even prioritorize optimization in this area, and indeed any optimizations may not be sufficiently repeatable to enable the user to reasonably action the warning with any reliability. Hence a blanket exemption is best.

## DoNotDeclareStaticMembersOnGenericTypes

Resolution : "Remove ... from ... or make it an instance member."

Category : Microsoft.Design (String)

CheckId : CA1000 (String)

RuleFile : Design Rules (String)

Info : "The syntax for calling static members on generic types

is complex as the type parameter has to be specified

for each call."

Static members in generic types are much less problematic for F# code, because F# can infer the types for the static members.

Indeed some static members in generic types are produced by compilation, e.g. compilation of

* extension members
* discriminated unions
* “instance” types that may use a null representation

Hence a blanket exemption for all F# code looks OK for now.

## DoNotInitializeUnnecessarily

Resolution : "... initializes field ... of type ... to 0. Remove this initialization as it will be done automatically by the runtime."

Category : Microsoft.Performance (String)

CheckId : CA1805 (String)

RuleFile : Performance Rules (String)

Info : "Do not make initializations that have already been

done by the runtime."

This fires on even simple code like:

let x = 0

It is extremely unlikely the F# team will adjust the F# code generator to avoid writing the value 0 into the storage used for “x” at this point. Hence the warning fires in many, many situations for F# code. To our knowledge the writing of null values has never been proven to be a signicant performance consideration for any application, etiher C# or F#. Hence a blanket exemption for all F# code looks OK for now.

# Optional Exemptions for F#-to-F# DLLs

Some DLLs are designed specifically to be used from F#. The F# library DLL FSharp.Core.dll is one such component. These use idioms that sometimes break or challenge invariants assumed by FxCop. While we are highly sympathetic to the goals of FxCop on each instance, the rules documented below seem candidates for blanket exemptions for F#-to-F# coding.

## DoNotNestGenericTypesInMemberSignatures (\*)

Resolution : "Consider a design where ...doesn't nest generic type ..."

Category : Microsoft.Design (String)

CheckId : CA1006 (String)

RuleFile : Design Rules (String)

Info : "Avoid API that require users to instantiate a generic

type with another generic type as type argument. The

syntax gets too complex."

This fires on simple F# code like:

let f (x,y) = match x,y with [(a,b)], [(c,d)] -> a+b+c+d

Generic tuples, lists and other types are used pervasively in F# coding. Hence this rule in nonsensical for F# or any libraries designed to be used specifically from F#. Hence a blanket exemption for all F# code looks OK for now.

(\*) The rule makes sense for libraries written in F# specifically designed to be used from C#

## GenericMethodsShouldProvideTypeParameter (\*)

Resolution : "Consider a design where Seq.tryfind\_index(FastFunc`2<T,

System.Boolean>, U):Option`1<V> doesn't require explicit

type parameter 'V' in any call to it."

Category : Microsoft.Design (String)

CheckId : CA1004 (String)

RuleFile : Design Rules (String)

Info : "Methods where the type parameter cannot be inferred

from the parameters and therefore has to be defined

in the method call are too difficult to understand.

Methods with a formal parameter typed as the generic

method type parameter support inference. Methods with

no formal parameter typed as the generic method type

parameter don't support inference."

This fires even on simple F# code like

let f (x: #seq<'a>) = for x in s do ()

The problem is that F# can infer types in many more situations than C#, hence many more generic signatures are acceptable in F# library design. This FxCop rule simply needs to be reconsidered when applied to F#-for-F# code.

(\*) The rule makes sense for libraries written in F# specifically designed to be used from C# and other .NET languages

## IdentifiersShouldBeCasedCorrectly (\*)

Resolution : "Correct the casing of member name ..."

Category : Microsoft.Naming (String)

CheckId : CA1709 (String)

RuleFile : Naming Rules (String)

Info : "Type, namespace, and member identifiers are pascal-cased.

Parameter identifiers are camel-cased. The pascal-casing

convention capitalizes the first letter of each word,

as in BackColor. The camel-casing convention formats

the first letter of the first word in lowercase and

capitalizes the first letter of all subsequent words,

as in backgroundColor."

F# uses substantially different naming conventions to C# and VB. For now a blanket exemption is realistic for most F# code, and particularly for the F# library. However this rule should be enabled if possible in production code.

NOTE: This rule will currently fire on a number of compiler-generated identifiers. This is being looked at by the F# team (FSharp 1.0 #1503)

(\*) The rule makes sense for libraries written in F# specifically designed to be used from C# and other .NET languages

## IdentifiersShouldBeSpelledCorrectly (\*)

Resolution : "In method ..., consider providing a more meaningful

name than the one-letter parameter name '...'."

