AFL – Another Fun Language

Sam Wilmott
420 Tweedsmuir Ave, Ottawa, ON, Canada
sam@wilmott.ca

Abstract

AFL is a new programming language, intended primarily as an aid in learning and exploring basic programming language issues. In its present form it has coroutines, "first-class" generators, continuations, first-class functions, multi-argument operator definitions, asynchronous threading, pattern matching and an XML parser. The current emphasis is on control/processing forms rather than data structure or type issues.

1 About AFL

AFL is intended as an aid to thinking about and experimenting with programming languages, their features, their programming models, and their implementation. As such it includes features and forms that would not normally be used in general programming tasks, but which aid in understanding, and which also aid in the creation of tools for use in general programming.

An overview of AFL in its current state can be found in (`www.wilmott.ca/afl/afloverview.html`) AFL Overview. An short introductory slide set, mostly about the motivation for the AFL project, can be found in the PDF file (`www.wilmott.ca/afl/aflintro.pdf`) AFL Introduction.

Although I've been thinking about some of the ideas in AFL for some years, real work started in October 2004. What's available here is the current state of AFL, as of October 2005. Although it's been under development for a while, tested and has settled down, it's still best to consider it an "alpha" release – no big promises as to reliability or stability.

AFL is not (currently) an Open Software project. It's still a prototype.

2 Getting AFL

To get AFL you first need to download (`www.wilmott.ca/afl/aflinstall09.zip`) aflinstall09.zip. Then unzip the file, preferably into a new directory. The ZIP file will work for most systems.

To run the AFL compiler and the sample programs, you need some .NET tools:

- If you are on a Windows machine:
  - If you've got .NET Framework on your machine, which you have if you've done a Windows Update, you can run .NET programs.
  - If you have one or more of Microsoft's Visual .NET products installed then you've got a CIL assembler (ilasm.exe), and maybe the C# compiler (csc.exe).
  - You can download Microsoft's .NET Framework SDK which contains a lot of goodies including the CIL assembler and the C# compiler. Go to (`www.microsoft.com`) www.microsoft.com and enter ".net framework sdk" in the search box to find it. Make sure you get version 2.0 "Release Candidate", and not the Beta.
- If you're on a Linux, BSD or Mac OS X machine or are on a Windows machine without the .NET tools mentioned above, then there are a number of ways to get the required tools:
  - The one I've been using for Mac OS X and SUSE Linux is the product of the "Mono Project" at (`www.mono-project.com`) www.mono-project.com.
  - There's the (`www.dotgnu.org`) DotGNU Project, which I've not used but which looks promising.
  - You can also download the Shared Source Common Language Infrastructure from Microsoft's web site. Again, go to (`www.microsoft.com`) www.microsoft.com, enter "Shared Source Common Language Infrastructure" in the search box, and you'll be there in short order. It takes some work to get it going though.
  - Borland released a "C# Builder" product in 2003. There's a non-commercial "Personal download edition" for Windows on their web site dated 29 July 2003. I've no idea what the current status of the product is.

The tools I've been using, and had success with are:

- Visual Studio .NET Standard 2002, under Windows XP.
- Visual Studio .NET 2005 Beta 1 under Windows 2000 Pro.
- Visual Studio .NET 2005 Beta 2 under Windows XP.
- Visual Studio .NET 2005 (final release) under Windows XP Pro.
- Mono (I'm currently using release 1.1.9.something) under Mac OS X 10.4 (Panther) and 10.5 (Tiger), SUSE Linux 9.1 and Windows XP.

3 Running AFL

3.1 Windows with Visual .NET or the .NET Framework SDK

The `aflc.bat` file compiles and runs an AFL program, and is a good place to start. In more detail:

```
afl/il aflprogram.afl
```

compiles aflprogram.afl into a CIL assembler (".il") file.

```
ilasm/quiet aflprogram.il
```

compiles the CIL assembler program into an executable (".exe").

You can then run the executable like any other Windows program:

```
aflprogram options filenames
```

An alternative route is via C#:

```
afl/cs aflprogram.afl
csc/debug+ aflprogram.cs /addmodule:aflruntime.netmodule
aflprogram options filenames
```

To run from the command line using the Visual Studio or .NET Framework SDK tools you need to have the PATH and other environment variables set properly. There are two easy ways of getting this done:

- Under the Start>Programs menu there's a .NET-specific command prompt that's already set up to run .NET programs on the command line:
  - For a Visual Studio .NET product, under the "Start" menu click on "(All) Programs>Microsoft Visual Studio .NET>Visual Studio .NET Tools>Visual Studio .NET Command Prompt".
  - For the downloaded SDK, "(All) Programs>Microsoft .NET Framework SDK v2.0>.NET Framework SDK Command Prompt" does the same thing.
- If you want to do the setup yourself, there's a batch file that sets up all the environment variables correctly. (No need to do this if you use one of the above prompt programs.) Run it and you're in business. It helps to copy it to where you're doing your work:
  - For a Visual Studio .NET product, it's "Program Files\Microsoft Visual Studio .NET\Common7\Tools\vcvars32.bat".
  - For the downloaded SDK, it's "Program Files\Microsoft Visual Studio 8\Common7\Tools\vcvars32.bat".

