The Yale Haskell Users Manual
Version Y2.0-beta
The Yale Haskell Group

Paul Hudak, John Peterson, Sandra Loosemore

Research Report YALEU/DCS/RR-935
September, 1992

The Yale Haskell Users Manual

Version Y2.0-beta

September 25, 1992

The Yale Haskell Group

Department of Computer Science

Yale University

New Haven, CT 06520

1 Introduction

This manual describes version Y2.0-beta-2 of the Yale Haskell system. The Haskell language is fully defined in the "Haskell Report" [2]. This document refers to version 1.2 of the Haskell report; we advise that copies of older versions be replaced by this new report as there are many significant differences.

1.1 About Yale Haskell

The Yale Haskell compiler is part of a Lisp environment; the compiler itself is written in a small Lisp dialect similar to Scheme or EuLisp which runs on top of Common Lisp. Haskell programs are translated into Lisp and then compiled by the underlying Lisp compiler. The Yale Haskell system retains the interactive nature of the Lisp environment: programs are compiled, executed, and modified within a single process. We do not yet generate stand-alone executable Haskell programs.

Yale Haskell can be used either by itself using a command style interface or as part of a larger programming environment. We supply an interface for GNU Emacs which runs Yale Haskell as a subprocess. Within the Emacs enviroment Haskell expressions can be evaluated during the editing of a Haskell source file. A tutorial on the Haskell language is also part of the Emacs environment.

1.2 The 2.0-beta-2 Release

This is the 'beta-2' release of the Yale Haskell compiler, version 2.0. We appreciate any and all user comments, suggestions, and bug reports. We will not be issuing patches to the beta system; instead, we hope to collect and fix all reported bugs in the official release of the system. This official release should occur about one month after the beta release.

We are releasing the system in two forms: an executable built on top of CMU Common Lisp and a full source release which can be used to build either on the CMU system, Lucid Lisp, or AKCL. We have not yet ported to other Lisp systems; porting instructions are included with the release but we do not feel that it is ready for porting by unassisted users.

While we are very concerned with both speed of compilation and the speed of execution, we have not yet had time to address either of these issues completely. We expect significant improvements in both aspects of our system in future releases. Execution time is especially dependent on the underlying Lisp compiler. We have not yet had time to tune our generated code (through declarations and compiler settings) to achieve optimal performance. Please avoid benchmarking our generated code until we have had time to improve this part of the system. Another problem we are working on is the size of the system — we currently generate a very large executable, mainly due to the embedded Lisp system.

A number of aspects of the system are as yet incomplete. These include:

- Some error messages are rather cryptic.
- Some errors are not recovered from correctly, causing the compiler to crash later during compilation.
- Some IO requests are not implemented or are not meaningful in the Haskell environment we supply.
- We do not handle interface files yet.

- The system has not been extensively tested.

Since we are releasing the full sources to our system, users are free to modify the compiler or port the code to other Lisp systems. However, we are not yet in a position to support these activities. The sources are still changing rapidly — compiler modifications you make may no longer be valid when the next release comes. The code is as yet poorly documented.

1.3 Haskell Mailboxes

There are a number of mailboxes associated with Haskell and the Yale Haskell Project, as described below.

- `haskell-requests@cs.yale.edu`
  Use this address to communicate with us about our compiler. We also maintain the Haskell mailing list.
- `haskell-bugs@cs.yale.edu`
  Send bug reports to this address.
- `haskell@cs.yale.edu`
  This is a mailing list for general issues concerning the language. Messages sent to this address will be forwarded to all members of the Haskell discussion list, which has a USA distribution managed by Yale and an European distribution managed by the University of Glasgow. Please do not send bug reports or other messages specifically about the Yale implementation to this list!

2 Using Haskell From Emacs

We supply two different programming environments with Yale Haskell: an editor based environment built on the Emacs editor and a command interpreter. Both provide the same functionality but the Emacs environment is much more convenient and is the recommended way of using Yale Haskell. If you plan to use the command interpreter, you can skip this section.

2.1 The Haskell Emacs Mode

Before using the Haskell Emacs mode, you must first configure your `.emacs` file as described in section 7. Once your `.emacs` file has been modified to recognize Haskell programs, the Emacs editor will enter Haskell mode whenever a file with extension `.hs` or `.lhs` is visited. Haskell mode provides commands to control the Haskell compiler as an inferior process under Emacs. There are two basic commands: evaluating and printing expressions (`C-c e`) and running dialogues (`C-c r`). They operate as follows:

1. Emacs prompts you for a Haskell expression. For `C-c e`, the value of expression must in class `Text`. For `C-c r`, the value must be of type `Dialogue`.
2. An inferior Haskell process is started if one is not already running.
3. All `.hs` and `.lhs` buffers are saved.