Category : Microsoft.Naming (String)

CheckId : CA1704 (String)

RuleFile : Naming Rules (String)

Info : "The individual words that make up an identifier should

not be abbreviated and should be spelled correctly.

If this rule generates a false positive on a term that

should be recognized, add the word to the FxCop custom

dictionary."

This is, on the whole, a good rule. However there are many situations where shorter parameter names apply to F# coding, even for public F#-to-F# APIS. It will be very frequent that F# programmers disable this rule.

However this rule should be enabled if possible in production code.

NOTE: This rule will currently fire on a number of compiler-generated identifiers. This is being looked at by the F# team (FSharp 1.0 #1503)

(\*) The rule makes sense for libraries written in F# specifically designed to be used from C# and other .NET languages

## IdentifiersShouldNotContainUnderscores (\*)

Resolution : "Remove all underscores from member 'Choice5\_41'."

Category : Microsoft.Naming (String)

CheckId : CA1707 (String)

RuleFile : Naming Rules (String)

Info : "Do not use underscores when specifying identifiers.

Some generated Visual Studio identifiers for applications

contain underscore characters. Underscore characters

should generally be avoided in public identifiers."

The F# team are highly sympathetic to this rule. However, F# has a long heritage in OCaml programming, and underscores are used everywhere there. For example

let int\_to\_string (x:int) = x.ToString(...)

The current F# guidelines in the Expert F# book say that the use of underscores should be avoided or minimized. However many, many F# programs use underscores heavily, including in their public APIs for F#-to-F# coding.

Thus we encourage F# teams to consider a blanket exemption to this rule for F#-to-F# coding if necessary.

NOTE: This rule will currently fire on a number of compiler-generated identifiers. This is being looked at by the F# team (FSharp 1.0 #1503)

(\*) The rule makes sense for libraries written in F# specifically designed to be used from C# and other .NET languages

## NestedTypesShouldNotBeVisible

Resolution : "Do not nest type '...'.

Alternatively, change its accessibility so that it

is not externally visible."

Category : Microsoft.Design (String)

CheckId : CA1034 (String)

RuleFile : Design Rules (String)

Info : "Do not use public, protected or protected internal (

Protected Friend) nested types as a way of grouping

types. Use namespaces for this purpose. There are very

limited scenarios where nested types are the best design.

Also, nested type member accessibility is not clearly

understood by all audiences. Enumerators are exempt

from this rule."

F# encourages to the occasional use of modules. These can occasionally contain type definitions (indeed normally will do so when coding in OCaml-style). Modules containing type definitions are compiled as nested types. This is not liked by CLS or FxCop.

We encourage F# programmers to learn how to place their types in namespaces rather than nested modules. However it is equally reasonable for some F# code to use the style that is idiomatic in the language, for F#-to-F# DLLs.

(\*) The rule makes sense for libraries written in F# specifically designed to be used from C# and other .NET languages

## OverloadOperatorEqualsOnOverloadingAddAndSubtract,OverloadOperatorEqualsOnOverridingValueTypeEqual, OperatorOverloadsHaveNamedAlternates, OverrideEqualsAndOperatorEqualsOnValueTypes, OverrideMethodsOnComparableTypes

Resolution : "Provide a method named 'Decrement' as a friendly

alternate for operator LayoutOps.op\_Decrement(Layout,

Layout):Layout."

Category : Microsoft.Usage (String)

CheckId : CA2225 (String)

RuleFile : Usage Rules (String)

Info : "When redefining operators, implement named methods

to provide access to the operator functionality from

languages that do not support operator overloading.

For example, the functionality of the '+' operator

should also be accessible using an 'Add' method."

Resolution : "Because 'Complex' is a value type that overrides

Object.Equals, it should also overload the equality

and inequality operators (== and !=)."

Category : Microsoft.Usage (String)

CheckId : CA2231 (String)

RuleFile : Usage Rules (String)

Info : "Value types that redefine System.Object.Equals should

redefine the equality operator as well to ensure that

these members return the same results. This helps ensure

that types that rely on Equals (such as ArrayList and

Hashtable) behave in a manner that is expected and

consistent with the equality operator."

Resolution : "Consider adding an overload of the equality operator

for 'BigInt' that takes the same parameters as BigInt.op\_Subtraction(

BigInt, BigInt):BigInt."