These paths were found using Visual Studio .NET under Windows XP and the SDK under Windows 2000. Things might be not quite where I've said for you, but they should be close.

3.2 Linux, BSD, Mac OS X and Windows without Visual .NET

This description uses Mono.

The `aflmc` file compiles and runs an AFL program on shell-script friendly systems, and is a good place to start. For Windows systems running Mono, use `aflmc.bat` instead. In more detail:

```
mono afl.exe /il aflprogram.afl
```

compiles aflprogram.afl into a CIL assembler (".il") file.

```
ilasm aflprogram.il
```

compiles the CIL assembler program into an executable (".exe").

You can then run the executable as follows:

```
mono aflprogram.exe options filenames
```

(You must specify the ".exe" extension.)

An alternative route is via C#:

```
mono afl.exe -cs aflprogram.afl
msc -debug+ aflprogram.cs -addmodule:aflruntime.netmodule
mono aflprogram.exe options filenames
```

Under Linux the "C#" compiler is "cscc" instead of "mcs".

Installing Mono on non-Windows (Linux and Mac OS X) systems seems to put it in appropriate locations on the system, so you don't have to fiddle to get it running. On a Windows system, there's a Mono-specific command prompt that's put into the Start>Programs menu.

3.3 Cross-Platform Performance Notes

I was really surprised by the performance differences between the Visual Studio and Mono implementations. Here's figures from a couple of programs. They were all run on the same machine: a 3.2GHz Hyperthreaded Pentium IV running Windows XP, and all run with the maximum optimization available for the compiler (times are in hours:minutes!):

platform	**Program #1**	**Program #2**
Visual Studio 2002 std	2:47	6:40
V.S. 2002 std + afldefs2.afl	2:12	(untried)
V.S. 2005 Beta 2	8:17	19:20
Mono 1.1.8	1:13	2:45
Mono + afldefs2.afl	0:53	2:10

Presumably, even with no debugging and with optimization on, the Beta release of Visual Studio is jam packed with checks – the Beta's for testing the product, after all. The difference between V.S. 2002 and Mono is harder to explain – maybe because you don't get full optimization with a "Standard" Visual Studio. In any case, given these figures, and given the superior quality of error checking and error messages issued by the Visual Studio product, I tend to use Visual Studio for program development and testing, and Mono for running programs.

There's one more issue that I have trouble with. As far as I can determine, using the Mono platform under Windows, object finalizers aren't necessarily run for a program's global objects. Hence the implementation of "atEnd" doesn't always work. This causes difficulty with the implementation of AFL's textWriter, for example, which uses atEnd to ensure that an output is always flushed and closed by the end of a program's execution. There are two ways of dealing with this difficulty: putting in explicit close's for each textWriter value, or switching back to using Visual Studio.

3.4 Running the Samples

To start with, you can use the provided batch and script files to compile AFL programs.

Some of samples require no input and they output to the console: `factorial.afl`, `mixedops.afl`, `coroutines.afl` and `threadtest.afl`. For example:

```
aflmc factorial
mono factorial.exe
```

Other samples require input, or input and output: `text2html.afl` and `iotest.afl`. These samples can be compiled using the batch or script file. And then running the result of the compile with appropriate parameterization. For example:

```
aflmc text2html
mono text2html.exe aflreadme.txt aflreadme.html
```

and

```
aflmc iotest
mono iotest.exe aflreadme.txt aflreadme2.txt
```

3.5 Common Issues

`aflruntime.netmodule`, `afldefs.afl`, `afliltemplate.txt` and any include files used must available to the AFL and other compilers. The simplest way is to put them all together in the same directory as you are compiling. A better way is to put them in a safe place and use the `/p` and `/i` options on the AFL compiler to tell it where to look.

An Appendix of the (`www.wilmott.ca/afl/afloverview.html`) AFL Overview describes available options for the executable.

Running the AFL compiler with the /h option lists the (many) compiler options:

```
afl/h
mono afl.exe /h
```

The AFL compiler recognizes simple syntactic and name-definition errors, and isn't all that bad at reporting them. However, it doesn't catch usage and type errors, which are left for run time, and which will result in typically very obscure messages. It's hard to distinguish program errors from errors in the AFL compiler's translation or AFL run-time library. Please think kind thoughts and remember that this is an alpha release of a language without static type checking.

The /e=d compiler option provides a bit more, but not necessarily friendly, information when things go wrong. The /tr option, when specified for both the compiler and at run time, emits a copious if obscure trace of the running program. Both options place extra code in the run-time program and slow it down.

It seems that the Mono .NET runner doesn't interpret tail calls efficiently, so you could run out of stack space quickly if you're using the /il option to compile. To work around this problem, use the /cs option, or use the /e=b option, which tells the compiler to "trampoline" all continuation transfers. (To specify two options with the same name, combine the options, as in /e=bd, rather than /e=b /e=d.)

