A guide to fl plugins

It is possible to extend the functionality of the fl language through the use of *plugins*. This document serves as a guide to using such plugins, as well as writing your own.

Using plugins

What is a plugin?

A plugin is a *shared library* (.so file), exporting a set of types and functions for use from within the fl language. While a plugin may technically be written in any language, we currently only officially support plugins written in Haskell, against the Haskell bindings provided in the VossII source distribution.

Loading a plugin written in Haskell normally require `libgmp` to be installed on the system.

Loading a plugin

Like `.fl` source files, plugins need to be loaded before they can be used. To load a plugin, use the `load_plugin` function from an fl source file or the fl REPL:

```
: load_plugin "hello";
"Hello World"
it::string
: hello "Mike";
"Hello, Mike!"
it::string
```

In the above example, `load_plugin` tries to find a file by the name of `libhello.so`, either in the same directory as the calling source file (or the current working directory, if called from the REPL), or in the VossII library directory.

If found, the file is loaded and the *plugin name* – which is not necessarily the same as the *file name* is returned. This name is important, because once fl has loaded a plugin with any given name, it will refuse to load another plugin with the same name.

Once a plugin is loaded, all of its exported types and functions are brought into scope, and can be used from fl just like any types or functions defined in the fl language itself.

At the time of writing, it is not possible to programmatically query a plugin about which types and functions it exports. Instead, refer to the documentation of each respective plugin.

Writing your own plugin

Requirements

While plugins may be used with only a binary installation of the VossII system, writing them requires a recent (>=8.2) version of the GHC Haskell compiler and a source distribution of the VossII system.

Plugins are (loosely) tied to the particular version of fl they are built against, and will not load if fl detects a version incompatibility. If you are using a binary distribution of VossII that you did not build from said source distribution, you must ensure that the version of the source distribution matches that of the binary distribution.

Setting up your environment

Once you meet these requirements, run `make -C src install_plugin_tools` from the root directory of the VossII source distribution. The Haskell support libraries will be installed globally for your user, and a tool called `create-fl-plugin` will be installed into your fl binary directory. By default, this directory is found at `<VossII root directory>/bin`.

To create a new plugin, create an empty directory and run the `create-fl-plugin` tool from within it. You will be asked to provide some metadata for your plugin, after which the tool will set up all the necessary files to start writing your plugin.

Compiling your plugin

To build and install the plugin you just created, run `make install` from the root of your plugin directory, then start the fl REPL. If everything went according to plan, you will be able to load your plugin by calling `load_plugin "whatever you named your plugin";`, and then test it by calling `hello "your name";`. The REPL should respond with `"Hello, your name!"`.

Congratulations, you're now ready to actually start developing your plugin!

Developing plugins

This part assumes a basic knowledge of Haskell on part of the reader, which can be picked up through any beginner tutorial such as the excellent Learn You a Haskell for Great Good.

If you open up the `src/Main.hs` file created by `create-fl-plugin`, you will see something like the following program:

```
module Main where
import Language.FL.Plugin

foreign export ccall "fl_init" fl_init :: FlInit
```

```
fl_init :: FlInit
fl_init = flInit main

main :: IO Plugin
main = return Plugin
  { pluginName = "hello"
  , pluginVersion = 0
  , pluginExports =
    [ export "hello" $ \name -> "Hello, " ++ name ++ "!"
    ]
  }
```

Let's take it apart line by line, to get an understanding of what it does and how you can modify it to do your own bidding.

```
module Main where
import Language.FL.Plugin
```

This is where we declare the `Main.hs` module to be the entry point of our program, and import the fl plugin library. The only interesting thing here (because you *did* read the Haskell tutorial, didn't you?) is the name of the module to import. For more information about the functionality provided by this module, see the Haskell plugin API documentation in `doc/fl-plugins/index.html`.

```
foreign export ccall "fl_init" fl_init :: FlInit

fl_init :: FlInit
fl_init = flInit main
```

This is where things start to get interesting. Because plugins are shared libraries, we need to export some function for the fl interpreter to call when we attempt to load the plugin. This function is called `fl_init`, and performs some rather shady black magic under the hood. Fortunately, you don't need to deal with this magic yourself. All you need to do is to call `flInit` with a function producing a `Plugin` structure as its argument – `main` by default – and then export the result under the name `fl_init`.

