The Hugs 98 User's Guide

Table of Contents

Next
The Hugs 98 License

Chapter 1. Introduction

In September 1991, Mark Jones released a functional programming system called *Gofer*, which provided a compact, portable implementation of a Haskell-like language. The system also included experimental type system extensions, many of which later became part of Haskell. On Valentine's Day 1995, Mark released *Hugs* (Haskell User's Gofer System), a derivative of Gofer with greater Haskell compliance. Hugs versions are named after the version of Haskell they support; Hugs 98 was released in January 1999. Mark gave up the maintainership of Hugs in January 2000.

Hugs 98 still aims to be a fairly lightweight, portable implementation, and now adheres closely to Haskell 98. It also supports several extensions shared with other Haskell implementations:

- Hugs supports standardized extensions (addenda) to Haskell 98, for interfacing to foreign languages and structuring the module space.
- With the appropriate options (see Section 3.1.1), it is also possible to turn on a number of language extensions, most of which are also supported by the Glasgow Haskell Compiler (GHC), though some are specific to Hugs.
- Hugs comes with a large collection of libraries, also shared with other Haskell implementations, and described in separate documentation.

Though these features make Hugs highly compatible with other implementations, it is primarily intended as interpreter and programming environment for developing Haskell programs. If your application involves large programs or speed is critical, you may strike Hugs's limitations, and may wish to try a Haskell compiler.

1.1. Other sources of information

1.1.1. Other documentation

The Hugs 98 User Manual

> This was the definitive reference for earlier versions of Hugs, though parts of it are now out-of-date. Much of it remains relevant, particularly Section 7 on Hugs extensions, and it should be consulted in several areas that this Guide does not cover well. The manual is available in several formats: HTML, PDF, gzipped Postscript, gzipped tar-ed html, dvi, WinHelp(zipped) and HTMLHelp(win32 help format).

Haskell 98 Language and Libraries: the Revised Report

> The definitive reference for the Haskell 98 language and standard libraries, published by Cambridge University Press, and also available online.

Haskell Core Libraries

> A collection of libraries shared by Haskell implementations, including Hugs.

comp.lang.functional FAQ

> General information about functional programming.

More information about Haskell may be found on the Haskell home page http://www.haskell.org/ and the Hugs home page http://www.haskell.org/hugs/.

1.1.2. Mailing lists

There are a number of mailing lists where people discuss Hugs and Haskell, all with archives of past postings:

hugs-users:

> This is the place for general discussion about using Hugs.

hugs-bugs:

> Use this list for reporting bugs. This is more likely to be effective than direct mail to the authors or maintainers of Hugs. We do read this mailing list – but so do many other people, who might be able to give you more appropriate or timely advice than us! Before reporting a bug, check the list of known deviations from Haskell 98 (see Section 4.1).

cvs-hugs:

> Discussion of the development of Hugs takes place on this list. This list also receives commit messages from the Hugs CVS repository.

haskell-cafe:

> An informal list for chatting about Haskell. This is an ideal place for beginners to get help with Haskell, but Hugs-specific questions would be better directed at the Hugs lists.

haskell:

> A lower-volume list for more technical discussion of Haskell. Please do not post beginner questions or Hugs-specific questions to this list.

There are several other Haskell-related mailing lists served by *www.haskell.org*. See *http://www.haskell.org/mailman/listinfo/* for the full list.

Some Haskell-related discussion also takes place in the Usenet newsgroup *comp.lang.functional*.

Chapter 2. Using Hugs

The interpreter may be started with a command line of the form

hugs [*option...*] [*file...*]

On many systems it can also be found in the system menus, and may be started by (double) clicking on a file with a "*.hs*" or "*.lhs*" extension.

Hugs takes options from the command line and elsewhere (see Section 3.1), and then loads the Haskell *Prelude* module, as well as as any modules specified on the command line.

Hugs starts with a banner like

```
__   __ __  __  ____   ___      ________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: November 2003  ________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Prelude>
```

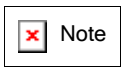
The prompt string *Prelude>* indicates that the current module is the Haskell *Prelude* (assuming no modules were specified on the command line). At this prompt, you can type Haskell expressions to be evaluated, and also enter commands of the form "***:cmd***", where *cmd* may be abbreviated to a single letter.

2.1. Basic operation

expr

> Evaluate a Haskell expression. The expression cannot be broken over multiple lines. Usually, the value is simply converted to a string (using *show*) and printed:
>
> ```
> Prelude> 1+2
> 3
> ```
>
> The printing style can be changed with the *-u* option (see Section 3.1.4).
>
> However, if *expr* has type *IO t* for some type *t*, the resulting IO action is performed:
>
> ```
> Prelude> print (1+2) >> putStrLn "bye"
> 3
> bye
> ```
>
> Usually the value produced by this action is ignored, but this can be changed with the *+I*

option (see Section 3.1.4).



On ambiguous types

If the type of *expr* is ambiguous, defaulting is applied to each ambiguous type variable *v* whose constraints all have the form *C v* where *C* is a standard class, and at least one of these classes is a numeric class, or is *Show*, *Eq* or *Ord*. (This is an extension of the Haskell 98 rule applied to top-level definitions in modules, which requires a numeric class.) It is an error if any ambiguous type variables cannot be handled in this way. For example, consider

```
Prelude> reverse []
[]
```

Here a *Show* constraint on the list elements arises from Hugs's use of *show* to display the result, so the type of the elements defaults to *Integer*, removing the ambiguity.

:type *expr*

Print the type of *expr*, without evaluating it. Usually the defaulting rules (discussed above) are not applied to the type before printing, but this can be changed with the *+T* option (see Section 3.1.4).

:set [*option...*]

Set command line options. See Section 3.1 for a list of available options. On Win32, the new option settings are saved to the registry, and so persist across Hugs sessions. To make settings persistent on other systems, put them in the *HUGSFLAGS* environment variable.

If no options are given, list the available options and their current settings.

:quit

Exit the interpreter.

Prev Introduction | Home | Next Loading and editing Haskell module files

2.2. Loading and editing Haskell module files

The Hugs prompt accepts expressions, but not Haskell definitions. These should be placed in text files containing Haskell modules, and these modules loaded into Hugs either by listing them on the command line, or by using the commands listed here. Hugs assumes that each Haskell module is in a separate file. You can load these files by name, or by specifying a module name.

Hugs maintains a notion of a *current module*, initially the *Prelude* and normally indicated by the prompt. Expressions presented to Hugs are interpreted within the scope of the current module, i.e. they may refer to unexported names within the module.

:load [*file-or-module*...]

Clear all files except the *Prelude* and modules it uses, and then load the specified files or modules (if any). The last module loaded becomes the current module.

You may specify a literal filename. The named file may contain a Haskell module with any name, but you can't load two modules with the same name together. To include a literal space in a filename, either precede it with a backslash or wrap the whole filename double quotes. Double quoted filenames may also contain the escape sequences "\ ", "\'" and "\\". Other bachslashes are interpreted literally.

When asked to load a module *M*, Hugs looks for a file *dir/M.hs* or *dir/M.lhs*, where *dir* is a directory in its search path. (The "/" is used on Unix systems; Windows systems use "\".) The search path may be changed using the *-P* option, while the set of suffixes tried may be changed using the *-S* option (see Section 3.1.2). The file found should contain a Haskell module called *M*.

In mapping compound module names like *A.B.C* to files, the dots are interpreted as slashes, leading to filenames *dir/A/B/C.hs* or *dir/A/B/C.lhs*.

Modules imported by Haskell modules are resolved to filenames in the same way, except that an extra directory is searched first when

- the importing module was loaded by specifying a filename in that directory, or
- the importing module was found relative to that directory.

This fits nicely with the scenario where you load a module

```
Prelude> :load /path/to/my/project/code.hs
Main>
```

where the directory */path/to/my/project* contains other modules used directly or indirectly by the module *Main* in *code.hs*. For example, suppose *Main* imports *A.B.C*, which in turn imports *D*. These may be resolved to filenames */path/to/my/project/A/B/C.hs*, and (assuming that is found), */path/to/my/project/D.hs*. However imports from modules found on the search path do not use the extra directory.

:also [*file-or-module*...]

Read the specified additional files or modules. The last module loaded becomes the current module.

:reload

Clear all files except the *Prelude* and modules it uses, and then reload all the previously loaded modules.

