Dynamic Linking in Haskell

Hampus Ram
d00ram@dtek.chalmers.se

1 Introduction

Today many applications can be enhanced by the use of so-called plugins, pieces of code that can be loaded into an application at any time. These plugins often come in the form of dynamically linked libraries (shared objects in UNIX-speek) and until very recently there has been no common way to use such libraries in Haskell programs. Now support for this exist in the Hierarchical Libraries as the `System.Posix.DynamicLinker` module.

However, this module only exports the operating-systems underlying functions for loading dynamic libraries, which are designed to load C-functions. Creating and using such modules written in Haskell adds extra problems.

What is needed is thus some other way to load code dynamically without the problems associated with using functions designed in first hand to work with C-code.

2 GHCi and dynamic loading

Enter GHCi. The interactive version of the Glasgow Haskell Compiler (GHC) has for some years been able to dynamically load its own object files and execute their code. Fortunately for us the functions doing this lies within the GHC runtime system (RTS) and is thus available to us in GHC-compiled programs as well.

The GHC RTS contains the following functions[1] that is used by GHCi to dynamically load functions: `loadObj, unloadObj, resolveObjs` and `lookupSymbol`.

To use a compiled module it is loaded with `loadObj` and then the functions are resolved with `resolveObj`. When that has been done functions can be loaded using `lookupSymbol` which will yield a pointer to the function. When the module is needed no more a call to `unloadObj` unloads it.

[1] Defined in `ghc/rts/Linker.c`

3 Design of a simple linker

Using the functions from the GHC RTS one can begin to structure a simple module that captures the basic functions that one needs to load objects dynamically.

Some of the most basic things one must be able to do are:

- Load packages as well as modules
- Use different paths and file extensions
- Functions must be real Haskell functions

The first demand is necessary because of the fact that a compiled module doesn't contain functions imported from the Prelude and other standard libraries and thus one must load the GHC base packages containing those functions oneself.

The second demand is due to the fact that this library is to be used by many different programs in need of plugin support. Thus one might want to have plugins end in something else than ".o" and of course those will be located in different places depending on the application.

The third comes from getting tired of using FunPtr:s when dealing with functions loaded with the `System.Posix.DynamicLinker`. Having a real Haskell-function is much better than having a function-pointer.

Of course there is some implicit demands on this library that cannot be set aside, such as the need to correctly handle hierarchical libraries and working on different platforms.

4 Preexisting software

There exist one library today that implements functions that exploit the GHC RTS to dynamically load object files. That library is the RuntimeLoader by Andre Pang. However that library is not much more than a simple wrapper around the RTS functions and is therefore somewhat hard to use. It also is totaly unaware of hierarchical modules which is, as stated before, a too big feature to not be included in such a library.

5 Implementation of a simple linker

First one need the functions exported by the RTS and they are introduced by using the Haskell Foreign Function Interface (FFI), then most work is really writing glue code to translate between the C-world of the RTS and the Haskell-world.

However one major design decision was made during this stage, namely to logically separate packages and modules. This since packages are really collections of modules and have more of a support-role than anything else. For instance you really do not want to load any functions from a package since you could use it directly in your program anyways.

The module will thus export the following: Types for dynamic modules and packages. Functions to load and unload such and a function to load functions from a module. Furthermore the module provides functions that return the path to the loaded modules and packages which can help debugging or if you want to implement so called *crisp loading*. Last, the module exports a function for loading dynamically linked libraries (shared objects) since sometimes Haskell packages needs them.

A simpler way to deal with situations when dynamically linked libraries are needed are however to link those libraries into the main program in the first place thus avoiding unecessary logic and if you really need control over your libraries, please use `System.Posix.DynamicLinker` instead.

When loading modules care must be taken to make sure that any hierarchical libraries are threated correctly and an equal care must be taken when loading libraries to ensure that any supporting cbits-packages are loaded too.

All functions will throw exceptions if anything fails and, unfortunately, they might write some messages to the standard error whether you catch the errors or not since the GHC RTS will output some diagnostics for certain errors.

6 Improving the simple linker

The simple linker provides basic functionality needed to dynamically link in Haskell modules into your program, but it does not provide any intelligence or added value. In a real program one probably needs to implement quite a few more things to simplify coding. A better library needs to be written.

Talking with friends of mine planning to write an implementation of Erik Meijer and Daan Leijen's Haskell Server Pages, a project that will rely heavilly on dynamic loading, a number of features needed in addition to those provided by the simple linker came up.

The most prominent were:

- Safe loading
- Thread safety
- Crisp loading
- Dependency chasing
- Cascading unloading
- Automatic symbol resolving

Safe loading is simply that loading a module twice should not throw an exception as in the simple linker but instead return the allready loaded library. Also if you load a module twice one call to unload it should not really do so since it might yet be needed.