The C# compiler may produce warnings: the C# produced by the AFL compiler is correct, but sometimes a bit silly.

One thing more: the AFL compiler runs acceptably fast, but the compiled AFL programs run very slow.

4 What's In The AFL Download Package

(`www.wilmott.ca/afl/aflinstall09.zip`) aflinstall09.zip unzips to the files listed below. There are AFL system files and AFL programming samples.

4.1 The AFL System

`afl.exe`

The AFL compiler. This is a .NET assembly file.

`aflruntime.netmodule`

The AFL runtime library. This needs to be incorporated when compiling the C# or CIL output of the AFL compiler.

`afldefs.afl`

The default AFL include file. This file implements many of the common features of the AFL language.

`afldefs2.afl`

An alternative to "`afldefs.afl`" that implements boolean values using .NET Boolean rather than Church-style choosing functions. Using it (as in `-use:afldefs2.afl`) speeds up a compiled AFL program by about 20%.

`afliltemplate.txt`

A template file for the CIL code generator.

`io.afl  patterns.afl  sourceio.afl  threading.afl`

Include files. They are described in the (`www.wilmott.ca/afl/afloverview.html`) AFL Overview.

`xmlparser.afl  xmlparser2.afl`

Include files for accessing the .NET XML parser. They perfom the same functions except that "xmlparser2.afl" it will only work for .NET 2.0. See the comments in "xmlparser.afl" for more detail.

`aflc.bat  aflmc.bat  aflmc`

Sample batch and shell files for compiling AFL programs (via CIL): "aflc.bat" for Microsoft's .NET under Windows, "aflmc.bat" for the Mono platform under Windows, and "aflmc" for the Mono platform under Linux and Mac OS X.

`aflreadme.pdf  afloverview.pdf  aflchanges.pdf`

Descriptions of the AFL project and system in PDF form.

`aflreadme.html  afloverview.html  aflchanges.html`

The documentation in HTML.

`aflintro.pdf`

An introductory why for the AFL project.

4.2 Samples

(`www.wilmott.ca/afl/aflinstall09.zip`) aflinstall09.zip also contains samples of using AFL, and appropriate input files for running the samples:

`coroutines.afl`

A simple coroutine example. No input file used.

`factorial.afl`

Print out factorials, 0! to 20! inclusive. No input file used.

`gentest.afl`

Examples of using the generator operators. No input file used.

`mixedops.afl`

Use different combinations of mixed operator and function definitions. No input file used.

`text2html.afl  text2html.txt`

Convert a simple (non-XML) marked up text to HTML. `test2html.txt` is a small sample input for `text2html.afl`.

`threadtest.afl`

A simple asynchronous threading example. No input file used.

`iotest.afl`

Simple line-by-line file copying. Running this program requires an input text file for it to copy, but any text file will do.

`xmltest.afl  xmltest.xml`

Parse an XML document. `xmltest.xml` is a sample input for `xmltest.afl`. This sample uses the .NET 1.x library's XML parser.

`xmltest2.afl`

The same as xmltest.afl, except that it uses the .NET 2.0-compliant XML parser interface.

`trytest.afl`

Exercise the **try** ... **except** and **try** ... **finally** facilities. No input file is used.

`filters.afl`

Playing with filters. This sample is a small example of filters using pattern matching to provide input to other filters, plus a mixture of pattern-matching and non-pattern matching text processing.

`mididump.afl  midiutil.afl`

Utilities for creating and displaying MIDI music files. "mididump.afl" displays a MIDI file. "midiutil.afl" is an included file used to create MIDI files. It's used by the samples below.

`midirand.afl  randombells.mid`

"midirand.afl" is a program that outputs a MIDI audio file consisting of bells played in a random sequence, as if blown about by gusts of wind. "randombells.mid" is the output of one run of this program. (It's a 79 minute sequence.)

`gymnopedie3.afl  gymnopedie3.mid`

"gymnopedie3.afl" is a program that creates a MIDI rendition of Eric Satie's 3rd Gymnopedie piano work. The program text consists mostly of the music notation – calling a midiutil utility to translate the notation to MIDI. "gymnopedie3.afl" is the output of of the program.

The MIDI files can be played by most audio utilities. Just click on one and there's a good chance it'll play. (I prefer Apples iTunes to Microsoft's Media Player – the former sounds a lot nicer, especially for the bell music.)

`xmlparsers.afl`

A very small, simple, doesn't-do-much XML parser that can be used as a "data" (DOM-like), "push" (SAX-like) or "pull" parser, with examples of different uses. The program contains its own test data. This program is intended to illustrate one advantage of a language that supports true coroutines and generators: it makes interfaces of different kinds easy to implement.

The include files and samples are not well commented at present. The include files are described in detail in the (`www.wilmott.ca/afl/afloverview.html`) AFL Overview.

Enjoy.

Sam

13 November 2005