:module *module*

Set the current module for evaluating expressions.

:edit [*file*]

The *:edit* command starts an editor program to modify or view a Haskell module. Hugs suspends until the editor terminates, and then reloads the currently loaded modules. The *-E* option (see Section 3.1.3) can be used to configure Hugs to your editor of choice.

If no filename is specified, Hugs edits the current module.

:find *name*

Edit the module containing the definition of *name*.

Prev Using Hugs
Home Up
Next Getting information

2.3. Getting information

:?

Display a brief summary of the commands listed here.

:names [*pattern...*]

List names that match any of the listed patterns and are defined in any of the currently loaded modules. Patterns resemble filename patterns: * matches any substring, *?* matches any character, *[chars]* matches any of *chars*, with *char-char* standing for a range of characters, and *\char* matches *char*.

If no patterns are given, all names defined in any of the currently loaded modules are listed.

:info [*name*]

Describe the named objects. Qualified names may be used for objects defined or imported in any loaded module other than the current one.

If no arguments are given, the names of all currently loaded files are printed.

:browse [*all*] [*module...*]

List names exported by the specified modules (which must already be loaded). If "*all*" is given, list all names in scope inside the modules. If no modules are given, it describes the current module.

:version

Print the version of the Hugs interpreter. Major releases of Hugs are identified by a month and year:

```
Prelude> :version
-- Hugs Version November 2003
```

Development snapshots are identified by a date in *YYYYMMDD* form.

2.4. Miscellaneous commands

:! [*command*]

Shell escape. If the *command* is omitted, run a shell.

:cd *dir*

Change the working directory of Hugs to *dir*. If *dir* begins with "~/", the "~" is replaced by your home directory.

:gc

Force a garbage collection.

Chapter 3. Changing the behaviour of Hugs

3.1. Hugs options

The behaviour of Hugs may be modified by options. These are initially set to default values, and then read from the following sources in order:

1. (Windows only) the registry under the *HKEY_LOCAL_MACHINE* key.
2. (Windows only) the registry under the *HKEY_CURRENT_USER* key. This step is skipped if the environment variable *IGNORE_USER_REGISTRY* is set, providing an emergency workaround if the settings are invalid.
3. (Hugs for Windows only) the GUI settings.
4. (Mac OS prior to Mac OS X) the preferences file "`Hugs Preferences`".
5. The environment variable *HUGSFLAGS*, if set.
6. The Hugs command line.

Most options can be changed within Hugs using the **:set** command (see Section 2.1).

Hugs takes two kinds of options:

- Toggles like *+t* or *-t*, which start with + or - to turn them on or off, respectively.
- Options that set a parameter value, like *-Pstr*, in which - could be replaced by +, the choice making no difference.

3.1.1. Language options

+98

Accept only Haskell 98. This is on by default, and cannot be changed within Hugs. Turning this off enables several special Hugs extensions, which are described in Chapter 5, Chapter 6 and the *Hugs 98 User Manual*.

-cnum

Set constraint cutoff limit in the type checker to *num* (default: 40). See Section 5.2.2.

+o

Allow certain overlapping instances (a Hugs extension; default: off). See Section 5.2.3 for details.

+O

Allow certain overlapping instances (a Hugs extension; default: off). These are the same overlapping instances as accepted by *+o*, but *+O* also accepts ambiguous uses of these instances, even though this is unsafe (see Section 5.2.3).

+H

Support *here documents* (named after similar things in Unix shells), another way of writing large string literals (see Section 6.3). This extension is turned off by default.

3.1.2. Module loading options

+l

Literate scripts as default (default: off).

Files with names ending in "*.hs*" are always treated as ordinary Haskell, while those ending in "*.lhs*" are always treated as literate scripts. This option determines whether other files are literate scripts or not. (See Section 3.3 for an example.)

+.

Print dots to show progress while loading modules (default: off).

+q

Print nothing to show progress while loading modules (default: on).

+w

Always show which files were loaded (default: off).

-Fcmd

Preprocess source files before loading. Instead of reading a source file directly, Hugs will execute *cmd* with the source file name as argument, and read the standard output.

This is handy for preprocessing source files with the C preprocessor, or some preprocessor implementing a language extension. However it is slower. In particular (because of the way Hugs handles imports), the preprocessor will be run twice on files that import modules that have not been loaded yet.

-Pstr

Set search path for source files to *str*, which should be a list of directories separated by colons (semicolons on Windows, DOS or Macs). A null entry in this list will be replaced by the previous search path; a null *str* means the default path. Any occurrences of *{Hugs}* in this string will be replaced by the Hugs library directory. Similarly, *{Home}* is expanded to your home directory. An entry of the form "*directory/**" means all the immediate subdirectories of *directory*. The default value is

```
.:{Hugs}/libraries:{Hugs}/oldlib
```

The interpreter won't let you change the search path if that would prevent it from reading the *Prelude*.

-Sstr

Set list of filename suffixes.

Normally, when you import a module *M*, Hugs looks for files *M.hs* and *M.lhs* in each directory in you search path. With this option, you can change this list, in a similar way to the *-P* option for the search path. By default, the suffix list is "*.hs:.lhs*", which gives the behaviour just described. (NB: the "*:*" is the Unix separator. Windows or Macs use "*;*" instead.) If you use *-S:.xhs* then the suffix list becomes "*.hs:.lhs:.xhs*", so Hugs will look for *M.hs*, *M.lhs* and *M.xhs*.

A null entry in this list will be replaced by the previous suffix list; a null *str* means the default list.

The interpreter won't let you change the suffix list if that would prevent it from reading the *Prelude*, i.e. you must include "*.hs*". Note also that the interpreter knows that files ending in "*.lhs*" are literate scripts; no other suffix is treated that way.

This option can be useful in conjunction with the preprocessor option (*-F*). The preprocessor can examine the filename to decide what to do with the file.

3.1.3. Specifying a source file editor

-Estr

Specify the editor used by the *:edit* command (see Section 2.2). For example, to have Hugs invoke *vi* to edit your files, use

```
-Evi
```

The argument string is actually a template string that gets expanded by Hugs, via the following rules:

- all occurrences of *%d* are replaced by the line number of where the last error occurred (if any). Please consult your editor's documentation for ways of specifying the line number.
- all occurrences of *%s* are replaced by the name of the file. If an occurrence of *%s* is both preceded by and followed by space, the filename is enclosed in double-quotes.
- all occurrences of *%f* are replaced by the absolute filename (provided your platform lets you find the absolute path to a file.) Most of the time, *%s* will be just fine, but in case your editor doesn't handle relative filenames correctly, try using *%f*.
- all occurrences of %% are replaced by %.
- (win32 only): if the *-E* string is prefixed with the character "&", then the invocation is asynchronous, that is, the editor process is created, but Hugs won't wait for the editor to terminate.

- (win32 only): if the *-E* string is prefixed with the character "*!*", then the invocation will be asynchronous and use the underlying command processor/shell to execute the command.

If neither *%s* nor *%f* occurs within the *-E* string, then the filename is appended before invoking the editor.

Here are some example editor configurations:

- TextPad

```
-E"c:/Program Files/TextPad 4/textpad \"%s\"(%d)"
```

- vi and clones

```
-E"vi +%d %s"
```

- gnuclient (for use with (X)Emacs)

```
-E"gnuclient +%d %s"
```

3.1.4. Evaluation and printing options

-pstr

Set prompt string to *str* (default: "*%s*> "). Any *%s* in *str* will be replaced by the current module name.

-rstr

Set the string denoting the last expression to *str* (default: "*$$*").

+k

Show kind errors in full (default: off).

In Haskell, each type expression has a *kind*. These kinds do not appear in the source language, but they are checked for consistency. By default, Hugs reports such errors as an *Illegal type*. For example, the declaration

```
instance Monad Int
```

gives rise to the error

```
ERROR "Test.hs":4 - Illegal type in class constraint
```

However if *+k* is given, the error message is identified as a *Kind error*, and is expanded to include the conflicting kinds:

```
ERROR "Test.hs":4 - Kind error in class constraint
*** constructor    : Int
*** kind           : *
*** does not match : * -> *
```

Also, when *+k* is given, the output of the **:info** will include kind information for classes and type constructors:

```
Prelude> :info Monad
-- constructor class with arity * -> *
...
Prelude> :info Int
-- type constructor with kind *
```

+T

Apply defaulting rules to types before printing (default: off).

