9th European BSD Conference (EuroBSDCon 2010)
Karlsruhe, Germany, October 8–10, 2010.

What Functional Programming Can Do for FreeBSD and Vice Versa?*

Gábor PÁLI[1], Giuseppe PILICHI[2], Ashish SHUKLA[3]

Department of Programming Languages and Compilers,
Eötvös Loránd University,
The FreeBSD Project
[1] `pgj@FreeBSD.org` [2] `jacula@FreeBSD.org` [3] `ashish@FreeBSD.org`

Abstract

In the last decades, scene of functional programming has changed dramatically as many new compilation and programming techniques were introduced, and then became available. Many research efforts put into development of better languages and compilers is focused in Haskell, a standardized, general-purpose, lazy, pure functional language. It supports several features to make today's software engineering easier with a minimal loss in performance. This paper investigates the possibilities on how to employ these features in supporting the development of an "unknown giant among free operating systems", namely FreeBSD. On the other hand, FreeBSD, as a descendant of AT&T UNIX via the Berkeley Software Distribution, represents reliable and stable UNIX-compliant internals and system APIs, which might help to keep development of the de facto standard Glasgow Haskell Compiler in a reasonably good shape. We believe that these open source projects would be able to help each other in a promising way.

Keywords: Functional Programming, Operating Systems, Software Engineering, Static Analysis, Haskell, Software Portability

1. Introduction

Programming languages with static typing tends to be more secure, reliable, and robust, albeit sometimes they are also harder to master. Development seems to be even more difficult when a programming language is also *pure*, i.e. it tries to isolate and control side effects of computations. However, it encourages software developers think in a different way, focusing on the solution in a rather abstract way, most of the time in terms of mathematical or computer science concepts. By abstracting away from the details of gritty details of implementation, very quick

*Supported by POSDRU/6/1.5/S/2008

prototyping becomes possible with many of the boilerplate work generated or added automatically.

We are confident that Haskell represents a great power in the right hands that can efficiently support real-world software engineering. But it can only be accessed if it is ported and carefully prepared for daily use. Beyond porting and packaging, we present a detailed case study on how to turn Haskell into a valuable tool for FreeBSD in this paper. Contributions of our work presented here could be summarized as follows.

– We have recently started to support the development of the Glasgow Haskell Compiler directly, and therefore we have earned many interesting experiences in making its implementation more robust and more compatible with standards (Section 2.1).

– A preliminary framework for porting Haskell Cabal packages has been also introduced, featuring `hsporter` (Section 2.2). This tool directly translates Cabal descriptions to ports. These are then tested, packaged, and distributed using the well-established methods in the FreeBSD Project, resulting in a stable binary package support.

– Haskell might be a good candidate for building static analysis tools, because it is a high-level language and sources compiled to native code compete with speed of decent C implementations while the time spent of development is shorter and the reliability of the resulted program is better. We show an example of this (Section 3).

2. FreeBSD in Development of Haskell

The Glasgow Haskell Compiler, also known as GHC, is the most supported and sophisticated Haskell compiler to date. It also represents the *de facto* standard for the Haskell programming language. GHC and Haskell are under continuous research and development, and the goals include high-performance implementation of concurrency and parallelism [1]. There is an active community around the language, and more than 2,200 third-party open source libraries and tools are available in the on-line package repository, called Hackage [2]. GHC is both an interpreter and a native-code compiler that runs on many platforms, including FreeBSD.

Haskell is increasingly being used in commercial applications, and GHC is often serves as a testbed for advanced functional programming features and optimizations. There are many popular Haskell software, for instance *Darcs*, a revision control system with several innovative features, *xmonad*, a tiny window manager for the X Window System, *pandoc*, a swiss-army knife for converting between many widely-used markup formats, and various web frameworks, like *Snap* or *Happstack*, competitors to classical web servers. Note that most of these software are already present in the FreeBSD Ports Collection. There are companies behind the development of Haskell, for example Galois, which develops high assurance software for demanding applications, and it currently runs Hackage.

2.1. Porting the Haskell Compiler

At the moment, GHC is mainly developed on Linux, Windows, and Mac OS X under the 3-clause BSD license. It has a FreeBSD port since August 1999, and `amd64` support on FreeBSD was introduced by porting version 6.8.3 in July 2008. It has been updated to version 6.10.4 in September 2009. From the beginning of 2010, upstream patches from FreeBSD developers have started to appear, and support for dynamic libraries on FreeBSD have been added to version 6.12.1. In April 2010, FreeBSD has become a Tier 1 (actively sponsored) level platform of GHC by offering vanilla binary 6.12.x distributions for FreeBSD 8.X and 7.X systems.

