The `Helium` Logging Facility

Jurriaan Hage (jur@cs.uu.nl)

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-055

www.cs.uu.nl

Abstract

`Helium` is a compiler for a subset of `Haskell` tailored for giving good error messages. To evaluate the quality of these messages, a logging facility was included within the compiler. In this report we describe aspects of how the logging between the compiler implemented in `Haskell` and a specially built `Java` server proceeds, described mainly from a technical point of view: what is sent, how is it sent, and some of the safety measures we have taken to avoid misuse and abuse. We also touch on the subject of ethics, and relate our general experiences in using socket communication for the purpose.

1 Introduction

The `Helium` compiler implements a substantial subset of the lazy functional programming language `Haskell`, and was built at Universiteit Utrecht, largely by Bastiaan Heeren, Daan Leijen and Arjan van IJzendoorn [3], instigated by the latter. The main motivation for building this compiler was the need for higher quality (type) error messages in order to lower the high threshold that exists for learning `Haskell` as observed by lecturers of functional programming at our institute. `Helium` was used in a number of incarnations of a functional programming course in Utrecht from 2002 onwards.

By design, some elements of the `Haskell` language were not included in order to obtain more easily understandable and intuitive error messages, although later versions of `Helium` support quite a bit more. Consult the compiler website for up-to-date information about what is supported and what is not [4]. Here you can also download the compiler, and if you are a lecturer who would like to contribute to our database of logged programs, be sure to send an e-mail to the author.

In this report we describe one specific aspect of `Helium`: its logging facility. The idea of having a logging facility for `Helium` came up in a discussion between the author and Bastiaan Heeren. The main motivation for this facility was to test and experiment with the type inference process in `Helium` which is based on a number of their papers, (consult the PhD thesis by Bastiaan Heeren for an overview [1]). Up to now, the logging facility has been used during three incarnations of a first year programming course and has, in total, yielded tens of thousands of first year functional programs, each of them time-stamped and grouped according to the student performing the compilation.

The main goal of this report is to document how we have organized and implemented the logging facility. As an aside we remark on some of the ethical aspects of logging. The intended audience thus consists of people in the Helium Development Team, lecturers who intend to use the compiler in their courses and wish to know about what is logged and what is not, compiler builders who are thinking of adding a similar logging facility to their own tools, and programmers generally interested in how to implement communication between `Haskell` and `Java` programs by means of the ubiquitous sockets.

Any given compilation is classified as succesful, or has lexical, parsing, static, type or critical errors[1]. If the facility is turned on, the compiler will send his classification together with the name of the student, the version of the compiler, the source of the compiled module and the sources it imports to the server. This also occurs if the compilation happens to be succesful. There are at least three reasons for logging correct programs:

- We get quite a complete picture of a programming session, because every logging obtains a timestamp from the server. Because the name of the student is sent along as well, this implies that we have for each student a programming history available. Note that students may also program at home or decide to work on a different account, so that this history can have a number of gaps.

- Versions of the compiler differ in the quality and the number of hints they give, which makes it important to know for each logging which compiler was responsible. This is why version information is part of the logging. Because everything is logged, we can inspect the sequence of compiled programs to determine

[1] Critical errors signify an error in the compiler itself. This facility allowed us to easily track down and correct mistakes made in the implementation of `Helium`

  - whether the hint is a good one,
  - the student followed our advice,
  - whether he implemented the hint correctly, and
  - how long this took him.

- We get a good indication of the relative amount of errors people make, i.e., how does the number of parse errors compare to the number of type errors, and how this might change over time.

Together, all the loggings should give us enough information to start evaluating the quality of our error messages and indicate in which places we can still improve our messages. Currently, Peter van Keeken is doing a master thesis on analyzing the loggings. His work thus far reveals that the older versions of `Helium` did not always correctly log all information. Part of his work has been to circumvent this problem by some manual preprocessing resulting in what we consider to be three distinct sets of logged `Haskell` programs (from the years 2002/2003, 2003/2004 and 2004/2005).

Obviously, in addition to evaluating our error messages, the set of programs now collected can also be used for various other purposes including:

- *Testing code analyzers*: can we discover patterns in the code of the students, maybe suggesting to use some higher-order function in some place instead of straightforward recursion.

- *Testing code optimization schemes*: in a laboratory assignment many students solve the same problem, and they tend to do so in many different ways. Applying a code optimization scheme to each of these solutions and seeing what the result is, can be very useful in determining the robustness and generality of the scheme by seeing in which cases it works, and in which cases it does not.