When printing out types, the interpreter will normally not try to simplify types by applying defaulting rules, e.g.,

```
Prelude> :t 1
1 :: Num a => a
Prelude>
```

With the *+T* option, the interpreter attempts to "default" types first, using the same rules as for expressions (see Section 2.1):

```
Prelude> :set +T
Prelude> :t 1
1 :: Integer
Prelude>
```

+Q

Qualify names when printing (default: off).

By default, the interpreter will print out names without qualifying them with their defining modules. Most of the time that's exactly what you want, but can become confusing if you re-define types and functions; the error messages not pinning down what entity it is referring to. To have the interpreter qualify the names, use *+Q*. Typically, you use *+Q* when resolving errors, but turn it back off again afterwards.

+t

Print the type of each expression evaluated (default: off).

Normally Hugs merely prints the value of each expression evaluated:

```
Prelude> 1+2
3
```

With the *+t* option, it also adds the type of the expression:

```
Prelude> :set +t
Prelude> 1+2
3 :: Integer
```

Note that defaulting has been applied to the type of the expression in order to evaluate it, so the type differs from that reported by the **:type** command (assuming that the *+T* option is not used):

```
Prelude> :type 1+2
1 + 2 :: Num a => a
```

+u

Use *show* to display results (default: on).

By default, the values of expressions typed at the prompt are printed using the *show* member of the *Show* class:

```
Prelude> [Just (2+3), Nothing]
[Just 5,Nothing]
```

You can define this function as desired for any new datatype. If the type of the expression is not an instance of the *Show* class, an error results:

```
Prelude> id
ERROR - Cannot find "show" function for:
*** Expression : id
*** Of type    : a -> a
```

With the *-u* option, a built-in printer is used instead, and this works for any type:

```
Prelude> :set -u
Prelude> id
id
Prelude> \x -> x
v1497
Prelude> [Just (2+3), Nothing]
[Maybe_Just 5,Maybe_Nothing]
```

Another feature of the built-in printer is its treatment of failures (or exceptions). Normally, an exception causes immediate failure of the expression:

```
Prelude> :set +u
Prelude> 1 + 2/0

Program error: divide by zero

Prelude> [1, 2 + error "foo", 3]
[1,
Program error: foo
```

However the built-in printer prints the whole value, with embedded exceptions:

```
Prelude> :set -u
Prelude> [1, 2 + error "foo", 3]
[1,{error "foo"},3]
```

Sometimes a component could produce one of two or more exceptions, but the built-in printer shows only one of them:

```
Prelude> 1 + error "foo" + error "bar"
{error "foo"}
```

+I

Display results of IO programs (default: off).

By default, an expression of *IO* type typed at the prompt is executed for effect, but the final value it produces is discarded. When *+I* is used, such an expression is evaluated, and then its result is printed with *Prelude.print*:

```
Prelude> :set +I
Prelude> (return 'a' :: IO Char)
'a'
Prelude>
```

i.e., evaluating an IO action *m* with *+I* in effect is equivalent to evaluating "*do { x <- m ; print x }*" with *-I*.

3.1.5. Resource usage options

-hnum

Set the maximum size in the Hugs heap (default: *250k*). The argument should be a decimal number, and may by suffixed with "*k*" (thousands), "*M*" (millions) or "*G*" (billions, if your machine has that much memory). Case is not significant. The heap size is measured in cells, each of which usually comprises two *int*s (taking up 8 bytes on most common architectures).

Setting this option with **:set** does not change the heap size for the current execution of Hugs. On Win32, however, all options are saved to the registry, so it will take effect the next time Hugs is run.

+s

Print statistics after each evaluation (default: off).

For each evaluation, this option shows

- the number of reductions performed (a crude measure of the amount of work performed by the interpreter),
- the total number of cells allocated during evaluation, and
- the number of garbage collections that occurred during evaluation (if any).

Note that even the most trivial evaluation involves several reductions and cells, because Hugs wraps the expression in code to print the value and catch exceptions:

```
Prelude> True
True
(25 reductions, 46 cells)
```

Note that the cell count measures the total amount of allocation, rather than the number of cells in use at any time (the *residency*). For that, the *+g* option may be more useful. In general these statistics cannot be safely used for much more than spotting general trends.

+g

Print the number of cells recovered after each garbage collection (default: off). This can be useful for analysing the *residency* of an algorithm, the amount of memory is is actually using at each point in time. For example,

```
Prelude> :set +g
Prelude> length [1..60000]
{{Gc:237618}}{{Gc:237617}}{{Gc:237616}}{{Gc:237623}}{{Gc:237621}}
{{Gc:237628}}{{Gc:237623}}{{Gc:237618}}60000
```

We see that the computation creates a lot of cells, but the number recovered on each garbage collection is roughly the same, so its residency is constant. In contrast, with

```
Prelude> let xs = [1..60000] in sum xs `div` length xs
{{Gc:237510}}{{Gc:213862}}{{Gc:190948}}{{Gc:170500}}{{Gc:152225}}
{{Gc:135925}}{{Gc:121350}}{{Gc:108350}}{{Gc:96750}}{{Gc:86375}}
{{Gc:77125}}{{Gc:68860}}{{Gc:61490}}{{Gc:72948}}{{Gc:97265}}{{Gc:129688}}
{{Gc:172916}}{{Gc:230551}}30000
```

we see that the amount reclaimed by each garbage collection is steadily falling until a certain point (because the original list is retained). These examples use the default heap size of 250000 cells; this may be changed with the *-h* option.

Since these garbage collection messages will be unpredictably interleaved with the desired output, you would usually only turn *+g* on to analyse memory problems, and then turn it off afterwards.

+R

Enable root optimisation (default: on).

This usually gives a small gain in speed, but you might want to turn it off if you're using the observation-based debugger (see Section 6.4.2.3).

3.2. Environment variables used by Hugs

Hugs also consults a number of environment variables on systems that support them. The method for setting these varies with the system.

EMACS

(Windows only) If this variable is set, Hugs is assumed to be running in an Emacs subshell (with different line termination conventions on input).

HOME

The user's home directory. This is substituted for *{Home}* in the argument of the *-P* option (see Section 3.1.2), and for "~" in the argument of the **:cd** command (see Section 2.4).

HUGSFLAGS

Additional options for Hugs, processed before any given on the command line (see Section 3.1). Within this string, options may be separated by whitespace. To include a literal space in an option, either precede it with a backslash or wrap the whole option in double quotes. The following example assumes a shell compatible with the Unix Bourne shell:

```
HUGSFLAGS='+k -E"vi +%d"'
export HUGSFLAGS
```

Double quoted options may also contain the escape sequences "\ ", "\'" and "\\". Other bachslashes are interpreted literally.

HUGSDIR

The Hugs library directory. This is substituted for *{Hugs}* in the argument of the *-P* option (see Section 3.1.2).

IGNORE_USER_REGISTRY

(Windows only) If this variable is set, options are not read from the user portion of the Registry (see Section 3.1). You might use this to recover if your registry settings get messed up somehow.

SHELL

The shell that is invoked by the **:!** command (see Section 2.4).

3.3. Standalone programs

runhugs [*option*...] *file*

The **runhugs** command is an interpreter for an executable Hugs script, which must be a file containing a Haskell *Main* module. For example, an executable file might contain the lines

```
#!/usr/local/bin/runhugs +l

> module Main where
> main = putStr "Hello, World\n"
```

When this file is executed, **runhugs** will invoke the *main* function. Any arguments given on the command line will be available through the *getArgs* action.

3.4. Compiling modules that use the Foreign Function Interface

ffihugs [*+G*] [*option...*] [*+Lcc-option...*] *file*

Suppose you have some C functions in *test.c* and some ffi declarations for those functions in *Test.hs* and the code in *test.c* needs to be compiled with *-lm*. To use these with Hugs, you must first use **ffihugs** to generate *Test.c*, compile it and link it against *test.c* with *-lm* to produce *Test.so*:

```
ffihugs +G +L"test.c" +L"-lm" Test.hs
```

(If *Test.hs* depends on other ffi modules, you'll have to compile them first.) Now you can run Hugs as normal; when *Test.hs* is loaded, Hugs will load *Test.so*.)

```
hugs Test.hs
```

and then use the imported or exported functions.