We have installed builder clients for the Glasgow Haskell Compilation System on FreeBSD `8.0-RELEASE i386` and `amd64` systems. These builders are written in Haskell, and they are controlled from Galois in order to provide nightly builds of GHC for their development (`-HEAD`) and release (`-STABLE`) branches. The development of GHC happens in a Darcs repository, which is checked out on each occasion and a daily snapshot is built and tested against a compiled test suite, maintained by the developers. The results of these tests help to show the problems springing up in the daily development, and they are also good indicators of usability on the supported platforms. The test suite also plays an important role in the so-called *validation* process which consists of applying a patch (usually a modification sent by contributors) and performing a test build, running the tests to check for correctness. Daily binary snapshots are useful for users who want to try the latest, cutting-edge version of the compiler without requiring them to compile the sources ourselves. Builds of `-STABLE` are used for publishing purposes, since a GHC release is just a carefully selected, automatically packaged snapshot of that branch.

Some parts of the compiler and the accompanying run-time system are written in C, but an existing Haskell compiler is used for bootstrapping the compilation. These bootstraps are present in form of pre-built tarballs for each major FreeBSD versions. There is an alternative compilation method that does not involve another Haskell compiler and based on using a C compiler only, but it is not usable at the moment. GHC has a refined build system, based on Perl, GNU autoconf, make, and Python 2.6. It is very developer-friendly, and it is easy to use, however sometimes we needed to put in a word against the creeping Linuxisms in the sources that caused the implementation of build scripts to move towards Perl. We had problems with respecting the non-standard installation prefixes, including the default `/usr/local/` directory on FreeBSD systems. It must be also noted that C sources use ISO C99 with POSIX extensions, but inconsistencies between GNU libc and FreeBSD libc did not allow us to define the corresponding macro settings in a uniform way. GHC performs several handcrafted optimizations on the GCC output to make the resulted assembly code more efficient (referred to as *mangling*) which is a relatively fragile part of the entire build process. C code compilation also tries to make use of visibility pragmas present in recent GCC versions, but it is failing to work with the FreeBSD system compiler so far.

According to the test case failures, the FreeBSD port of GHC still lags a bit behind the Linux version, but it has definitely improved in the last months, and hopefully this tendency will continue in the future. Note that FreeBSD seems to be more strict than Linux in many aspects, which might mean that there are still hidden problems even in that version. It is supported by a bug which was found in the garbage collector and lead to an excessive memory leak on FreeBSD but not on Linux. In summary, GHC is up-to-date and usable on FreeBSD.

2.2. Porting Haskell Cabal

Haskell comes with many third-party packages to extends its functionality, called Cabal packages or *hackages* for short. These hackages usually contain libraries or modules supporting software development in Haskell, but one might find complete applications wrapped in such packages. Cabal is a standard Haskell library that comes with GHC, and it has an optional user interface, `cabal-install` that might be considered a package manager in the Haskell world, although it does not fulfill all standard expectations. For example, it does not support removal of the previously installed packages, and it only works by compilation of source code. On the other hand, it is able to handle dependencies, and maintains a central package list together with a used-based database of persistently stored versions, supports various build modes and generation of documentation (via the Haskell documentation tool, `haddock`). Therefore it is indeed a handy utility in addition to the advanced multi-source compilation mechanism of GHC, which makes use of Makefiles unnecessary.

2.2.1. Translation

We have started to build a framework around the features implemented by Cabal, and we recently introduced a BSD make include files in the Ports Collection, `bsd.cabal.mk` and `bsd.hackage.mk` respectively, to support direct transcription of Cabal package descriptions to FreeBSD ports. (They can be checked out from the FreeBSD Project's CVS repository.) By using the aforementioned include files, size and complexity of Haskell Cabal ports have been reduced to a manageable level. We have written a tool in Haskell, called `hsporter` [3], that is capable of almost automatic production of complete FreeBSD ports out of package descriptions. The development of `hsporter` and `bsd.cabal.mk` is in progress to cover package descriptions correctly while respecting the established traditions in the FreeBSD Ports Collection during the translation. Based on these files, we are also able to run automatic periodic checks to be notified on version changes.