- *Obtaining real life information about what kind of errors programmers make*: how does the number of parse errors change over time? What kind of errors do students continue to make?[2]

- *Investigating how people go about writing code*: do they write their comments along the way, or just before the deadline? Do they use explicit type signatures?

2 What is being sent?

Versions of `Helium` that have been made available to the public and which generally reside outside our network domain, do not log information. At the moment of writing, only the `Helium` compiler which is installed on the students network logs its results.

Every time a source program is compiled, the result of compilation is, as usual, given back to the user (sometimes via an interpreter). In addition, when the logging facility is not turned off (this can be done by means of a compiler flag), information about the compilation, the compiled source file and all the modules it imports will be sent to a server which runs within the network. The reason that we also sent all the imported modules, is to be able to easily

[2] This information might be very useful as input when a new version of the language `Haskell` is defined.

recompile the program and obtain the same error message that the programmer got. Note that we do not send along the standard imported modules such as the `Prelude`, because they are part of the distribution.

There is one exception here: if the user of the interpreter (`hint`) which comes with `Helium` enters an expression, say e, and presses enter, all the necessary sources are recompiled (in case they have changed since the last time), but also a special file `Interpreter.hs` is constructed which includes the text of e. This file like all others is sent to the `Helium` compiler. However, errors made in this expression are not logged. We simply did not feel this to be worthwhile. This is admittedly an ad-hoc solution: if somebody writes a module of the same name it will also not be logged. Note though that if such an expression leads to the recompilation of another module, then the compilation of this module will be logged. Finally, it is important to realize the following: consider a situation in which a source `A` imports a module `B` that has been modified. In this case, compiling `A` results in two loggings: first one for `B` including all the modules it imports, and then a logging for `A` which includes the source for `B`. If the compilation of `B` results in an error of some kind, then `A` will not be compiled, and therefore not logged.

Different versions of `Helium` support different kinds of logging information. Here we only describe the lastest, most extensive version which is part of the `Helium` compiler used during the functional programming course of 2004/2005 (Version 1.2.2). The number of aspects we store has grown over the years, and is likely to grow in the near future. At the moment of writing, we include the following information:

- *The name of the student:* as we stated earlier, we would like to get some idea of how the development of programs proceeds, as well as being able to verify if our hints were useful and used by the programmer. This implies that we have to know which loggings originate from the same person/group. We do postprocess all the program files to remove references to the names of students.

- *Version information for the compiler:* This information helps us cope with critical errors which are logged, because we can tell from which compiler it originates. A second reason, one to which we alluded earlier on, we need to have the compiler version to be able to reconstruct the message that the programmer got (the level of detail in a message tends to vary between versions of the compiler).

- *A logging code:* every type of error has a special code so that we can easily look for only those programs which, e.g., exhibit a parse error. At this moment the following codes exist:

  L: lexer errors like illegally escaped characters, unterminated comments and strings and unexpected characters, but also unbalanced parentheses,

  P: parse errors,

  R: operator resolving errors, for expressions for which the given priorities and associativities do not lead to an unambiguous expression structure,

  S: static errors, including a subclassification such as `unva` for undefined variable, etc.

  T: type errors,

  C: correct programs, i.e., successful compilation

- To be able to recompile programs easily, we send both the compiled source and the sources of the modules it imports. There is no need to import the `Helium` *Prelude*, so it is not sent along. We assume nobody touches that specific module. The compiler version implicitly tells us which version of the Prelude we should use (although not completely).

3 Ethics and other complications

As described in the previous section, loggings also include a lot of individual information such as the login name of the user (although this name can be manually changed by an individual by changing the environment on his computer), the time of compilation, the name of the program file in case it is sent along, and also the contents of the program files. In many cases, the name of student occurs in these program files, either in comments or in a literal string.

When the programs are analyzed we are not interested in the name of this person, only in the fact that a set of programs originates from the same programmer(s). What we currently do is remove all comments from the program, replacing them with empty comments (we want to keep the layout of the code and so on exactly the same). Additionally, we remove all information inside literal strings. Although we have no experimental results to support our supposition, it is quite likely that only a few cases are thus left unnoticed, e.g., a programmer uses his name in the name of an identifier, or even as the name of a source file. Note that be erasing the contents of literal strings and the like, we do influence the semantics of the program (i.e. we might perform a pattern match on such a string to control the flow of the program). This implies that the loggings can only be used for testing run-time aspects of the program with some reservations. However, the author still has the original loggings, so if a use of this kind ever comes up, then all is not lost.