Chapter 4. Hugs *vs* Haskell 98 and addenda

In *+98* mode, Hugs supports Haskell 98 and some standardized extensions (described by addenda to the Haskell 98 report).

4.1. Haskell 98 non-compliance

Hugs deviates from Haskell 98 in a few minor ways, listed here corresponding to the relevant sections of the Report.

4.1.1. Lexical structure

Restricted character set

> The Haskell report specifies that programs may be written using Unicode. Hugs only accepts the ISO8859-1 (Latin-1) subset at the moment.

Floating point literals

> Hugs is confused by such things as "*0xy*", "*0oy*", "*9e+y*" and "*9.0e+y*", because it doesn't look far enough ahead.

4.1.2. Expressions

Interaction of fixities with the *let*/lambda meta-rule

> Hugs doesn't use the fixity of operators until after parsing, and so fails to accept legal (but weird) Haskell 98 expressions like
>
> ```
> let x = True in x == x == True
> ```

Restricted syntax for left sections

> In Hugs, the expression must be an fexp (or *case* or *do*). Legal expressions like *(a+b+)*) and *(a*b+)*) are rejected.

4.1.3. Declarations and bindings

Slight relaxation of polymorphic recursion

> Hugs's treatment of polymorphic recursion is less restrictive than Haskell 98 when the functions involved are mutually recursive. Consider the following example:
>
> ```
> data BalancedTree a = Zero a | Succ (BalancedTree (a,a))
>
> zig :: BalancedTree a -> a
> zig (Zero a) = a
> ```

```
zig (Succ t) = fst (zag t)

zag (Zero a) = a
zag (Succ t) = snd (zig t)
```

As with many operations on non-regular (or nested) types, *zig* and *zag* need to be polymorphic in the element type. In Haskell 98, the bindings of the two functions are interdependent, and thus constitute a single binding group. When type inference is performed on this group, *zig* may be used at different types, because it has a user-supplied polymorphic signature. However, *zag* may not, and the example is rejected, unless we add an explicit type signature for *zag*. (It could be argued that this is a bug in Haskell 98.)

In Hugs, the binding of *zig* depends on that of *zag*, but not vice versa. (The binding of *zag* is considered to depend only on the explicit signature of *zig*.) It is possible to infer a polymorphic type for *zag*, and from that for *zig*. This type matches the declared signature, so Hugs accepts this example.

Relaxation of type classes

Contrary to the the Report (4.3.1), Hugs allows the types of the member functions of a class *C a* to impose further constraints on *a*, as in

```
class Foo a where
  op :: Num a => a -> a -> a
```

Different implementation of the monomorphism restriction for top-level bindings

For example, Hugs rejects the following example from the Haskell 98 Report, 4.5.5:

```
module M where
  import List
  len1 = genericLength "Hello"
  len2 = (2*len1) :: Rational
```

This module consists of two binding groups, containing *len1* and *len2* respectively. Type inference on the first (*len1*) triggers the monomorphism restriction, so that *len1* is assigned the monomorphic type *(Num a => a)*. The next step differs between Haskell 98 and Hugs:

- In Haskell 98, type inference is then performed on *len2*, resolving the type variable *a* to *Rational*, and the module is legal.
- In Hugs, the defaulting rule is applied to *len1*, instantiating the type variable *a* to *Integer*. Then type inference on *len2* fails.

4.1.4. Modules

Implicit module header

In Haskell 98, if the module header is omitted, it defaults to "*module Main(main) where*". In Hugs it defaults to "*module Main where*", because many people test small modules without module headers.

Implicit export list

In Haskell 98, a missing export list means all names defined in the current module. In Hugs, it

is treated as "*(module M)*", where *M* is the current module. This is almost the same, differing only when an imported module is aliased as *M*.

Type synonyms in export and import lists

Hugs allows the *T(..)* syntax for type synonyms in export and import lists. It also allows the form *T()* for type synonyms in import lists.

Mutually recursive modules are not supported

Note that although the Haskell 98 specification of the *Prelude* and library modules is recursive, Hugs achieves the same effect by putting most of these definitions in a module *Hugs.Prelude* that these modules import.

Weird treatment of *(:)*

The Hugs prelude exports *(:)* as if it were an identifier, even though this is not permitted in user-defined modules. This means that Hugs incorrectly rejects the following:

```
module Foo where
import Prelude()
cs = 'a':cs
```

4.1.5. Predefined types and classes

Unicode is not supported

The type *Char* is limited to the ISO8859-1 subset of Unicode.

Rational literals lose precision

In Haskell 98, a floating point literal like *1.234e-5* stands for "*fromRational (1234 % 100000000)*". In particular, if the literal is of *Rational* type, the fraction is exact. In Hugs such literals are stored as double precision floating point numbers. If the literal is of *Rational* type, it usually denotes the same number, but some precision may be lost.

Floating point values are printed differently

Haskell 98 specifies that *show* for floating point numbers is the function *Numeric.showFloat*, but Hugs uses an internal function with slightly different semantics.

Derived *Read* instances do not work for some infix constructors.

Derived instances for large tuples are not supplied

In Haskell 98, all tuple types are instances of *Eq*, *Ord*, *Bounded*, *Read*, and *Show* if all their component types are. Hugs defines these instances only for tuple types of size 5 or less (3 or less in the small Hugs configuration).

4.2. Addenda to Haskell 98

These addenda describe extensions that have been standardized across haskell implementations.

4.2.1. Foreign Function Interface

The Haskell Foreign Function Interface, as described in the FFI addendum is implemented except for the following limitations:

- Only the *ccall* calling convention is supported. All others are flagged as errors.
- *foreign export* is not implemented.
- *foreign import wrapper* are only implemented for the x86, PowerPC and Sparc architectures and has been most thoroughly tested on Windows and Linux using gcc.

Modules containing *foreign* declarations must be compiled with **ffihugs** before use (see Section 3.4).

4.2.2. Hierarchical Namespace Extension

The Haskell Hierarchical Namespace Extension allows dots in module names, e.g. *System.IO.Error*, creating a hierarchical module namespace. Hugs has supported this since the December 2001 release. When searching for the source file corresponding to a hierarchical name, Hugs replaces the dots with slashes.

Chapter 5. Language extensions supported by Hugs and GHC

These experimental features are enabled with the *-98* option. Most are described in Section 7 of the *Hugs 98 User Manual*. Those described in this chapter are also supported by GHC with appropriate options, though in some cases the GHC versions are more general

5.1. Syntactic extensions

5.1.1. Recursive do-notation

The recursive do-notation (also known as mdo-notation) is implemented as described in: *A recursive do for Haskell*, Levent Erkök and John Launchbury, *Haskell Workshop 2002*, pages: 29–37. Pittsburgh, Pennsylvania.

The do-notation of Haskell does not allow recursive bindings, that is, the variables bound in a do-expression are visible only in the textually following code block. Compare this to a let-expression, where bound variables are visible in the entire binding group. It turns out that several applications can benefit from recursive bindings in the do-notation, and this extension provides the necessary syntactic support.

Here is a simple (yet contrived) example:

```
import Control.Monad.Fix

justOnes = mdo xs <- Just (1:xs)
               return xs
```

As you can guess *justOnes* will evaluate to *Just [1,1,1,...*

The *Control.Monad.Fix* module introduces the *MonadFix* class, defined as

```
class Monad m => MonadFix m where
    mfix :: (a -> m a) -> m a
```

The function *mfix* dictates how the required recursion operation should be performed. If recursive bindings are required for a monad, then that monad must be declared an instance of the *MonadFix* class. For details, see the above mentioned reference.

The *Control.Monad.Fix* module also defines instances of *MonadFix* for *List*, *Maybe* and *IO*. Furthermore, several other monad modules provide instances of the *MonadFix* class, including the *Control.Monad.ST* and *Control.Monad.ST.Lazy* modules for Haskell's internal state monad (strict and lazy, respectively).

There are three important points in using the recursive-do notation:

- The recursive version of the do-notation uses the keyword *mdo* (rather than *do*).

- You should "*import Control.Monad.Fix*".
- Hugs should be started with the flag *-98*.

The web page: "http://www.cse.ogi.edu/PacSoft/projects/rmb" contains up to date information on recursive monadic bindings.

Historical note: The old implementation of the mdo-notation (and most of the existing documents) used the name *MonadRec* for the class and the corresponding library.