Wrapping is being implemented in `bsd.cabal.mk`. At the moment, it takes care of controlling the process of installation, invocation of `haddock` and `hscolour` for generating documentation on demand, resolving Cabal dependencies to port dependencies, providing the correct package naming and directory paths. The task of the `hsporter` tool is just to create the corresponding `Makefile`, `distinfo`, `pkg-descr`, and `pkg-plist` files, along with placing the port in the right category. This approach helps us to avoid bloating in ports, although the required logic can be generated by `hsporter`. Making use the Ports Collection better also results in easy-to-overview ports without demanding knowledge of Haskell.

For example, if we want to port the `TypeCompose` hackage, we will need to issue only one command (Figure 1). This will generate the `devel/hs-TypeCompose` directory, containing the port. Obviously, manual checks and refinements are always recommended, but as it can be seen here, vast of the porting work boilerplate is automatically done. Note that it might be improved further to be *correct by generation*, i.e. instead of checking for correctness of the result, the quality of translation itself will directly determine the quality of the port.

```
$ hsporter http://hackage.haskell.org/packages/archive/TypeCompose/0.8.0/TypeCompose.cabal devel
Fetching http://hackage.haskell.org/packages/archive/TypeCompose/0.8.0/TypeCompose.cabal...
Fetching http://hackage.haskell.org/packages/archive/TypeCompose/0.8.0/TypeCompose-0.8.0.tar.gz...
Creating directory devel/hs-TypeCompose...
Conversion in progress... [ Makefile distinfo pkg-descr pkg-plist ]
Do not forget to do a 'portlint -C'
```

Figure 1: Porting the `TypeCompose` hackage with `hsporter`.

2.2.2. Packaging

An own custom Ports Tinderbox is installed to do quality assurance and build binary packages for all active FreeBSD branches on both `i386` and `amd64` architectures. This way we can work around the missing binary package and uninstall support of Cabal while carefully engineering the results. All ported hackages are tested and therefore guaranteed to work with the GHC in the ports tree. This requires us to write patches against the original upstream version sometimes, because certain modules cannot work with each other or the in-tree GHC without a little bit of fine-tuning.

```
# pkg_add -r hs-TypeCompose
Fetching ftp://ftp.freebsd.org/pub/FreeBSD/ports/amd64/packages-8-stable/Latest/hs-TypeCompose.tbz... Done.
Reading package info from stdin ... done.
Writing new package config file... done.
```

Figure 2: Installing the ported binary package of `TypeCompose`.

This set of FreeBSD-ported hackages can be seen as an informal “FreeBSD Haskell Platform” at the user's service. Using the package distribution network of the FreeBSD Project, anyone can install and update GHC together with the important Cabal packages quickly (Figure 2). However, there is an on-going standardization process amongst important third-party packages, under the name of “Haskell Platform”. It incorporates only a relatively small but fundamental subset of hackages, including `cabal-install`, and all the other hackages can be installed via it. We are trying to avoid enabling the users to use Cabal directly as it interferes with the Ports Collection at the moment, but it might be possible to use

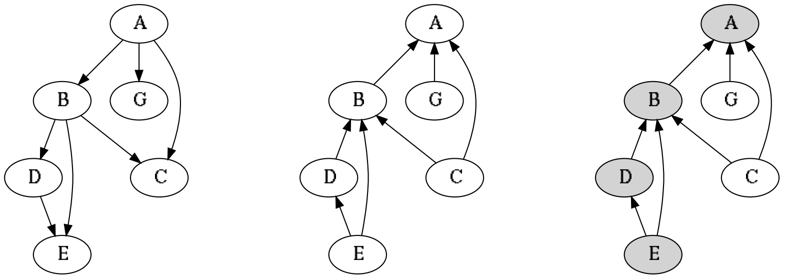
Haskell Platform on FreeBSD as an alternative solution. The latter would be possibly preferred by professional bleeding edge Haskell users who do not want to wait for the FreeBSD version of packages.