The use of student name as identification also introduces a number of problems: It is easy for students to fake the name (we use the special environment variable `USERNAME`, which is set to the account name of the student in our student network). Students usually program in pairs, and therefore they might work on the same assignment under different names (this complicates tracing the development of a complete program). Also, recall that students working outside the student network are not logged at all (the `Helium` executables available for download do not have a logging facility, although admittedly with a bit of effort a student can compile a version of the compiler that does). With the amount of information we gather, such incidents are not likely to influence our findings, which does not mean we should forget all about it.

4 Within the compiler

The part within the `Helium` compiler is quite straightforward. We have made the actual code a little less verbose by shortening some of the variable names.

```
module Logger ( logger ) where

import Network
import Control.Concurrent
import Monad
```

```
import System
import List
import IO
import Version
```

The logger is parameterized by the following declarations. For concision, we have omitted the type declarations.

```
HOSTNAME = "favouritehost.cs.uu.nl"
PORTNUMBER = 5010   -- choose a safe number
DELAY = 10000       -- in micro-seconds
TRIES = 2           -- higher results in long delays for the programmer

SEPARATOR = "\NUL\NUL\n"  -- marker between different files
TERMINATOR = "\SOH\SOH\n" -- end of message
USERNAME = "USERNAME"     -- to retrieve user name from the environment
ENABLED = True            -- turning the logger off and on


logger :: String -> Maybe ([String],String) -> Bool -> IO ()
logger logcode maybeSources DEBUGMODE
    | not ENABLED || isInterpreterModule maybeSources = return ()
    | otherwise      = do
        username <- (getEnv USERNAME) `catch` (\_ -> return "unknown")
        sources  <- case maybeSources of
            Nothing ->
                return (TERMINATOR)
            Just (imports,hsFile) ->
               let f name = do debug ("Logging file " ++ name) DEBUGMODE
                               program <- readFile name
                               return (  fileNameWithoutPath name
                                      ++ "\n"
                                      ++ program
                                      ++ "\n"
                                      ++ SEPARATOR
                                      )
               in (do
                    xs <- mapM f imports
                    x  <- f hsFile
                    return (concat (SEPARATOR:x:xs)++TERMINATOR)
                   ) `catch` (\_ -> return (TERMINATOR) )
        sendLogString (username++":"++logcode++":"++version++"\n"++sources)
                      DEBUGMODE
```

Figure 1: The logger function

The function `logger` is responsible for starting the logging process (see Figure 1). It is called with the logging code ("T" for type error etc.) and possibly a list of names of source

files which are to be sent along. This includes the name of the file which was being compiled and a list of modules which are imported by that module. As mentioned earlier, if a file called `Interpreter.hs` is being compiled we do not log any information. Otherwise every expression typed in the interpreter, like `4 + 5`, followed by an enter would result in a logging. Note, though, that compilations of files that are reloaded and recompiled as a consequence of evaluating `Interpreter.hs` will in fact be logged. The decision criterion can be found in the function `isInterpreterModule`.

The value of `DEBUGMODE` determines whether diagnostics are displayed on screen. Sometimes we wish to avoid this, for instance, because it would hamper the working of the interpreter. For instance, if a logging needs to be made, and the server is not up, a message of this fact should only be displayed if the user has indicated that he is interested in this type of information. Usually, this is not the case.

A logging consists of a message followed by a list of sources (possible empty). All the information, the log string, the module names and the module sources are simply compiled into one large string with various special character sequences being inserted to make the structure explicit so that the server can deserialize it again. Note that we also use the colon as a structuring character, but we do not escape it in for instance the username. This might foul up the administration although it cannot foul up the server. The separators between the administration part and the program sources can also be used to foul up a message, but we use special ASCII characters here to prevent this as much as possible. The SEPARATOR constant is used as a separator between different sources, the name of the source file and the source file itself between such SEPARATORs are only separated by a newline. The message is always terminated with a TERMINATOR string.

Summarizing, we usually send a message of the following form:

```
msg ::= username:logcode:version\nSEPARATOR(source*)TERMINATOR
source ::= filename\nfilecontent\nSEPARATOR
```

In words, we start with the basic information followed by a possible empty list of source files separated by SEPARATOR, and finally terminated by TERMINATOR. Each source is the name of the file (stripped of its path, in fact) followed by the contents.