5.1.2. Parallel list comprehensions (a.k.a. zip-comprehensions)

Parallel list comprehensions are a natural extension to list comprehensions. List comprehensions can be thought of as a nice syntax for writing maps and filters. Parallel comprehensions extend this to include the *zipWith* family.

A parallel list comprehension has multiple independent branches of qualifier lists, each separated by a "|" symbol. For example, the following zips together two lists:

```
[ (x, y) | x <- xs | y <- ys ]
```

The behavior of parallel list comprehensions follows that of *zip*, in that the resulting list will have the same length as the shortest branch.

We can define parallel list comprehensions by translation to regular comprehensions. Given a parallel comprehension of the form:

```
[ e | p1 <- e11, p2 <- e12, ...
    | q1 <- e21, q2 <- e22, ...
    ...
]
```

This will be translated to:

```
[ e | ((p1,p2), (q1,q2), ...) <- zipN [(p1,p2) | p1 <- e11, p2 <- e12, ...]
                                      [(q1,q2) | q1 <- e21, q2 <- e22, ...]
                                      ...
]
```

where "*zipN*" is the appropriate zip for the given number of branches.

5.2. Type class extensions

5.2.1. More flexible contexts

In Haskell 98, contexts consist of class constraints on type variables applied to zero or more types, as in

```
f :: (Functor f, Num (f Int)) => f String -> f Int -> f Int
```

In class and instance declarations only type variables may be constrained. With the *-98* option, any type may be constrained by a class, as in

```
g :: (C [a], D (a -> b)) => [a] -> b
```

Classes are not limited to a single argument either (see Section 5.2.4).

5.2.2. More flexible instance declarations

In Haskell 98, instances may only be declared for a *data* or *newtype* type constructor applied to type variables. With the *-98* option, any type may be made an instance:

```
instance Monoid (a -> a) where ...
instance Show (Tree Int) where ...
instance MyClass a where ...
instance C String where
```

This relaxation, together with the relaxation of contexts mentioned above, makes the checking of constraints undecidable in general (because you can now code arbitrary Prolog programs using instances). To ensure that type checking terminates, Hugs imposes a limit on the depth of constraints it will check, and type checking fails if this limit is reached. You can raise the limit with the *-c* option, but such a failure usually indicates that the type checker wasn't going to terminate for the particular constraint problem you set it.

Note that GHC implements a different solution, placing syntactic restrictions on instances to ensure termination, though you can also turn these off, in which case a depth limit like that in Hugs is used.

5.2.3. Overlapping instances

With the relaxation on the form of instances discussed in the previous section, it seems we could write

```
class C a where c :: a
instance C (Bool,a) where ...
instance C (a,Char) where ...
```

but then in the expression *c :: (Bool,Char)*, either instance could be chosen. For this reason, overlapping instances are forbidden:

```
ERROR "Test.hs":4 - Overlapping instances for class "C"
```

```
*** This instance   : C (a,Char)
*** Overlaps with   : C (Bool,a)
*** Common instance : C (Bool,Char)
```

However if the *+o* option is set, they are permitted when one of the types is a substitution instance of the other (but not equivalent to it), as in

```
class C a where toString :: a -> String
instance C [Char] where ...
instance C a => C [a] where ...
```

Now for the type *[Char]*, the first instance is used; for any type *[t]*, where *t* is a type distinct from *Char*, the second instance is used. Note that the context plays no part in the acceptability of the instances, or in the choice of which to use.

The above analysis omitted one case, where the type *t* is a type variable, as in

```
f :: C a => [a] -> String
f xs = toString xs
```

We cannot decide which instance to choose, so Hugs rejects this definition. However if the *+O* option is set, this declaration is accepted, and the more general instance is selected, even though this will be the wrong choice if *f* is later applied to a list of *Char*.

Hugs used to have a *+m* option (for multi-instance resolution, if Hugs was compiled with *MULTI_INST* set), which accepted more overlapping instances by deferring the choice between them, but it is currently broken.

Sometimes one can avoid overlapping instances. The particular example discussed above is similar to the situation described by the *Show* class in the *Prelude*. However there overlapping instances are avoided by adding the method *showList* to the class

5.2.4. Multiple parameter type classes

In Haskell 98, type classes have a single parameter; they may be thought of as sets of types. In Hugs, they may have one or more parameters, corresponding to relations between types, e.g.

```
class Isomorphic a b where
  from :: a -> b
  to :: b -> a
```

5.2.5. Functional dependencies

Multiple parameter type classes often lead to ambiguity. Functional dependencies (inspired by relational databases) provide a partial solution, and were introduced in *Type Classes with Functional Dependencies*, Mark P. Jones, In *Proceedings of the 9th European Symposium on Programming*, LNCS vol. 1782, Springer 2000.

Functional dependencies are introduced by a vertical bar:

```
class MyClass a b c | a -> b where
```

This says that the *b* parameter is determined by the *a* parameter; there cannot be two instances of *MyClass* with the same first parameter and different second parameters. The type inference system

then uses this information to resolve many ambiguities. You can have several dependencies:

```
class MyClass a b c | a -> b, a -> c where
```

This example could also be written

```
class MyClass a b c | a -> b c where
```

Similarly more than one type parameter may appear to the left of the arrow:

```
class MyClass a b c | a b -> c where
```

This says that the *c* parameter is determined by the *a* and *b* parameters together; there cannot be two instances of *MyClass* with the same first parameter and second parameters, but different third parameters.

Prev
Language extensions supported by Hugs and GHC

Home
Up

Next
Quantified types

5.3. Quantified types

5.3.1. Rank 2 types

In Haskell 98, all type signatures are implicitly universally quantified at the outer level, for example

```
id :: a -> a
```

Variables bound with a *let* or *where* may be polymorphic, as in

```
let f x = x in (f True, f 'a')
```

but function arguments may not be: Haskell 98 rejects

```
g f = (f True, f 'a')
```

However, with the *-98*, the function *g* may be given the signature

```
g :: (forall a. a -> a) -> (Bool, Char)
```

This is called a *rank 2* type, because a function argument is polymorphic, as indicated by the *forall* quantifier.

Now the function *g* may be applied to expression whose generalized type is at least as general as that declared. In this case the choice is limited: we can write

```
g id
g undefined
g (const undefined)
```

or various equivalent forms

```
g (\x -> x)
g (id . id . id)
g (id id id)
```

There are a number of restrictions on such functions:

- Functions that take polymorphic arguments must be given an explicit type signature.
- In the definition of the function, polymorphic arguments must be matched, and can only be matched by a variable or wildcard (_) pattern.
- When such functions are used, the polymorphic arguments must be supplied: you can't just use *g* on its own.

GHC, which supports arbitrary rank polymorphism, is able to relax some of these restrictions.

Hugs reports an error if a type variable in a *forall* is unused in the enclosed type.

An important application of rank 2 types is the primitive

```
runST :: (forall s. ST s a) -> a
```

in the module *Control.Monad.ST*. Here the type signature ensures that objects created by the state monad, whose types all refer to the parameter *s*, are unused outside the application of *runST*. Thus to use this module you need the *-98* option. Also, from the restrictions above, it follows that *runST* must always be applied to its polymorphic argument. Hugs does not permit either of

```
myRunST :: (forall s. ST s a) -> a
myRunST = runST

f x = runST $ do
      ...
      return y
```

(though GHC does). Instead, you can write

```
myRunST :: (forall s. ST s a) -> a
myRunST x = runST x

f x = runST (do
      ...
      return y)
```

5.3.2. Polymorphic components

Similarly, components of a constructor may be polymorphic:

```
newtype List a = MkList (forall r. r -> (a -> r -> r) -> r)
newtype NatTrans f g = MkNT (forall a. f a -> g a)
data MonadT m = MkMonad {
                  my_return :: forall a. a -> m a,
                  my_bind :: forall a b. m a -> (a -> m b) -> m b
                }
```

So that the constructors have rank 2 types:

```
MkList :: (forall r. r -> (a -> r -> r) -> r) -> List a
MkNT :: (forall a. f a -> g a) -> NatTrans f g
MkMonad :: (forall a. a -> m a) ->
           (forall a b. m a -> (a -> m b) -> m b) -> MonadT m
```

As with functions having rank 2 types, such a constructor must be supplied with any polymorphic arguments when it is used in an expression.

The record update syntax cannot be used with records containing polymorphic components.