Category : Microsoft.Design (String)

CheckId : CA1013 (String)

RuleFile : Design Rules (String)

Info : "When overloading the addition and subtraction operators,

make sure that the equality operator (==) is defined

in a consistent manner."

Resolution : "'Microsoft.FSharp.Math.BigInt' should override

the equality (==) and inequality (!=) operators."

Category : Microsoft.Performance (String)

CheckId : CA1815 (String)

RuleFile : Performance Rules (String)

Info : "The default System.ValueType implementation might not

perform as well as a custom implementation."

Resolution : "Unit should define operator '==' since it implements

IComparable."

Category : Microsoft.Design (String)

CheckId : CA1036 (String)

RuleFile : Design Rules (String)

Info : "Types that implement IComparable should redefine Equals

and comparison operators to keep the meanings of less

than, greater than, and equals consistent throughout

the type."

In all these cases the defined F# types are highly usable from F# code without needing further operators. The (==), (!=) operators are never called by F#, which implements a notion of “generic equality and comparison” via Object.Equals and System.IComparable.

For this reason, these rules only really make sense for libraries written in F# specifically designed to be used from C# and other .NET languages, because it is only use from these languages that may reveal the “inconsistencies” referred to in the messages. For F#-to-F# coding actioning the rules is optional and indeed discouraged.

# Exemptions currently required but being be actioned by F# Team

## DoNotDeclareVisibleInstanceFields (FSharp 1.0 #1385)

Resolution : "Make '... private or internal and provide a public

or protected property to access it."

Category : Microsoft.Design (String)

CheckId : CA1051 (String)

RuleFile : Design Rules (String)

Info : "Public or protected instance fields limit your ability

to change the implementation details for those data

items. Use properties instead. They do not compromise

usability or performance and they do provide flexibility

in that they conceal the implementation details of

the underlying data."

This is important for the F# team to fix. The contents of discriminated unions are being made public.

Until this is fixed the exemption will be required by F# code.

## AvoidUnsealedAttributes (FSharp 1.0 #1502)

Resolution : "Seal ...if possible."

Category : Microsoft.Performance (String)

CheckId : CA1813 (String)

RuleFile : Performance Rules (String)

Info : "Seal attribute types for improved performance. Sealing

attribute types speeds up performance during reflection

on custom attributes."

F# does not yet permit ‘sealed’ on class types, and attributes are defined using class types. This is an F# team work item.

Until this is fixed the exemption will be required by F# code declraing attribute types.

## DoNotRaiseReservedExceptionTypes (FShap 1.0 #975)

Resolution : "... creates an exception of type 'System.IndexOutOfRangeException', exception type that is reserved by the runtime and should never be raised by managed code. If this exception instance might be thrown, use a different exception type."

Category : Microsoft.Usage (String)

CheckId : CA2201 (String)

RuleFile : Usage Rules (String)

Info : "User code should not create and raise exceptions of

certain types that are reserved by the runtime or which

are of a too general exception type. Exception types

that are too general include Exception, SystemException,

and ApplicationException. Exception types that are

reserved by the runtime include ThreadAbortException,

OutOfMemoryException, ExecutionEngineException, and

IndexOutOfRangeException."

F# incorrectly uses IndexOutOfRangeException for the OCaml exception “Not\_found”. This will be corrected by the F# team. There are no security concerns with doing this: it is a somewhat arbitrary limitation placed by the CLS and FxCop. See the F# bug for more details.

Until this is fixed the exemption will be required by F# code.

## ImplementStandardExceptionConstructors (FSharp 1.0 1504)

Resolution : "Add the following constructor to ...Exception:

public ...Exception()

Category : Microsoft.Design (String)

CheckId : CA1032 (String)

RuleFile : Design Rules (String)

Info : "Multiple constructors are required to correctly implement

a custom exception. Missing constructors can make your

exception unusable in certain scenarios. For example,

the serialization constructor is required for handling

exceptions in XML Web services."

An F# exception declaration such as

exception Fail of string

will not include the extra constructors required by these rules. These can be automatically generated by the F# compiler. This is an F# team work item.

Until this is fixed the exemption will be required by F# code.

## InterfaceMethodsShouldBeCallableByChildTypes (FSharp 1.0 #1502)

Resolution : "Make '...' sealed (a breaking change if this

class has previously shipped), implement the method

non-explicitly, or implement a new method that exposes

the functionality of '...' and is visible to derived classes."

Category : Microsoft.Design (String)

CheckId : CA1033 (String)

RuleFile : Design Rules (String)

Info : "Explicit method implementations are defined with private

accessibility. Classes that derive from classes with

explicit method implementations and choose to re-declare

them on the class will not be able to call into the

base class implementation unless the base class has

provided an alternate method with appropriate accessibility.