4. The file containing the cursor is compiled by the Haskell compiler if necessary.
5. The expression is evaluated or run in the context of the module containing the cursor (there may be more than one module in a file). The Haskell dialogue buffer, named `*haskell*`, will pop up.

All interaction with the Haskell process occurs in the Haskell dialogue buffer. This buffer receives status and error messages and well as IO operations on `stdin` and `stdout`. The cursor moves to this buffer automatically if a running program requests input.

2.2 Scratch Pads

While developing Haskell programs it is often convenient to temporarily add definitions to a module. A *scratch pad* is a special buffer associated with a module (not with a file!) in which you may place definitions for use during the editing session. Changes to the scratch pad definitions do not require full recompilation of the module and are especially useful when large programs are being developed. The scratch pad is also a good place to enter and test new functions before moving them into the source file. When you use either the `C-c e` or `C-c r` commands are used in a scratch pad, you may reference the definitions in the pad.

To avoid global recompilation, there are a number of restrictions on definitions which may be placed in a scratch pad:

- A scratch pad may not contain any import or fixity declarations. Only names visible in the associated module or within the pad itself can be referenced.
- No definition in a pad can be exported from the associated module.
- The pad cannot redefine anything in the associated module.

When Haskell is not started from within a Haskell source file, a scratch pad onto an empty program (`*Main-pad*`) pops up. You can use this pad to play with the system without creating a `.hs` file.

2.3 Emacs Commands

This section describes in detail the commands available when running Haskell mode.

In the following, the *current module* is determined by the type of buffer containing the cursor. In a `.hs` buffer, the current module is the module definition containing the cursor. In a pad, it is the module associated with the pad. In the `*haskell*` buffer, the most recently used module is remembered.

A *dialogue* is any Haskell expression that is of type `Dialogue`.

The minibuffer commands `M-p` and `M-n` allow you to search back and forth through the history of the last 30 expressions you have typed in when Haskell prompts you for something to evaluate.

haskell-eval (`C-c e`) This command prompts you for a Haskell expression to be evaluated. Evaluation occurs in the context of the current module. The result is then printed; the value must be in class `Text`.

haskell-run (`C-c r`) This command prompts you for a Haskell dialogue and runs it. Its behavior is similar to `haskell-eval`.

haskell-run-main (`C-c m`) This command runs the dialogue named `Main` in the current module.

haskell-run-file (`C-c C-r`) Runs all the dialogues in the current file.

haskell-get-pad (`C-c p`) Pops up the buffer containing the scratch pad for the current module, and makes it the current buffer.

haskell-printers (`C-c C-p`) This command pops up a buffer that lets you set Haskell compiler print options from a menu and makes it the current buffer. Use `?` to get further help.

haskell-compile (`C-c c`) The Lisp code produced by the Yale Haskell compiler can either be interpreted or compiled. Since compiling the generated Lisp code into machine language takes much longer than generating it from the Haskell source program, you probably don't want to do this during program development. So, commands such as `C-c e` and `C-c r` do not compile the code generated or write any compiled code into files. When you are ready to create fully compiled code and save it in output files, use the `C-c c` command. This command recursively compiles all imported compilation units. Use the `:cspeed` command to select the fast (nonoptimizing) lisp compiler or the slow (optimizing) one.

haskell-exit (`C-c q`) This command terminates the Haskell subprocess. It leaves all buffers open.

haskell-switch (`C-c h`) Pops up the `*haskell*` buffer and makes it the current buffer.

haskell-interrupt (`C-c i`) This command sends an interrupt to the Haskell subprocess. You can use it to terminate execution of a running program (for example, if it gets stuck in an infinite loop) and return to the command loop.

haskell-please-recover (`C-c d`) Sometimes Lisp errors may cause the synchronization between Emacs and Haskell to be lost. This command attempts to reset both the Emacs interface and the Haskell subprocess to a known state. You should not ordinarily need to use this command, since the Emacs interface tries to recognize when Lisp errors have occured and reset itself automatically.

haskell-command (`C-c :`) You can submit commands directly to the command interface with this command. Use `C-c :cspeed fast` and `C-c :speed slow` to select the lisp compiler used by `C-c c`.

haskell Starts up a Haskell subprocess, popping up the `*haskell*` buffer. You normally don't need to do this explicitly since any of the commands that cause code to be evaluated or compiled will also start up a Haskell subprocess if there isn't already one running.

haskell-mode Puts the current buffer into the Haskell editing mode.

haskell-tutorial Starts the online Haskell tutorial.