After `logger` composes the message, the function `sendLogString` is called to handle the transmission part. It uses the function `sendToAndFlush` to make sure the message arrives and, if necessary, tries again when an exception is thrown, e.g., the server is not up or busy, or a file transfer goes wrong. The number of retries can be set to any value, but for performance reasons it should not be too high (one or two should do). Between retries there is a small sleep period to, e.g., give the server some time to recover. The value is somewhat arbitrary and can be tuned for good behaviour.

After implementing the socket communication in `Haskell` it turned out that sending a message and immediately afterwards closing the socket did not work: the message was lost (Note that this is not the case for the `Java Socket` class for instance). This is why the `Helium` compiler waits for a message back from the `Java` program that handles this message (it does so by `handshake <- getRetriedLine 0`). The recursive definition `rec` makes sure that the logger tries to connect a number of times, and to sleep in between tries (call to the `threadDelay` function). Each failed try results in an exception being raised, so we catch it and try again, until we have used up the maximum number of tries. A similar retry structure is used for the receipt of the handshake code in `sendToAndFlush`.

```
sendLogString :: String -> Bool -> IO ()
sendLogString message loggerDEBUGMODE = withSocketsDo (rec 0)
 where
    rec i = do
              handle <- connectTo loggerHOSTNAME
                          (PortNumber (fromIntegral loggerPORTNUMBER))
              hSetBuffering handle (BlockBuffering (Just 1024))
              sendToAndFlush handle message loggerDEBUGMODE
            `catch`
                \exception ->
                   if i+1 >= loggerTRIES
                     then debug ( "Could not make a connection: no send ("
                                  ++ show exception ++ ")" ) DEBUGMODE
                     else do debug ( "Could not make a connection: sleeping ("
                                     ++ show exception ++ ")" ) DEBUGMODE
                             threadDelay loggerDELAY
                             rec (i+1)

sendToAndFlush :: Handle        -- Hostname
               -> String        -- Message to send
               -> Bool          -- Debug logger?
               -> IO ()
sendToAndFlush handle msg loggerDEBUGMODE = do
  hPutStr handle msg
  hFlush handle
  debug "Waiting for a handshake"  loggerDEBUGMODE
  handshake <- getRetriedLine 0
  debug ("Received a handshake: " ++ show handshake) loggerDEBUGMODE
  where
    getRetriedLine i =
      do
        line <- hGetLine handle
        return line
      `catch`
        \_ ->
          if i+1 >= loggerTRIES
            then do
                   debug "Did not receive anything back" loggerDEBUGMODE
                   return ""
            else do
                   debug "Waiting to try again" loggerDEBUGMODE
                   threadDelay loggerDELAY
                   getRetriedLine (i+1)
```

5 The server side

The server side should follow exactly the same protocol as the sender. Unfortunately, they are programmed in different languages, which makes it a bit more difficult to maintain this invariant. Additionally, the server takes care when reading the message to give as precise errors as possible when a problem is detected. This also complicates the code for the server somewhat, because all kinds of exceptions can be raised and handled.

I do not care to go into much detail on the `Java` program, but some architectural remarks may enlighten somewhat: `HeliumServer` contains the `main` method. It checks that a server is not yet running, and launches a `Server` object if this is the case. The `Server` listens on the socket (the number of which and other global settings for the program can be found in `Globals.Java`), and when a new connection is made, it launches a `HeliumThread` object which handles that connection. Then it simply continues listening on the port. This way of doing things avoids jams when concurrently a few large loggings are made, and also protects the system from flooding attempts.

The `HeliumThread` class is the core of the server: this part is most subject to change since it embodies the protocol of the message. It first reads in what is called the `adminline`. At the time of writing, that line contains the name of the student, the type of logging and the version of the compiler (it is likely more information will be added here such as the parameters passed to the compiler). To signify termination a special `adminline` is used to denote this fact (see `Globals.Java`). If such a request for termination is made, this is signalled to the `Server` by means of a Terminator object.

The admin line must be followed by at least one other line: either a message `TERMINATOR` indicating that no files are to follow, or a `SEPARATOR` indicating the opposite. If the former is the case, the admin line is sent to the `SyncBuffer` to be sent to the logfile (which is a shared protected resource). Obviously, the `TERMINATOR` and `SEPARATOR` strings should be the same as their counterparts used on the `Helium` side.