When overriding a base class method that has been hidden

by explicit interface implementation, in order to call

into the base class implementation, a derived class

must cast the base pointer to the relevant interface.

When calling through this reference, however, the

derived class implementation will actually be invoked,

resulting in recursion and an eventual stack overflow."

On the whole the F# team disagrees with this rule: it is perfectly reasonable to not have derived classes be able to call interface implementations. Encouraging this “opening up” of functionality may indeed be a security hole.

That said, the main problem here is that F# does not yet permit ‘sealed’ on class types, and attributes are defined using class types. This is an F# team work item.

Until this is fixed the exemption will be required by F# code.

## RemoveUnusedLocals (FSharp 1.0 #1506)

Resolution : "... declares a local, '...', of type ..., which is never used or is only assigned to. Use this local or remove it."

Category : Microsoft.Performance (String)

CheckId : CA1804 (String)

RuleFile : Performance Rules (String)

Info : "Remove locals that are not used or are only assigned

to in method implementations."

It is reasonable for the F# compiler to never emit unused locals. This is an F# team work item.

Until this is fixed the exemption will be required by F# code.

# Exemptions Used by the F# Library

## AbstractTypesShouldNotHaveConstructors

Resolution : "Change the accessibility of all public constructors

in 'TypeFunc' to protected."

Category : Microsoft.Design (String)

CheckId : CA1012 (String)

RuleFile : Design Rules (String)

Info : "Public constructors for abstract types do not make

sense because you cannot create instances of abstract

types."

This stems from the lack of “protected” qualifiers in F# code.

## AvoidExcessiveParametersOnGenericTypes

Resolution : "Consider a design where ... has no more than 2 type parameters."

Category : Microsoft.Design (String)

CheckId : CA1005 (String)

RuleFile : Design Rules (String)

Info : "Avoid generic types with more than two type parameters

as users have difficulties understanding what type

parameters represent in types with long type parameter

lists."

Many types in the F# library have many type parameters by design (e.g. tuples). We should apply this exemption on a case-by-case basis.

## AvoidNamespacesWithFewTypes

Resolution : "Consider merging the types defined in...with another namespace."

Category : Microsoft.Design (String)

CheckId : CA1020 (String)

RuleFile : Design Rules (String)

Info : "A namespace should generally have more than five types."

We should apply this exemption on a namespace-by-namespace basis.

## DoNotExposeGenericLists, PreferJaggedArraysOverMultidimensional

Resolution : "Change '...' in ... to use Collection<T>,

ReadOnlyCollection<T> or KeyedCollection<K,V>"

Category : Microsoft.Design (String)

CheckId : CA1002 (String)

RuleFile : Design Rules (String)

Info : "Do not expose List<T> in object models. Use Collection<T>,

ReadOnlyCollection<T> or KeyedCollection<K,V> instead.

List<T> is meant to be used from implementation, not

in object model API. List<T> is optimized for performance

at the cost of long term versioning. For example, if

you return List<T> to the client code, you will not

ever be able to receive notifications when client code

modifies the collection."

Resolution : "'...' uses a multidimensional array of 'type

parameter.A[,,]'. Replace it with a jagged array if

possible. (Jagged arrays are not CLS Compliant.)"

Category : Microsoft.Performance (String)

CheckId : CA1814 (String)

RuleFile : Performance Rules (String)

Info : "Multidimensional arrays can have a negative impact

on performance. Use a jagged array if possible. Jagged

arrays are not CLS-compliant. If CLS compliance is

needed, exclude violations of this rule."

One F# library module ResizeArray is dedicated to functions for List<T>. Other functions are as well. We should apply this exemption freely on a case-by-case basis.

Likewise two F# library modules are dedicated to functions for multi-dimensional arrays. We should apply this exemption freely on a case-by-case basis.

## OperationsShouldNotOverflow

**Target** : **pred(System.SByte):System.SByte** (IntrospectionTargetMethodBase)

**Resolution** : **"Correct the potential overflow in the operation**

**'x-1' in 'SByteModule.pred(SByte):SByte'."**

**Category** : Microsoft.Usage (String)

**CheckId** : CA2233 (String)

**RuleFile** : Usage Rules (String)

**Info** : "Arithmetic operations should not be done without first

validating the operands to prevent overflow."

Checking the arguments would change the semantics of the operations being implemented. We should apply this exemption freely on a case-by-case basis.