Don't worry if you didn't get all of that; just leave this part of the plugin alone and you will be fine.

```
main :: IO Plugin
main = return Plugin
  { pluginName = "hello"
  , pluginVersion = 0
  , pluginExports =
    [ export "hello" $ \name -> "Hello, " ++ name ++ "!"
    ]
  }
```

This is where we actually define the functionality to export to fl. A plugin is a

value of type `Plugin`, which contains some metadata – the name and version of your plugin – and a list of functions that should be exported to fl.

The plugin produced by `create-fl-plugin` exports a function called `hello` to fl, which when applied to the name of a person returns another string greeting the given person.

Exporting functions

Each entry in the list of exports is defined using the `export` function. The API documentation gives the type of `export` as `FlExport a => String -> a -> Export`. While the technical details of the `FlExport` class are a bit on the hairy side, intuitively `export` takes a name and a plain Haskell function. As long as the given function a) is monomorphic, and b) only mentions types understood by fl in its type signature, it will be made available to fl under the given name when the plugin is loaded.

If those two requirements are not fulfilled, you will get a type error when attempting to compile the plugin. For a list of types understood by fl, and information on how to make your own types available to fl, see the section on Types in Haskell and fl.

Of course, exported functions need not be defined in-line like the example function. They may be declared anywhere, as long as they are in scope when producing the plugin structure. For instance, the example above could just as well have been written like this:

```
greet :: String -> String
greet name = "Hello, " ++ name ++ "!"

main :: IO Plugin
main = return Plugin
  { pluginName = "hello"
  , pluginVersion = 0
  , pluginExports =
    [ export "hello" greet
    ]
  }
```

Plugin safety and errors

An exported function's fl type is determined by its Haskell type, and arguments and return values are automatically marshalled between the fl and Haskell worlds. Any Haskell exceptions or errors (including those raised using the `error` function‘) that occur either during marshalling or execution will be caught and converted into an fl failure.

This means that you can rest assured that your plugin will not crash the fl interpreter, and your programs will be just as type-safe as if they were written

in fl alone.

Arguments to plugin functions are always fully evaluated. While this may be a bit wasteful, performance-wise, it ensures that no argument will ever throw an fl error to Haskell land during marshalling.

At the time of writing, function names are *not* checked for conformity with fl's parsing rules. This means that you're free to name your functions whatever weird thing you like and the plugin will still load just fine, but you won't be able to call those functions from fl.

Errors and higher order functions

Fl plugins may export *higher order functions*. That is, functions which take other functions as arguments. As any fl function may decide to fail, an exported function taking one or more functions as arguments should take care to handle any errors raised by those functions.

If such a function raises an error during evaluation, an `FlError` will be thrown, containing the same error message as the fl error that caused it. `FlError` is the *only* exception that should be thrown by a function argument.

To handle an `FlError`, import the `Control.Exception` module in your plugin and use `try`, `catch`, etc. Any uncaught error will cause the function to fail, propagating the error back to fl once more.

Fixities

In addition to a name and a type, functions exported to fl may also have a *fixity*. By default, functions don't have a fixity. However, if you want to use your function as an operator, you need to give it a fixity.

As an example, the following plugin will export a left associative operator named `+++`, with precedence level 5:

```
add :: Integer -> Integer -> Integer
add a b = a + b

main :: IO Plugin
main = return Plugin
  { pluginName = "hello"
  , pluginVersion = 0
  , pluginExports =
    [ (export "+++" add) { fixity = Infix L 5 }
    ]
  }
```

After loading the plugin, this operator may then be used as any other fl operator:

```
: 1 +++ 1;
```

```
2
it::int
```

Pre and postfix operators are also supported. For details, see the Haskell plugin API documentation.