5.3.3. Existential quantification

It is also possible to have existentially quantified constructors, somewhat confusingly also specified with *forall*, but before the constructor, as in

```
data Accum a = forall s. MkAccum s (a -> s -> s) (s -> a)
```

This type describes objects with a state of an abstract type *s*, together with functions to update and

query the state. The *forall* is somewhat motivated by the polymorphic type of the constructor *MkAccum*, which is

```
s -> (a -> s -> s) -> (s -> a) -> Accum a
```

because it must be able to operate on any state.

Some sample values of the *Accum* type are:

```
adder = MkAccum 0 (+) id
averager = MkAccum (0,0)
                   (\x (t,n) -> (t+x,n+1))
                   (uncurry (/))
```

Unfortunately, existentially quantified constructors may not contain named fields. You also can't use *deriving* with existentially quantified types.

When we match against an existentially quantified constructor, as in

```
runAccum (MkAccum s add get) [] = ??
```

we do not know the type of *s*, only that *add* and *get* take arguments of the same type as *s*. So our options are limited. One possibility is

```
runAccum (MkAccum s add get) [] = get s
```

Similarly we can also write

```
runAccum (MkAccum s add get) (x:xs) =
    runAccum (MkAccum (add x v) add get) xs
```

This particular application of existentials – modelling objects – may also be done with a Haskell 98 recursive type:

```
data Accum a = MkAccum { add_value :: a -> Accum a, get_value :: a}
```

but other applications do require existentials.

Prev Type class extensions | Home Up | Next Type annotations in patterns

5.4. Type annotations in patterns

Haskell 98 allows expressions to be annotated with type signatures. With the *-98* option, these annotations are also allowed on patterns:

```
f (x::Int) = fromIntegral x :: Double
```

Moreover type variables in pattern annotations are treated specially: unless the type variable is already bound (by another pattern annotation), it is universally quantified over the pattern and its scope, e.g.

```
snoc (xs::[a]) (x::a) = xs++[x] :: [a]
```

Occurrences of the type variable in type signatures within this scope are bound to this type variable. In the above example the second and third occurrences of *a* are bound by the first. This permits locally defined variables to be given signatures in situations where it would be impossible in Haskell 98:

```
sortImage :: Ord b => (a -> b) -> [a] -> [a]
sortImage (f::a->b) = sortBy cmp
    where
        cmp :: a -> a -> Ordering
        cmp x y = compare (f x) (f y)
```

Note that the relationship between signature declarations and pattern annotations is asymmetrical: pattern annotations may capture type variables in signature declarations, but not vice versa. There is no connection between the type variables in the type signature of *sortImage* and those in its definition, but the occurrence of *a* in the signature of *cmp* is bound by the pattern *(f::a->b)*.

In GHC, type variables bound by pattern annotations are existentially quantified, and so may be instantiated. Thus the following is accepted by GHC but not Hugs:

```
g (xs::[a]) = xs ++ "\n"
```

GHC also allows *result type signatures*, where a type signature is attached to the left side of a function definition, but Hugs does not.

5.5. Implicit parameters

Implicit parameters are implemented as described in *Implicit parameters: dynamic scoping with static types*, J Lewis, MB Shields, E Meijer, J Launchbury, *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, Boston, Jan 2000. Note however that the binding syntax in that paper, using keywords *dlet* and *with*, has been replaced by the form presented below.

(Most of the following, still rather incomplete, documentation is due to Jeff Lewis.)

A variable is called *dynamically bound* when it is bound by the calling context of a function and *statically bound* when bound by the callee's context. In Haskell, all variables are statically bound. Dynamic binding of variables is a notion that goes back to Lisp, but was later discarded in more modern incarnations, such as Scheme, as dynamic binding can be very confusing in an untyped language. Unfortunately typed languages, in particular Hindley-Milner typed languages like Haskell, only support static scoping of variables.

However, by a simple extension to the type class system of Haskell, we can support dynamic binding. Basically, we express the use of a dynamically bound variable as a constraint on the type. These constraints lead to types of the form *(?x::t') => t*, which says "this function uses a dynamically-bound variable *?x* of type *t'*". For example, the following expresses the type of a *sort* function, implicitly parameterized by a comparison function named *cmp*.

```
sort :: (?cmp :: a -> a -> Bool) => [a] -> [a]
```

The dynamic binding constraints are just a new form of predicate in the type class system.

An implicit parameter occurs in an expression using the special form *?x*, where *x* is any valid identifier (e.g. *ord ?x* is a valid expression). Use of this construct also introduces a new dynamic-binding constraint in the type of the expression. For example, the following definition shows how we can define an implicitly parameterized sort function in terms of an explicitly parameterized *sortBy* function:

```
sortBy :: (a -> a -> Bool) -> [a] -> [a]

sort   :: (?cmp :: a -> a -> Bool) => [a] -> [a]
sort    = sortBy ?cmp
```

5.5.1. Implicit-parameter type constraints

Dynamic binding constraints behave just like other type class constraints in that they are automatically propagated. Thus, when a function is used, its implicit parameters are inherited by the function that called it. For example, our *sort* function might be used to pick out the least value in a list:

```
least   :: (?cmp :: a -> a -> Bool) => [a] -> a
least xs = fst (sort xs)
```

Without lifting a finger, the *?cmp* parameter is propagated to become a parameter of *least* as well. With explicit parameters, the default is that parameters must always be explicit propagated. With

implicit parameters, the default is to always propagate them.

An implicit-parameter type constraint differs from other type class constraints in the following way: all uses of a particular implicit parameter must have the same type. This means that the type of *(?x, ?x)* is *(?x::a) => (a,a)*, and not *(?x::a, ?x::b) => (a, b)*, as would be the case for type class constraints.

You can't have an implicit parameter in the context of a class or instance declaration. For example, both these declarations are illegal:

```
class (?x::Int) => C a where ...
instance (?x::a) => Foo [a] where ...
```

Reason: exactly which implicit parameter you pick up depends on exactly where you invoke a function. But the "invocation" of instance declarations is done behind the scenes by the compiler, so it's hard to figure out exactly where it is done. The easiest thing is to outlaw the offending types.

Implicit-parameter constraints do not cause ambiguity. For example, consider:

```
f :: (?x :: [a]) => Int -> Int
f n = n + length ?x

g :: (Read a, Show a) => String -> String
g s = show (read s)
```

Here, *g* has an ambiguous type, and is rejected, but *f* is fine. The binding for *?x* at *f*'s call site is quite unambiguous, and fixes the type *a*.

5.5.2. Implicit-parameter bindings

An implicit parameter is *bound* using the standard *let* or *where* binding forms. For example, we define the *min* function by binding *cmp*:

```
min :: [a] -> a
min  = let ?cmp = (<=) in least
```

A group of implicit-parameter bindings may occur anywhere a normal group of Haskell bindings can occur, except at top level. That is, they can occur in a *let* (including in a list comprehension or do-notation), or a *where* clause. Note the following points:

- An implicit-parameter binding group must be a collection of simple bindings to implicit-style variables (no function-style bindings, and no type signatures); these bindings are neither polymorphic or recursive.
- You may not mix implicit-parameter bindings with ordinary bindings in a single *let* expression; use two nested *let*s instead. (In the case of *where* you are stuck, since you can't nest *where* clauses.)
- You may put multiple implicit-parameter bindings in a single binding group; but they are *not* treated as a mutually recursive group (as ordinary *let* bindings are). Instead they are treated as a non-recursive group, simultaneously binding all the implicit parameters. The bindings are not nested, and may be re-ordered without changing the meaning of the program. For example, consider:

  ```
  f t = let { ?x = t; ?y = ?x+(1::Int) } in ?x + ?y
  ```

The use of *?x* in the binding for *?y* does not "see" the binding for *?x*, so the type of *f* is

```
f :: (?x::Int) => Int -> Int
```

Prev
Type annotations in patterns

Home
Up

Next
Hugs-specific language extensions

Chapter 6. Hugs-specific language extensions

These experimental features are unique to Hugs. Except of the debugging primitives, they require the *-98* option.

6.1. Typed extensible records

Trex is a very powerful and flexible record system. See Section 7.2 of the *Hugs 98 User Manual* for details.