Thread safety should be implemented to guarantee that two threads does not interfere with eachother by for example trying to load the same module at the same time.

Crisp loading means that if the modules file on disk has changed since last loaded it should be reloaded to provide the latest functionality.

Dependency chasing means that you let the system load dependencies by itself thus letting the programmer worry about other things. This is however, as we will see, not a pice of cake to implement as simply as one would like.

Cascading unloading is quite similar to dependency chasing and has the same problems when it comes to implementation. When a module is unloaded all modules it depends on should also be unloaded, unless of course they are needed by other modules still loaded.

Automatic symbol resolving is just what is says. Unlike the simple linker with which you need to call a separate function when you want to resolve functions this should be done automatically when needed.

7 Implementation of a smart linker

To implement a smarter linker with the above properties was not a simple thing to do. Some compromises had to be done to not make the code overly complicated and unportable.

Safe loading was quite simple to solve by using a state in which all loaded modules reside and then reference count them. For each load the reference count was increased and for each unload decreased. Only when the module cant't be found in the state it is

loaded and only when the count reach zero the module really is unloaded.

Thread safety was also quite simple to achieve. By storing all state in an MVar and taking hold of that MVar in each public function one is guaranteed that only one function at a time is doing dynamic linking, at least as long as only one MVar is created (which is not guaranteed).

Crisp loading is implemented by a reload-function that optionally can perform this reload only if the module has been changed since last loaded. This is implemented by storing the last known modification time in the state and comparing with it.

Dependency chasing and cascading unloading is not very simple to do automatically since all information that you can get from a object file is which functions it export and which it has undefined, not where to find the objects/packages containg the undefined references. It would be too much trouble implementing this, especially since the format of object files are platform-dependent. GHCi solves this by looking at the .hi-files that the compiler emits but this library takes a simpler way out.

It does not support true dependency chasing, but lets the programmer specify dependecies between modules and packages and then automatically looks up those when loading a module. The dependencies are then saved in the state and used when unloading a module ensuring that everything is unloaded correctly.

When it comes to automatic symbol resolving one could do many things, for instance one could resolve functions each time one tries to load a function since it is first then symbols need to resolve. This is however not very good since this can lead to many unecessary calls to resolve functions. What is really done is resolving functions after a module has been loaded at top level. Since all dependencies should have been loaded by then all functions should resolve.

Unlike the simple linker no distinction is made between modules and packages, mostly due to simplicity, but since the underlying library is the simple linker one still can't load functions from packages. Dynamically linked libraries are however still treated the same way and can thus only be loaded and nothing more.

The state contained in the MVar is the loaded modules stored in a hash table, a dependency map stored in a hash table and some paths and other strings used in constructing the real paths from module and package names. The use of hash tables have many purposes, firstly it provides a quite fast lookup method, secondly copying the state only copies pointers to the tables so it will be resonaby fast compared to if FiniteMaps or similar were used, lastly they eliminate the need for updating the state thus providing a consistent state if any functions would fail.

The state could have been provided implicitly by embedding it in a state monad but this has not been done due to the fact that it probably would be necessary to embed large parts of the program in this new monad leading to the need for heavy lifting of the normal IO operations.

8 Further improvements

One thing that is lacking in the smart linker is direct support for loading modules using paths directly as can be done in the simple linker. However it is not simple to add such a feature to the current library since it would be very hard, or at least confusing, when it comes to resolving dependencies with a mixture of absolut paths and qualified names to deal with.

Further improvements would be adding functions to inspect the current state and perhaps some to alter it in bigger ways such as unloading all currently loaded modules.

9 Examples

Here are two very simple examples highlighting the differences between the two libraries, the simple linker (DynamicLinker) and the smart linker (SmartLinker). Both examples loads a module Foo.Bar that depends on the base library and then loads the function foo from that module. The most important thing to note is the lack of need to explicitly load and unload the base package in the example with SmartLinker. Also there is no need for explicit function resolving.

```
import DynamicLinker

main = do basep <- loadPackage
                        "base"
                        (Just ppath)
                        Nothing Nothing
          modm  <- loadModule
                       "Foo.Bar"
                       Nothing Nothing
          resolveFunctions
          func  <- loadFunction modm
                           "foo" :: IO Int
          print func
          unloadModule modm
          unloadPackage basep
      where ppath = "/usr/lib/ghc"
```

```
import SmartLinker

main = do env <- createSmartEnvironment
                        (Nothing, Nothing)
                        (Just ppath,
                         Nothing,
                         Nothing)
          addDependency env
                         "Foo.Bar"
                         "base"
          modm <- loadModule env "Foo.Bar"
          func <- loadFunction env modm
                           "foo" :: IO Int
          print func
          unloadModule env modm
     where ppath = "/usr/lib/ghc"
```