A note about dependencies

Note that the plugins produced by the default build flow are, at the time of writing, not suitable for binary distribution. The standard GHC compiler will not allow shared libraries to statically link against other Haskell libraries, which means that the plugins so produced will depend on the presence of several system-specific Haskell libraries.

To produce plugins for binary distribution, a patched GHC distribution such as ghc-sd is needed.

Types in Haskell and fl

fl types in Haskell

Out of the box, several of fl's basic types have Haskell equivalents. At the time of writing, those types include:

- `int`: `Integer`
- `float`: `Double`
- `string`: `Text` or `String`
- `bool`: `Bool` (passing fl formulas other than `T` or `F` is an error)
- `void`: `()`
- `a##b##...`: `(a, b, ...)`
- `a list`: `[a]`
- First and second order functions over any of the above types.

Functions may be either pure, or compute their return value in the `IO` monad. Thus, exported functions may have side effects, just like some of fl's built-in functions.

Custom types

When a function exported by a plugin refers to a type which is not recognised by the fl plugin library as an "ordinary" fl type (i.e. the ones outlined above), that type is automatically exported to fl as a custom "black box" type.

The type seen by fl code will have the same name as the Haskell type, except:

- it will be prefixed with `hs_`,
- any spaces in the type will be replaced by underscores, and
- all non-alphanumeric (or underscore) characters will be removed from the name of the type.

For instance, the Haskell type `Foo Bool` will look like `hs_Foo_Bool` to fl. Note that only monomorphic types are currently exportable to fl. For this reason, a polymorphic type such as the aforementioned `Foo` may be exported multiple times; once for each instantiation of its type variables.

As fl will refuse to load a plugin exporting a type with a name that was already exported by another loaded plugin, care must be taken to keep type names reasonably unique.

The `FlType` type class

Every exported type needs to implement the `FlType` type class, which describes how to marshal values of the type, compare values for equality, etc. Fortunately, this type class has sensible defaults for all of its operations, so all you normally need to do is to make your type an instance of `Eq`, `Show` and `Read`, and then use the default implementation of `FlType`.

As an example, following plugin implements a very simple vector library for fl:

```
module Main where
import Language.FL.Plugin

foreign export ccall "fl_init" fl_init :: FlInit

fl_init :: FlInit
fl_init = flInit main

data Vec2 = Vec2
  { x :: Double
  , y :: Double
  } deriving (Eq, Show, Read)
instance FlType Vec2

main :: IO Plugin
main = return Plugin
  { pluginName = "vec2"
  , pluginVersion = 0
  , pluginExports =
    [ export "vec2" Vec2
    , export "add" $ \(Vec2 x1 y1) (Vec2 x2 y2) -> Vec2 (x1+x2) (y1+y2)
    , export "magnitude" $ \(Vec2 x y) -> sqrt (x*x + y*y)
    ]
  }
```

Note how the prerequisite type classes for `FlType` are derived automatically and no implementation is given for `FlType` itself. Normally, this is the only thing you need to do to introduce your own types.

As an added bonus, types created this way are represented to fl as an opaque pointer, meaning that they do not need any conversion when passed from fl to Haskell and back again.

Overriding the defaults

If you really need to, you may implement part or all of the `FlType` class yourself. For instance, if your type can grow large you may not want to use `Read`/`Show` for de/serialization, and you *certainly* don't want to use `Show` for pretty-printing.

To override these functions, you will need to import the `Language.FL.Plugin.Internal` module in your plugin, alongside the usual `Language.FL.Plugin`. For more information on what the methods of `FlType` do, and the low-level fl plugin API at your disposal to override them, see the Haskell plugin API documentation, which can be found in `doc/fl-plugins/index.html`.