To use equality and *show* on extensible records, a module must import *Hugs.Trex*. This module also defines an empty record value and type:

```
emptyRec :: Rec EmptyRow
```

6.2. Restricted type synonyms

Restricted type synonyms are a mechanism for defining abstract datatypes. You can achieve similar effects, and more portably, using the Haskell 98 module system.

The idea is that you can say that a type synonym is transparent in the definitions of certain functions (the operations on the type), but opaque elsewhere, by writing

```
type Table a b = [(a,b)] in
      empty :: Table a b,
      isEmpty :: Table a b -> Bool,
      add :: a -> b -> Table a b -> Table a b,
      search :: a -> Table a b -> Maybe b

empty = []
isEmpty = null
add a b t = (a,b):t
search = lookup
```

or equivalently

```
type Table a b = [(a,b)] in empty, isEmpty, add, search

empty :: Table a b
empty = []

...
```

See Section 7.3.5 of the *Hugs 98 User Manual* for details.

6.3. Here documents

These expressions (named after similar things in Unix shells) are another way of writing string literals, often useful for large strings. Everything from `` to '' (including newlines and backslashes, but not *$* characters) is treated as literal text, and layout is ignored. The exception is the *$* character, so that you can embed the value of the variable *var* in the string by writing *$(var)*. To get a literal *$* character, write *$$* — single *$* characters are not allowed.

When the *+H* option is given, the following

```
  letter name = ``Dear $(name),
Here are some characters: \ ' ` ".
To learn more, send $$10 to the address below.''
```

is equivalent the Haskell 98 declaration

```
  letter name = "Dear " ++ quote name ++ ",\n\
                \Here are some characters: \\ ' ` \".\n\
                \To learn more, send $10 to the address below."
```

The function

```
  class Quote where
      quote :: a -> String
```

(basically no change for *String* and *Char*, and *show* for everything else) comes from the *Hugs.Quote* module, which also defines several common instances, and should be imported if you use the *$(var)* form. (This module also requires the *-98* option.)

6.4. Hugs debugging primitives

This release of Hugs contains support for debugging by observations inspired by the Andy Gill's Hood library:

1. Andy Gill, *Debugging Haskell by Observing Intermediate Data Structures*, in *Draft Proceedings of the 2000 Haskell Workshop*.

2. The Haskell Object Observation Debugger http://www.haskell.org/hood/.

Hood is a portable Haskell library that implements the combinator

```
Observable a => observe :: String -> a -> a
```

The partial application

```
observe tag
```

behaves exactly like the identity function, but also records the value of data to which it is applied. Any observations made are reported at the end of the computation. The *tag* argument is used to label the observed value when it is reported. Non-strict semantics is preserved — *observe* does not evaluate its second argument.

HugsHood uses the same observation model but differs in a number of ways.

- It is much faster. This is because HugsHood is implemented within the Hugs evaluator and uses primitive builtin functions. Performance depends upon the volume of observations. More frequent observations incur a higher overhead. As a simple comparison, a test program which executed 1 million reductions and made 250 observations incurred a 625 percent overhead when observations were made with the Hood library but just 10 percent when using HugsHood.

  Caveat: When not using observations, the modifications to the evaluator to support HugsHood imposes an overhead of about 6 percent.

- It is possible to easily observe arbitrary data structures. HugsHood implements the primitive

  ```
  observe :: String -> a -> a
  ```

  which is unconstrained by the need to build instances of the *Observable* class for each user defined data type whose values are being observed. HugsHood uses an internal primitive function to display observed values. This may be considered both an advantage and a disadvantage: one does not need to define how to observe values, but one cannot define special user views of data.

- No modification to the program (apart from instrumentation with *observe*) is required. The Hood library must be invoked using a special IO monadic combinator to ensure that observations are collected and displayed.

- There are a number of minor differences in the display format which are a consequence of the Hugs implementation. These are described below.

6.4.1. Using HugsHood

Modules that use HugsHood combinators must import the module *Hugs.Observe*. Its only role is to provide the necessary primitive definitions, namely:

```
primitive observe :: String -> a -> a
primitive bkpt    :: String -> a -> a
primitive setBkpt :: String -> Bool -> IO ()
```

6.4.1.1. Breakpoints

HugsHood implements breakpoints. A program can be instrumented with the *bkpt* function. The partial application

```
bkpt bkpt_name
```

behaves exactly like the identity function, except that before it returns its argument it checks if *bkpt_name* is enabled, and if it is the user is presented with the opportunity to view observed data. A small set of commands is available when Hugs halts due to a breakpoint:

p [*tag_name*]

> Print observations made since the computation began. If an observation tag is suppled then only the associated observations will be displayed. Otherwise all observations will be displayed.

c [*n*]

> Continue with program evaluation. With no arguments, evaluation will continue until another active breakpoint is encountered. The optional numeric argument will skip *n* active breakpoints before stopping.

s *bkpt_name*

> Set a breakpoint.

r [*bkpt_name*]

> Reset a named breakpoint or, if no breakpoint name is supplied, reset all breakpoints.

A breakpoint is by default disabled. It can be enabled by using the **s** command in the debug breakpoint dialogue, or by using the *setBkpt* combinator. Clearly at least one breakpoint must be enabled using *setBkpt* before a breakpoint dialogue can be triggered.

6.4.1.2. Breakpoint Example

Here is a very simple program using the three combinators.

```
import Hugs.Observe

prog n = do { setBkpt "fib" True; putStr $ show (observe "fun" f n) }
```

```
f 0 = 1
f n = n * (bkpt "fib" $ observe "fun" f (n-1))
```

The following sample session shows how the **p** and **c** commands can be used.

```
Main> prog 4
Break @ fib> p

>>>>>>> Observations <<<<<<

fun
  { \ 4  -> _
  }

Break @ fib> c
Break @ fib> p

>>>>>>> Observations <<<<<<

fun
  { \ 4  -> _
  , \ 3  -> _
  }

Break @ fib> c 2
Break @ fib> p

>>>>>>> Observations <<<<<<

fun
  { \ 4  -> _
  , \ 3  -> _
  , \ 2  -> _
  , \ 1  -> _
  }

Break @ fib> c
24
(98 reductions, 299 cells)

>>>>>>> Observations <<<<<<

fun
  { \ 4  -> 24
  , \ 3  -> 6
  , \ 2  -> 2
  , \ 1  -> 1
  , \ 0  -> 1
  }

10 observations recorded
```

6.4.2. Differences from Hood

HugsHood uses a similar style of display to Hood, though there are differences. One trivial difference is that Hood reports tags with a leading "--" while HugsHood does not.

Consider now more significant differences.

6.4.2.1. Observing character strings

HugsHood (and Hood) reports lists using the cons operator.

```
Observe> observe "list" [1..3]
[1,2,3]

>>>>>>> Observations <<<<<<

list
  (1 : 2 : 3 : [])
```

This is too verbose for lists of characters, so HugsHood reports strings in the usual format:

```
Observe> observe "string" ['a'..'d']
"abcd"

>>>>>>> Observations <<<<<<

string
  "abcd"
```

If only the initial part of the string is evaluated, a trailing "..." is reported.

```
Observe> take 2  $ observe "string" ['a'..'d']
"ab"

>>>>>>> Observations <<<<<<

string
  "ab..."
```

This is clearly ambiguous, because evaluating the expression

```
observe "string" "ab..."
```

will give the same result, but in practice the ambiguity should be easy to resolve.

6.4.2.2. Unevaluated expressions

The "_" symbol is used to indicate an unevaluated expression. In Hood all unevaluated expressions will be displayed using "_". In HugsHood, "_" denotes an unevaluated expression, but not all unevaluated expressions are denoted by "_".

For example the expression *fst $ observe "pair" (1,2)* yields

```
-- pair
  (1, _)
```

in both Hugs and HugsHood. However, *fst $ observe "pair" ('a','b')* yields

```
pair
  ('a','b')
```

in HugsHood, and *('a', _)* in Hood. This is because HugsHood (unlike Hood) does not actually record evaluation steps. It merely maintains an internal pointer to that part of the heap representing the tagged expression. If the expression in not in weak head normal form, then it obviously has not been evaluated and so it is reported as just "_"; otherwise it displayed. Integer constants like *1* and *2* are not in WHNF, as they must be coerced to the correct type when evaluated. Characters though are

in WHNF so it is not possible to discern whether a character was evaluated.