Usually, a number of files, separated by `SEPARATOR` and terminated by a `TERMINATOR`, will follow the adminline. Each of those consists of a filename followed by the contents of the file. When we know these files to be present, we must set up the directory structure to receive the files in. For that purpose a `DirMaker` object sets up a directory to receive the files in: this directory is of the form username/timestamp. The timestamp is computed using a methode `timeString` (see `Globals.Java`), based on the current time (in some versions the month was a zero indexed number, so beware drawing conclusions about the actual logging date). In this way all loggings for the same user are grouped together, and they occur in chronological order. If the user logs for the first time, this automatically creates a directory for the user. Finally, we can copy all the files in the message into a directory for this specific logging. This is handled by the method `copyFilesInto`. We assume that the first file is in fact the source file that was compiled. This is why its name is added to the logged message signifying that when we recompile later on, we should use this file.

The `SendToServer` class is a separate program which allows the programmer to test the logger without using `Helium` itself. The `TestMe` script that is provided calls the program associated with this class as a form of self test. Other scripts that are available are `Startserver` (as in `Startserver logfilename`), `TerminateServer` (sends a special termination message to the server), `Backup` (which collects all the logged entries into a zip file). In our case, it assumes that all files and directories consisting of lower case letters are involved (this is also why all the scripts and other code start with capital letters). To be able to tell that the server

is already running, there is a simple `Checkrunning` script. It only checks that a certain lock file is present in a certain directory. The entire program and associated scripts are combined into an `Ant` project and compiled into two jar files, to which the self test can then be applied.

Because we do not expect large programs, and we would not like our server to be sent huge pieces of data, a special buffer is used in the `Java` program, that, for each message, keeps track of how many bytes were sent. When the number of bytes rises over that a value (for a single given message), then the socket connection is terminated. This is conveyed to the program by means of a user-defined exception called `OverflowException` which is thrown by the `OFBufferedReader` class. The overflow size can be changed in `Globals.Java`.

The server program can be debugged via the `Debug` class.

A small problem I encountered when using sockets and readers and writers constructed on top of sockets in `Java` was the following: since the communication is not two way, but two one way communications in sequence, I thought it was cleaner to open a socket, open a reader, read, close the reader, open a writer, write and finally close the writer and the socket. For some reason, however, the implementors of `Java` decided to close a socket whenever a reader or writer for that socket is closed. These days, they even say so explicitly in the documentation, and consider it a feature. I guess it comes from how readers and writers may be used on files (or maybe because they are afraid people forget to close the socket, who knows), but for bi-directional sockets which are two independent one-way connection, it does not make sense (at least not to me). However, we are stuck with it, so I only close the reader after finishing reading and writing altogether.

6 Preparing the loggings for analysis

Although this is not the place for processing the log files, in the sense of analyzing them, some words about how we prepare the log files for analysis at a later point are in order. The loggings of the courses mentioned in this report are all cleaned up. A `Perl` script is used to anonimize the loggings which are grouped under the `USERNAME` of the student who compiled it. After the course has run, we replace all the usernames with a unique number (i.e., unique within this course). This amounts to permuting the names randomly and replacing them with 'groupX' where X is the rank of the name in the permutation. Of course, the logfile must be updated to reflect this change. This operation is performed by a straightforward `Perl` script.

As we indicated in the section of ethics earlier on, we want to remove as much information about the individual programmer as possible. Because students must mention their name and student number and the like in comments in the code, we have chosen to remove all nested and single line comments from the program, as well as all text within a literal string. Of course, the name of a student can still pop up in the code, e.g., in the guise of an identifier, but chances are small and the collection of loggings is huge. Recall that we do not actually remove comments: only the contents are replaced by spaces. Thus, the identation of the program does not change: the golden rule of cleaning up before analysis is, to keep the changes to a minimum.

We shortly summarize for each year what we obtained by our logging activities. At this point all these loggings have been cleaned of comments and literal strings and garbage loggings have been removed. In the case of the first two courses (2002/2003 and 2003/2004) the module imports have been reconstructed by Peter van Keeken).

Course year 2002/2003

We had 119 groups (sequential logging histories) (groups consist of up to two students, some of these groups might actually be two singleton groups who worked together, but from different accounts.), 29433 loggings over the period March 12th 2003 up to and including April 25th 2003. Unfortunately, we do not have the original (unanonimized) loggings for this year, so it would take some effort to determine which groups might 'belong together'.