3. Static Analysis of the FreeBSD Ports Collection

With growing number of ported modules and a maintained port of the compiler, Haskell slowly goes unleashed on FreeBSD, only waiting for to be applied in practice. Because of its interpreter front-end, the GHC interactive environment, called `GHCi`, development in Haskell is easy and simple as it allows on-line construction and debugging of programs. It is not even mandatory to compile Haskell sources, programmers can write scripts to be run with the `runhaskell` command, and the same program can be compiled to native code for efficiency, which is a great method of prototyping and solving problems.

3.1. Ports as a Graph

An example of such a program might be a reverse dependency calculation between FreeBSD ports. The authors found this program quite useful when they had to determine which ports are affected by modifying a given one in order to test the change in question properly. The discussed program calculates *all* affected ports, i.e. clusters the ports in the tree having a build dependency on the touched one.



(a) Normal port graph with build dependencies. (b) Reversed port graph with build dependents. (c) Calculated effect of port E.

Figure 3: Ports and their dependency relations represented in a graph.

It uses the dependency information given in the ports' Makefiles, referred to as a *describe file* that can be generated by invoking the `describe` target for all FreeBSD ports. This `make(1)` target builds up a description line for the given port, including its build- and run-time dependencies. We used only the build dependencies in the program so far. One might have the intuition that it is enough to `grep(1)` over these lines to find the corresponding ports, but note that determining all the ports requires a more precise solution, like tracking indirect dependencies between ports, which would result in a more complicated and rather inefficient implementation. We feel important to make this difference to have the correct testing coverage. Some build dependencies (typically shared libraries) might be built, but it cannot be told whether their dependencies will be able to build as well. It can be seen that this leads to checking arbitrary level of reverse dependencies that be can represented as a sub-tree.

Thus the Haskell implementation is based on the assumption that the Ports Collection can be represented as a directed acyclic graph, where ports are the nodes, and edges are the dependency relations (Figure 3). Fortunately, a graph module, `Data.Graph` [4] is immediately at the implementers' hands as it can be found the base Haskell libraries. Hence all we have to do is to parse the describe file, turn the ports tree into a graph, and query which nodes can be reached from the one representing the changed port. The complete source code can be reached at [5], some implementation details are omitted for the sake of simplicity.

```
type Name              = String
type BuildDependencies = [Name]
type Port              = (Name, BuildDependencies)
type Ports             = [Port]
type PortGraph         = Graph
```

Figure 4: Basic abstractions for solving the problem.

Building the program is surprisingly intuitive. We only need to pick the right abstractions (Figure 4). Here comma-separated items in parentheses are tuples and items in square brackets are lists of elements of the same type. That is a port is written as a binary tuple of a name and build-time dependencies (as list of names), and ports as list of such tuples. `Graph` is just an abstract data type. We defined a `|>` operator to introduce a data-flow-style programming: there are "pipes" created in the program similar to the UNIX pipes, but these ones are strongly typed, i.e. their invalid combinations result in compile-time errors. Pipe elements might be also "fused" together during the compilation, which is a source of efficiency. There is also a dot (`.`) operator, which is technically the same, but it combines functions in a reverse order, and it is more handy in certain cases. As a result, the source code gets a shell-script-like look, however, this is not only style of writing Haskell programs.

With the powerful pattern matching facilities of Haskell, reading the file becomes trivial. The `portify` function (Figure 5) takes a describe file as an argument in binary format, and breaks into individual lines by `lines`, and turns every line into a port by `toPort`. Note that since there is no computational dependency between the operations, it can be even done in parallel. The processing function called `toPort` matches a pattern on its argument (a list of fields received from

```
portify describe
  = describe |> lines |> map toPort
  where
    toPort line = line |> split '|' |> parse
    parse (package : path : prefix : comment : pkgdescr :
      maintainer : categories : _ : _ : _ : bdepends :
      rdepends : www : _) = (name, builddeps)
        where name      = convertPath category path
              category  = firstOf categories
              builddeps = extract bdepends
              firstOf   = head . split ' '
              extract   = map (convertPath category) . split ' '
```

Figure 5: Parsing the describe file: the `portify` function.

`split`) and lifts the required information, `name` and `builddeps`. Here we construct the former out of the primary category and the name of the port, and the latter out of list of dependencies.

```
generateGraph ports
  = ports |> map toEdges |> graphFromEdges
  where toEdges (name, deps) = (name, name, deps)
```

Figure 6: Building a graph: the `generateGraph` function.