2.4 The Haskell Tutorial

An online supplement to the Hudak and Fasel tutorial (supplied in the `$HASKELL/doc` directory and published in Sigplan Notices) is available. Start this tutorial using the command `M-x haskell-tutorial`. This will explain some of the Emacs commands as well as the Haskell language.

3 The Command Interface

This section gives brief descriptions of all commands used by the command interface. Users of the Emacs environment can skip this section.

To enter the command interface, execute the program `$HASKELLPROG`.

The command interface gives you control over the compiler and surrounding environment and provides an incremental compilation ability through program *extensions*. An extension is a Haskell program which is scoped within an existing module.

The command interface reads both Haskell program extensions and system commands. Commands start with a `:` and control the compiler and the environment. Commands which refer to an extension always deal with the most recently entered extension within the context of the current module. The current module name is used as the system prompt.

Any input line which does not begin with `:` is assumed to be a line of Haskell source code and is added to the current extension. There are two abbreviations provided within extensions. A line beginning with `=` prints an expression. This expands into a definition of a dialogue to accomplish the printing when executed. Lines beginning with `@` run user dialogues. All other lines are treated as ordinary Haskell source code.

Command arguments are separated by whitespace. All commands may be abbreviated using the shortest unique prefix of the command name. For a description of them see Figure 1.

The `:eval`, `:save`, `:clear` and `:kill` commands terminate the current extension. All other commands allow the current extension to be continued.

Any command using a file refers to the most recently referenced file when no file is provided. Files extensions are implicit in the commands and should not be supplied.

To interrupt a running Haskell program, use `^C` (or whatever interrupt character is used by your system).

3.1 An Example

```
Haskell Y2.0 Command Interface. Type :? for help
Initially, Haskell evaluates in a module containing only the Prelude
Main> fact 0 = 1
Main> fact n = n*fact(n-1)
Main> :save This makes fact an addition to Main
Main> = fact 4 This will print 4! when evaluated. No need to save this.
Main> :eval
Evaluating temp_1 temp1 is generated by expanding = to Haskell syntax
24
Main> :cd $HASKELL/progs/demo
Main> :run fact An interactive factorial supplied in demo
Evaluating main. The name of the variable in fact being evaluated
Type in N: 4
24
Extensions to the original Main are now lost.
Main> adds (x:xs) (y:ys) = x+y : adds xs ys
Main> :s
Main> f = 0 : 1 : adds f tail f
```

Commands:	
`:?`	Prints a help file.
`:eval`	Evaluate any executable definitions in the current extension. The extension may still be saved.
`:save`	Save the current extension. If any errors are found, nothing will be saved.
`:clear`	Remove all saved extensions in the current module
`:Main`	Enter the empty Main module.
`:quit`	Exit the Haskell system.
`:module` *module*	Set the current module. The `:run` command also sets the current module. Initially, an empty module `Main` which imports the prelude is available.
`:run` *file*	Compile and load a file. The directory and filename default to the most recently used. If the file contains a definition of `main` it will be run.
`:compile` *file*	Like `:r` except than no execution occurs.
`:load` *file*	Load a Haskell program into memory without compiling generated Lisp code or creating a compiled output file.
`:cd` *dir*	Set the current directory (absolute path only)
`:list`	List the current extension.
`:kill`	Kill (erase) current extension.
`:p?`	List available printers.
`:p=` *printers*	Set printers.
`:p+` *printers*	Turn on printers.
`:p-` *printers*	Turn off printers.
`:cspeed fast | slow`	Select the compiler for `:c`.
`:(`*lisp – code*`)`	Evaluate a Lisp expression.

Figure 1: Command interface directives

```
Main> :s
[TYPE-ERROR] Phase error in phase TYPE:
Type conflict: type [a] does not match [a] -> [a]
Error occurred at line 2 in file interactive.
While type checking
adds (f) tail
Argument type mismatch
Types: Num b => [b]
       [c] -> [c]
Cannot save: errors encountered.
Main> f = 0 : 1 : adds f (tail f)
Main> :s
Main> = take 5 f Since f is infinite, print only the first 5 elements
Main> :e
Evaluating temp_2
[0,1,1,2,3]
```

4 The Compilation System

The compilation system is responsible for gathering all components of a program for compilation. When a single file contains the entire program, this task is trivial. However, when a program contains more than one file, compilation units must be used.

4.1 Compilation Units

A Haskell program consists of a collection of *modules*. The Haskell report does not define how this collection is to be assembled; this is left to the implementation.