Another consequence of the HugsHood implementation by pointers rather than Hood's implementation by tracing evaluation is that the strictness behaviour of a function can be masked. Consider the example:

```
lazy pair = let x = observe "fst" fst pair
                y = snd pair
            in  (y,x)
```

For the expression *lazy (1,2)* Hood reports

```
-- fst
  { \ (1, _)  -> 1
  }
```

while HugsHood reports

```
fst
  { \ (1,2)  -> 1
  }
```

HugsHood should not be used to deduce the strictness behaviour of a function, or it should be done only with caution.

6.4.2.3. Interaction with the root optimisation

The Hugs compiler uses an optimisation when generating code that builds expressions on the heap. If a function definition has the form

```
f arg1 .. argN = ..... f arg1 .. argM .....
```

where $1 \leqslant M \leqslant N$, then the expression graph for *f arg1 .. argM* is copied rather than rebuilt from individual application nodes. This interacts with the observation algorithm so that observing functions of the above form gives unexpected results.

For instance consider the expression

```
observe "fold" foldl (+) 0 [1..3]
```

When the root optimisation is applied to the compilation of *foldl*, we see

```
fold
  { \ primPlusInteger 6 []  -> 6
  , \ { \ 3 3  -> 6
      } 3 (3 : [])  -> 6
  , \ { \ 1 2  -> 3
      } 1 (2 : 3 : [])  -> 6
  , \ { \ 0 1  -> 1
      } 0 (1 : 2 : 3 : [])  -> 6
```

instead of the expected

```
fold
  { \ { \ 0 1  -> 1
      , \ 1 2  -> 3
      , \ 3 3  -> 6
```

```
        } 0 (1 : 2 : 3 : [])  -> 6
    }
```

The first form reports the arguments at each application of *foldl*, while the second reports the arguments for just the initial application (the one marked by *observe*).

The root optimisation can be disabled using the *-R* option. This can be done from the command line or by using **:s** *-R* at the Hugs prompt. If you want to compile the prelude definitions without the root optimisation you must invoke Hugs with the *-R* option.

Testing of execution time with and without the root optimisation for a selection of 23 benchmarks from the nofib suite has been carried out. All but 5 tests resulted in an execution time penalty of less than 3% when running without root optimisation (some even showed a very minor speedup).

6.4.2.4. Known problems

Hugs can produce infinite (cyclic) dictionaries when implementing overloading. The observation reporting mechanism does not detect these at present, which leads to a non-terminating report. We plan to address this in a future release.

6.4.3. Reporting HugsHood bugs

Please report bugs to Richard Watson, <rwatson@usq.edu.au>

In particular, if the message

```
Warning: observation sanity counter > 0
```

appears, and your program has not terminated abnormally, please report the error situation.

Chapter 7. Miscellaneous

7.1. Hugs 98 release history

These are the release notes for the program since it was renamed Hugs 98, reflecting substantial compliance with Haskell 98 (though with numerous optional extensions). Archives of older versions of Gofer and Hugs are still available from Mark Jones's web page.

7.1.1. January 1999 (Beta release)

Headline news for this release includes:

- Hugs goes Haskell 98! Hugs 98 is the first released Haskell system to support the new standard for Haskell 98.
- Hugs goes Open Source! Hugs 98 is the first Hugs release to be distributed as Open source software. Responding to requests from users, this relaxes the conditions of use and distribution of previous releases, and will hopefully make it easier to use Hugs for a wide range of projects.

This release of Hugs also merges the features of several earlier releases into one single system. This includes:

- The module system and dynamic linking facilities of Hugs 1.4 (June 1998);
- The type system extensions (multi-parameter classes, TREX, rank-2 polymorphism, existentials, etc.) of Hugs 1.3c p1 (March 1998);
- New features and modifications to support the draft Haskell 98 standard;
- A whole range of bug fixes and additions for all of the above.

7.1.2. May 1999

This release is largely conformant with Haskell 98, including monad and record syntax, newtypes, strictness annotations, and modules. In addition, it comes packaged with the libraries defined in the most recent version of the Haskell Library Report and with extension libraries that are compatible with GHC 3.0 and later.

Additional features of the system include:

- "Import chasing": a single module may be loaded, and Hugs will chase down all imports as long as module names are the same as file names and the files are found in the current path.
- A simple GUI for Windows to facilitate program development.

- Library extensions to support concepts such as concurrency, mutable variables and arrays, monadic parsing, tracing (for debugging), graphics, and lazy state threads.
- A Win32 library for complete access to windows, graphics, and other important OS functionalities and a graphics library for easy access to Win32 graphics.
- A "foreign interface" mechanism to facilitate interoperability with C.

7.1.3. November 1999

- BSD-style license (replacing the Artistic License)
- new commands **:browse** and **:version**
- experimental multi-instance resolution and **:xplain** command
- functional dependencies
- zero parameter type classes
- better handling of overlapping instances
- various bug fixes

7.1.3.1. February 2000

This is purely a bug-fix release of Hugs98 November 99. It fixes the following problems:

- If you defined an instance which inherited a method via a superclass, hugs would go into an infinite loop. Fortunately, most people weren't doing this (except Chris Okasaki...).
- There were a couple of holes in the implementation of implicit parameters ("*with*" wasn't always being scoped properly, sometimes capturing implicit parameters outside of its scope).
- Functional dependencies weren't being properly propagated in some cases with derived instances ("*instance P ... => Q ...*").

7.1.3.2. July 2000

This is purely a bug-fix release of Hugs98 February 2000.

7.1.4. February 2001

This is a major release that incorporates bug fixes as well as several new features and enhancements that have been developed for Hugs over the last year. It is announced with the intention that it will remain a stable and lightweight implementation of Haskell 98 + extensions for some considerable time.

A list of the most important new features looks as follows:

- A Foreign Function Interface closely modelled after the one provided by GHC.

- Built-in, Hood-like debugging support.
- Parallel list comprehensions, a.k.a. zip-comprehensions.
- A new syntax for recursive monad bindings.
- A new GUI under Windows that doesn't consume all CPU time.
- Support for the MacOS platform integrated into the main distribution.
- Corrections of all bugs reported for the January 2001 beta release.

7.1.5. December 2001

The most important features of this new release are:

- The incompatibilities between Hugs and the Haskell Graphics Library have been fixed, and binaries for the HGL are now available on the Hugs download page.
- The missing standard libraries *Directory*, *CPUTime*, *Time* and *Locale* have been added along with a complete implementation of Haskell 98 *IO*.
- Hugs is now delivered with most of the *hslibs* libraries installed in the *lib/exts/* directory. The added modules cover the *Edison*, *Parsec*, *HaXml*, *QuickCheck*, *concurrent*, *monad* and *html* subdirectories of *hslibs*.
- The **:set** option now refuses the user to set a module search path that doesn't contain the *Prelude*. This is to protect users from accidentally rendering their Hugs setups unusable, esp. so on Windows machines where the options are persisted to the Registry.
- MacOS X is now one of the supported unix ports, with pre-built binaries available on the download page.
- Experimental support is provided for hierarchical module names, where a module name *A.B.C* is mapped onto the file path *A/B/C*{*.hs,.lhs*} and appended to each of the path prefixes in *HUGSPATH* until the name of a readable file is found.

7.1.6. November 2002

Feature highlights of this new release are:

- Much improved FFI support (contributed by Alastair Reid), bringing Hugs really very close to the Haskell FFI specification.
- Adoption of a significant subset of GHC's hierarchical libraries (contributed by Ross Paterson).
- An (allegedly) complete implementation of the Haskell98 module system (Sigbjorn Finne).
- Numerous bug fixes since the previous major release in Dec 2001.

7.1.7. November 2003

There has been substantial internal restructuring. In addition to numerous bug fixes, user-visible changes include:

- The beginnings of a User's Guide (though still incomplete).
- The *Double* type is now double-precision on most architectures.
- Hugs now relies on the same hierarchical libraries as GHC and Nhc98, and provides almost all of them. For now, compatibility with the old libraries is provided by stub modules, but users are encouraged to migrate to the new libraries.
- Full support for imprecise exceptions (but not asynchronous ones). Most runtime errors are now reported by applying *print* to an *Exception* (formerly the built-in printer was applied to the faulty redex).
- Integrated .NET support (on Windows platforms).
- The *-e*, *-f*, *-i*, *-N*, *-W* and *-X* options and the **:project** command have been removed.

Prev Home
Hugs debugging primitives