The graph can be generated from the list of ports, using the `graphFromEdges` function (Figure 6) with a `toEdges` helper function, which doubles the `name` field: first it is a label for the node, second it is vertex. Dependency calculation is defined by the `depends` function (Figure 7). It simply calls the `reachable` function to cluster the affected nodes, and it uses two helper functions to convert between the internal representation of the graph (`fromVertex` and `toVertex`, respectively).

```
depends graph port fromVertex toVertex
  = vertexed |> reachable graph
             |> map (translate . fromVertex) |> sort
  where
    vertexed = port |> toVertex |> Data.Maybe.fromJust
    translate (name,_,_) = name
```

Figure 7: Calculating dependents: the `depends` function.

Finally, `main` combines the introduced functions in a sequential, imperative style. It gets the command line arguments, assigns them to the corresponding names (`describe` and `port`), reads the file, builds the graph, transposes (reverses) it, and collects the members of the cluster. As a last step, concatenates the list of port names into one string with `unlines` and prints it.

```
main = do
  args <- getArgs
  let describe = args !! 0
  let port     = args !! 1
  contents <- readFile describe
  let (h,f,g) = contents |> portify |> generateGraph
  let graph   = transposeG h
  depends graph port f g |> unlines |> putStrLn
```

Figure 8: Combining everything: the `main` function.

3.2. Evaluation of Costs

As it can be seen, the written code is abstract, captures the problem to be solved by a concrete mathematical concept. It is easy to understand as well as to maintain. According to our measurements (Figure 9), it is also efficient to run, since it is only *2 times* slower than the corresponding implementation in C [6] when it is compiled to native code. The performance evaluation was done with `time(1)` on an Intel Core2 Quad CPU Q8400 @ 2.66GHz running `FreeBSD/amd64 8-STABLE`. We used the FreeBSD port of GHC 6.12.3 and the FreeBSD system compiler, GCC 4.2.1 with compiler optimizations (`-O` for GHC, `-O2` for GCC) enabled. The C version uses portions of the `make_index` utility for `portsnap(8)` and the ported version of the *igraph* library (version 0.5.3)[1]. The result length is approximated because it depends on the actual state of the ports tree. It does not seem to affect on the performance, it is presented for demonstration purposes only.

The sources were measured by `sloccount(1)`[2] (version 2.26, default settings), where the difference in cost of implementation is conspicuous: the Haskell version is below 100 lines (SLOC: 67) while the C version is above 400 lines (SLOC: 433). Theoretically, while the C version would take a month to be developed by a single person, the Haskell version would take only about 4 days, costing nearly 7 times less. (As a matter of fact, it took a day for the authors.)

Speed of the Haskell implementation clearly half of the C one in every aspect. There is no definite bottleneck, the ratio can be taken constant. It is possibly caused by the difference in the run-time systems and graph implementations, and the fact that *Haskell programs do not contain variables.* Every variable-like entity in the source code is just a name of an expression, i.e. Haskell works with immutable values. This is where the reliability of the code comes from: no value will be

[1]`http://igraph.sourceforge.net/`
[2]`http://www.dwheeler.com/sloccount/`

overwritten as *there is no assignment* in the language. This remarkable difference sometimes makes harder to build efficient implementations, but it also a strong guarantee of correctness and reliability.

Query	Ports	Haskell			C		
		Total	Real	Sys	Total	Real	Sys
`devel/gettext`	11164	1.36	1.27	0.09	0.64	0.59	0.05
`devel/gmake`	10947	1.34	1.25	0.09	0.64	0.59	0.05
`graphics/png`	4577	1.28	1.19	0.09	0.62	0.57	0.05
`lang/ghc`	145	1.32	1.21	0.09	0.61	0.56	0.05
`math/gmp`	972	1.24	1.15	0.09	0.61	0.56	0.05

Figure 9: Average execution times for specific queries.

3.3. Potential Uses

This simple tool can be used in many different ways to assist the work in the FreeBSD Project. Some examples are as follows. There might be similar checks added in the future, based on this concept or by building other static analysis frameworks in Haskell.

- It can be guaranteed that all the ports will be checked if the resulted list of ports are tested for package building. This is how it is usually done with the so-called "exp runs" at the FreeBSD package building cluster to determine the potential problems with a given change in the ports tree. By limiting the number of ports to be tested to the ones that really need testing, an exp run might be shortened, and it also might allow the developers to test similar huge changes themselves without wasting the valuable resources of the package building cluster. We already use this tool to do "mini exp runs" for Haskell ports (i.e. testing about 150 ports as time of writing) to ensure that updates will not break anything in the ports tree, or re-basing of ported hackages to a new GHC is correct.