Course year 2003/2004

142 groups, 23112 loggings over the period February 3rd 2004 up to and including March 26th 2004.

Course year 2004/2005

82 groups, 11256 loggings over the period February 7th 2005 up to and including April 8th 2005.

7 Lessons and experiences

In this section, we shortly summarize our experiences in setting up this logger. First of all, sending a message takes up quite a bit of serializing and deserializing, in other words: packing the message into a single string. Of course, we could have decided to send multiple messages, and then telling beforehand how many of these ought to arrive. The advantage is that no special code is needed, such as `SEPARATOR`, to distinguish the various parts (which is "dangerous", especially if people try to send binary files as well). A drawback is maybe that the decoupling of building the message, and the actual communication is not so strong, and it is more likely that message can be 'half sent'.

A bigger problem is the fact that the packaging code on the `Haskell` side and the unpacking on the `Java` side seem quite unrelated when you look at the code. Thus it is quite a job to keep these matched. We tried to get around this by building a `Java` sender as well, to test the receiver part without involving `Helium` and `Haskell`. A small script, which is in fact a part of the distribution, exists to test the workings of the server program with this sender program. 1Of course, then it is still a problem of keeping the match between the sender in `Haskell` and the sender in `Java`. The situation is even more complex, because the unpacking code in `Java` includes quite a few checks, and also does many side effected things such as time stamping, saving files, making new directories and so on.

Also, it is difficult to come up with a protocol that allows us to reestablish exactly the same environment as the programmer had (assuming for now he did not do his best to tweak things like faking the compiler version; if we cannot make that assumption we are lost). One thing we omitted was to send the parameters which are passed to the compiler along with the logging information. Another omission is whether or not, and which, `.type` files are used. At this point, these are not serious problems, but it can very well become important later on, so it is likely what we add this information at a later stage.

Because we ran at some point into the problem that closing a socket on the `Haskell` side, effectively killed the message, we opted for a handshake protocol instead. The problem is that the handshake is not a logical necessity so that if a version of the library is released that

does not have this problem, we would like to use that instead. But keeping track of this kind of information is not likely to be worth that.

It is important to test the system in the target environment, also to see if the server side can handle the estimated number of students. We have done this experiment and it turns out the server can easily keep pace with four computers continually sending large files. Flooding cannot be used to tie up the computer and crash the server, although a downgrade in service is possible.

The server is not so well-defended at this point: we make sure that incoming messages are not large, but we do not keep track of the number of loggings that come in. Thus it may be possible to flood the server (we have thus far not succeeded in doing so, but it is likely to be possble). Fortunately, the information is written on an account protected by a quotum, so we cannot flood the complete disc space.

8 Future work

It is becoming more and more important with the newer versions of `Helium` to know which options were passed to the compiler, especially where it concerns the choice between the overloaded and non-overloaded side of the compiler. It strongly affects what kind of message one gets (especially if the student used the overloaded library, but not the overloaded flag). To make this process easier, it would be nice to have the complete compiler invocation available in the log file.

Especially, when type directive files become part of the everyday programming experience [2], it will also become necessary to log these as well (and then we also need the compiler flags to see whether they were used or not).

Besides these obvious improvements, other extensions of functionality and use are:

- logging students at other institutes that use `Helium`,
- extending the logging facility to students not part of the network
- use the logging facility to give a student feedback about how his fellow students are doing. Thus, the handshake may convey useful information to the student, for instance by telling him that the average length of his functions is much higher than that of other students. This implies that analysis of the logged files can take more or less place on-the-fly. This simple idea opens up a host of possibilities that we shall certainly look into in the future.

Finally, if anyone wants to look at the server code (the `Helium` side of the code is part of the `Helium` source distribution), then he can e-mail me at `jur@cs.uu.nl`. The same address can be used if a lecturer at some institute wants to log the programs of this students.

Acknowledgements

I am grateful to Bastiaan Heeren for his help with the logger (both in constructing and using it), and Peter van Keeken for his clean-up work and preparation of the loggings for actual use. The author additionally thanks Bastiaan Heeren for comments on a draft version of this report.

References

[1] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/phdthesis`.

[2] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[3] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.

[4] A. van IJzendoorn, D. Leijen, and B. Heeren. The Helium compiler. `http://www.cs.uu.nl/helium`.