Yale Haskell defines the following objects:

Module The syntax for a module (actually, a module *implementation*) is as described in the Haskell report. Modules cannot span file boundaries.

File Each file must contain one or more modules.

Compilation Unit A compilation unit is described by a `.hu` file. This file defines a set of source file names and a set of imported compilation units.

Program A program consists of the modules in a compilation unit combined with all modules found in imported compilation units.

The set of modules in a program must satisfy two criteria:

1. No module may appear more than once. It is an error for two modules in a program to have the same name.

2. All modules referenced in import statements must be declared in the program. The module `Prelude` is implicitly added to every program by the system.

Compilation units in Yale Haskell are defined by files with the `.hu` extension. They are only necessary for programs containing more than one source file. The contents of the `.hu` file are similar to the command line used to invoke other compilers from a command line. The `.hu` file simply contains a list of source files to be compiled as well as other units to be imported. Some of the functionality of the `make` utility is also provided since the compiler will compile any uncompiled units needed and determine whether a source file has been updated after the last compile.

Yale Haskell places one significant restriction on compilation units: each compilation unit must itself be a valid program. The effect of this restriction is that compilation units cannot be mutually dependent. If two modules are mutually recursive, each importing the other, you must place them in the same compilation unit. Within a compilation unit, there are no restrictions on the importing of modules.

4.2 Compilation Unit Files

Compilation units are defined in files with a `.hu` extension. A `.hu` file contains a list of file names, one per line. Each file name should be either a source (`.hs` or `.lhs`) file or a unit file (`.hu`). The former specifies a consituent file of the compilation unit, while the latter specifies a unit to import. If you specify a file name without an extension, the system will look for a unit file first and then a source file. File names may appear in any order. You can omit directories from the file names if the file is in the same directory as the `.hu` file.

Compiling a unit with the `C-c c` or `:compile` writes two output files: an object code file and an interface file. Both of these files contain compiled Lisp code. The name of the interface file is suffixed with `-hci`. The extension used for the output files varies depending on the underlying Lisp implementation.

4.3 Recompilation as Needed

Compilation units serve to break a program into separately compilable pieces. To run the program, all component pieces must be compiled and loaded into the system. Yale Haskell attempts to reduce the amount of processing needed when compiling a program by saving previously compiled units in files and only recompiling them when they change.

Once compiled, a compilation unit may become *outdated* in one of several ways:

- A source file within the compilation unit changes.
- An imported unit becomes outdated.
- The definition of the unit changes.

In addition to these three possibilities, a unit must certainly be compiled if no compiled version exists.

The compilation system will not compile a unit based on outdated imported information; it will attempt to bring the imported units up-to-date by compiling them if necessary.

4.4 Using Compilation Units

Some care must be taken when using compilation units from either the Emacs interface or the command interface. The program being compiled must be specified by a compilation unit. To avoid creating `.hu` files for one file programs, the compiler can be called with a single source file. Otherwise, the system needs to know about the compilation unit in use.

In the command interface, the file used by the `:compile` and `:load` commands is actually a compilation unit. The `.hu` extension is not needed: the system will look for a `.hu` before looking for a source file. In a program consisting of a unit `U` with two source files `A` and `B`, you need to load unit `U` after editing either `A` or `B`. Trying to compile `A` or `B` directly will result in an error.

The Emacs interface assumes that the unit file associated with a source file has the same name but an extension of `.hu`. If the name is different, add a comment line containing `unit: file` at the start of the file, as in: `-- unit: myunit`

4.5 Defining Primitive Units

Primitive functions in the Yale Haskell system are defined in Lisp; they are compiled to produce units whose definitions can be imported just like those arising from compiling Haskell source. We have not yet documented the use of primitives.

5 Errors and Debugging

5.1 Type Errors

Type errors can be difficult to deal with. If the error becomes especially difficult to uncover, turn on the `type` printer. Do this from Emacs via `C-c C-p` and selecting the printer for `type` or use `:p+ type` in the command interface.

We suggest that top level declarations be affixed with type signatures. This practice has a number of advantages:

- Incorrect definitions of a function are caught by the type signature instead of causing a type error at a call site.
- Unwanted overloading is eliminated.
- The signatures provide an additional level of documentation.

When dealing with an ordinary type signature mismatch, check that the proper number and types of arguments are being passed to the function. Also examine infix operators and make sure that their precedences are what you expect.

The pattern binding rule can be the source of unexpected errors. This rule restricts overloading of variables bound in pattern bindings. Global overloaded constants, structures which contain overloaded functions, and functions without arguments (that is, defined by a pattern binding) can cause an error when they have no explicit type signature.