- Recently it has also proved to be useful in checking whether a port is feature safe[3], since it is just enough to tell many ports are affected by changing the port in question, and set a sensible limit on that (e.g. less than 5). It might be part of the pre-commit check and help to notify the committers, even replacing the current solution.

- It might be integrated to various package building scheduling algorithms. It can be used in QA Tindy[4] to decide what to build to test not only the port itself but its dependent ports, or it can be used in the real package building cluster.

[3]A restriction applied on ports during preparation of FreeBSD releases.
[4]A quality assurance suite for FreeBSD ports.

4. Related Work

Porting the Glasgow Haskell Compiler is in progress for other related platforms, including Mac OS X, OpenBSD, NetBSD, DragonFly BSD, and Solaris with many of them having their own builder clients. To our knowledge, only OpenBSD offers explicit support for porting Haskell Cabal. The original concept of automatic conversion to a given package format comes from Olivier Thauvin, who implemented `cabalmdvrpm` to provide packages for Mandriva. It was followed by other similar porter tools for Red Hat Linux, Arch Linux, Gentoo Linux, Slackware Linux, Debian Linux, so today many popular Linux distributions support hackages in some form. As these solutions are for Linux and most of the hackages are being developed on Linux, they are usually not required to patch the sources in order to make them work. In case of FreeBSD, every freshly imported hackage needs to be checked for building and patched on demand, i.e. it is ported and not simply packaged that cannot be completely automated.

Haskell is already employed in development of operating systems, most notable examples include Linspire, where engineers have been using functional programming in the following tasks: hardware detection and configuration, creation of installation CDs, and internal CGI/web applications. Much of this work was done in O'Caml, but as of April 2006, the developers standardized on Haskell. The Haskell Graphical Interface for Emerge (Himerge) for Gentoo Linux also offers calculating reverse dependencies and fast indexing of packages information. The API for a new iteration of the L4 microkernel, called seL4 is represented by an executable specification [7] written in Haskell. It is the first-ever general-purpose operating system kernel that has been verified formally.

5. Conclusions

We believe that the presented results are not unique to Haskell as they might be applied in merging any software distribution set to the Ports Collection. Translation and wrapping foreign package formats are often needed, and making them automatic and automatically correct is a key to success in properly maintaining a large number of ported packages. Primarily due to the higher level of abstraction and language capabilities of Haskell, we are confident that it is quite suitable for writing various *high-performance tools quickly*. The development of Haskell is surrounded by a very active and supportive community, it is open and there are many tutorials, therefore anybody can learn it. Fortunately, porting the flagship Glasgow Haskell Compiler is free from serious problems, and the builder clients are doing a great job in keeping an eye on occasionally introduced Linuxisms in the sources.

Finally, we would like acknowledge the support of the FreeBSD Donations Team (Daniel Seuffert), the Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007-2013 (POSDRU/6/1.5/S/3-2008). We would like to thank Alexey Dokuchaev and Gábor Kövesdán for their valuable comments, and Maxime Henrion, Oliver Braun, Volker Stolz, and Simon Marlow for their work on porting the Glasgow Haskell Compiler to FreeBSD.

References

[1] Marlow, S., Peyton Jones, S., Singh, S. *Runtime Support for Multicore Haskell* ACM SIGPLAN International Conference on Functional Programming, Edinburgh, United Kingdom. September 2009.

[2] *HackageDB.* `http://hackage.haskell.org/`, August 2010.

[3] *Repository of* ***hsporter****.* `http://code.haskell.org/~pgj/projects/hsporter/`, April 2010.

[4] King, D.J., Launchbury, J. *Lazy Depth-First Search and Linear Graph Algorithms in Haskell*, Glasgow Workshop on Functional Programming, 1994.

[5] `http://www.freebsd.org/~pgj/eurobsdcon2010/depcalc.hs`

[6] `http://www.freebsd.org/~pgj/eurobsdcon2010/depcalc.c`

[7] Derrin, P., Elphinstone, K. et al. *Running the Manual: An Approach to High-Assurance Microkernel Development* ACM SIGPLAN Haskell Workshop. Portland, Oregon. pp. 60–71, September 2006.