The class system causes type checking to fail in a different way for overloaded function parameters than for non-overloaded parameters. For example,

```
length True
```

gives a standard type mismatch error. An overloaded function, like +, handles type mismatches differently. The + function accepts any type of argument in class `Num`. An error in a call to +, as in:

```
True + False
```

results in a message indicating that `Bool` is not in class `Num`. Such messages usually result from ordinary type mismatches.

Another source of subtle type errors are overloaded numeric constants. An integer constant has the typing `Num a => a`. When integers are used unexpectedly, the type unification algorithm complains that the parameter type is not in class `Num`. For example, in:

```
length 3
```

the type error generated indicates that lists (the expected argument type to `length`) are not a member of class `Num`.

Application is represented internally by a type constructor named `Arrow`. When a functional object is used in a situation in which it is constrained by a class, an "instance not found" for type `Arrow` is generated. For example, in:

```
(max 1) + 1
```

the argument to + must be in class `Num`. The value of `max 1` is a function since both arguments have not been supplied. Since functions are not in class `Num` an error message indicating that `Arrow` is not a member of `Num` is generated. The omission of a function argument is the most common way of generating this error. Tuples and lists also have internal type names used in error messages: `Tuple`*k* and `List`.

Another typing error that can be difficult to understand is the ambiguous context. These are often generated when a specific type constructor does not instantiate all type parameters in a data type. For example, the typing of `[]` is `(List a)`. By itself, `[]` does not require a specific element type in the list. However, instance declarations for the list data type often place constraints on the list components. For example, `[a]` is in class `Text` only when a is. In

```
print []
```

an ambiguous context occurs since `print` requires that the list elements, whose type remains unknown, be in `Text`. This can be avoided using a type signature:

```
print ([] :: [Int])
```

since this removes the type variable a from the signature of `[]`.

Another common type-related error can occur in instance declarations. Instance declarations require the user to explicitly state the entire context associated with an instance. If the body of an instance requires a more general context than declared in the context component of the instance declaration, an insufficient context error message will occur.

5.2 Runtime Errors

When the Haskell `error` function is called at runtime, the system displays the message associated with the error and aborts execution. There is currently no way to examine the system state when this occurs. Some constructs, such as `case` statements, add error calls to the program.

Add explicit "error handlers" to non-exhaustive case statements and function definitions; this is good coding practice and eases debugging. It is also good coding practice to avoid non-disjoint patterns whenever possible (such patterns also tend to translate into less efficient code than patterns written disjointly).

Using `Int` arithmetic can lead to serious runtime errors when overflows occur. Only unsafe `Int` operations are currently provided. If you run into memory protection problems, illegal instructions, or other mysterious errors when the program executes, check for potential overflow problems. Changing `Int` to `Integer` should fix this problem.

If you think your program is in an infinite loop, use `C-c i` from the Emacs or `^C` from the command interface to interrupt the Haskell process.

6 Performance Issues

Many different factors affect the speed of the compiled Haskell code and the compilation time. This section deals with both of these issues.

6.1 Improving Compilation Time

The best way to reduce the time of compilation is to break your program into small compilation units. This is not only good programming style, but also helps to isolate compilation errors and allows you to take advantage of the separate compilation capability of the system. In particular, you should place common datatype declarations in a separate file whenever possible — this conforms with good application of abstract datatype principles, if nothing else. Smaller compilations units also use much less memory during compilation.

Some Haskell constructs expand into large amounts of code; this must be taken into account when considering the size of a compilation unit. Derived instances for types with many data constructors contain large amounts of code. Pattern matching and list comprehensions can generate unexpectedly large amounts of code.

The underlying lisp compiler can be used in either a fast (nonoptimizing) mode or a slow (optimizing) mode. By default, the fast compiler is used. If you want to get the best possible code out of the system you should select the slow compiler using the `:cspeed` command.

6.2 Generating Faster Code

Two factors affect the speed of the compiled code: the efficiency with which Haskell constructs are translated into Lisp code and the speed of the primitives used. The math primitives are the most visible. The performance of the primitives reflects the underlying Lisp system. Some facts about the primitives:

- `Int` arithmetic is very fast. It is usually done inline with a single instruction.

References

[1] Chuck Consel. Fast strictness analysis via symbolic fixpoint iteration. Technical Report YALEU/DCS/RR-867, Yale University Department of Computer Science, September 1991.

[2] Paul Hudak, Simon Peyton Jones, Philip L. Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language, version 1.1. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, August 1991.

[3] S. L. Peyton Jones. Flic – a functional language intermediate code. *ACM SIGPLAN Notices*, 23(8):30–48, 1988.