GOFER

Gofer 2.28 release notes

Mark P. Jones

February 1993

Contents

This document is intended to be used as a supplement to the original user manual "An introduction to Gofer version 2.20" and release notes for Gofer 2.21 (previously supplied in a file called 'update').

If you would like to be informed when bug-fixes or further versions become available, please contact me at `jones-mark@cs.yale.edu` (if you have not already done so) and I will add your name to the list.

Please contact me if you have any questions about the Gofer system, or if you need some advice or help to complete a port of Gofer to a new platform.

This LaTeX version of the release notes was prepared by Jeroen Fokker, `jeroen@cs.ruu.nl`.

Acknowledgements

A lot of people have contributed to the development of Gofer 2.28 with their support, encouragement, suggestions, comments and bug reports. There are a lot of people to thank:

Ray Bellis, Brent Benson, David Bolton, Rodney Brown, Dave Cattrall, Manuel Chakravarty, Rami El Charif, Stuart Clayman, Andy Duncan, Bernd Eckenfels, Stephen Eldridge, Jeroen Fokker, Andy Gill, Annius Groenink, Dipankar Gupta, Guenter Huebel, Jon Hallett, Kevin Hammond, Peter Hancock, Ian Holyer, Andrew Kennedy, Marnix Klooster, Tom Lane, Hiroyuki Matsuda, Aiden McCaughey, Tobias Nipkow, Rainer Orth, Will Partain, Simon Peyton Jones, Ian Poole, Mark Raemer, Dave Rushall, Julian Seward, Carol Tumey, Goran Uddeborg, Gavin Wraith, Bryan Scattergood, Matthew Smith, Bernard Sufrin, Philip Wadler.

This list isn't complete, and I apologize in advance if I have inadvertently left your name out.

1 Minor enhancements and bugfixes

The following sections list the minor enhancements and bugfixes that have been made to Gofer since the release of Gofer version 2.23. More significant changes are described in later sections.

1.1 Enhancements

- For systems without the restrictions of older PCs, Gofer now uses multiple hash tables to speed the lookup of globally defined functions. Loading large programs into Gofer is now much faster as a result. In one example, the time taken to load a 13,000 line program spread across 40 individual script files was reduced by a factor of five!
- For the most most part, internal errors (which shouldn't normally appear anyway) no longer terminate the interpreter.
- Better handling for programs with objects whose type involves more than 26 type variables (though whether anyone has real practical applications for such beasts, I'm rather doubtful).
- The Gofer system now supports I/O requests `GetProgName`, `GetArgs` and `GetEnv`. The first two requests don't have any sensible interpretation within the interpreter, so `GetProgName` always returns `""`, while `GetArgs` returns `[]`. These I/O requests are most useful when producing standalone applications with the Gofer compiler where they do indeed give the name of the program and the list of command line arguments as expected.
- Added primitives for direct comparison of characters. The original definitions of character equality and ordering in terms of the equality and ordering on integers was elegant, but for some examples, a substantial number of the total reductions in a given program was taken up with calls to ord, an unnecessary distraction.
- Small improvements in the speed of execution of the runtime machine, particularly when Gofer is compiled using the GNU C compiler.
- Enabled the use of GNU C specific options to store frequently used global variables in CPU registers. This is perhaps most useful for speeding up the performance of standalone applications produced using the Gofer compiler.
- Changed definitions in standard preludes to provide overloaded versions of `sum`, `product`, `sums`, `products`, `abs`, `signum` and `(^)`. Also added a `genericLength` function as in Haskell. Finally, added `Text` as a superclass of `Num`, again for Haskell compatibility.
- Added a new primitive function: `openfile :: String -> String` that can be used to read the contents of a file (named by the argument string) as a (lazy) stream of characters. (The implementation is in terms of a primitive which can also be used to implement the hbc `openFile` function, provided that you also define the Either datatype used there.)
- Added support for a simple selection of operators for monadic I/O, mutable variables etc. based on Lambda var (developed at Yale) and the Glasgow I/O system. I will provide more documentation on this as soon as there is a better consensus on the names of the datatypes and functions that should be included in systems like this.

- The error function is now implemented using a primitive function.
- Added support for floating point primitives:

```
pi          :: Float
sin, asin,
cos, acos,
tan, atan,
log, log10,
exp, sqrt   :: Float -> Float
atan2       :: Float -> Float -> Float
truncate    :: Float -> Int
```

- Added support for the use of GNU readline (or equivalent) library to be used to enhance the user interface with command line editing. See the source makefile for instructions on how to use this.
- Added floating point support to PC version of Gofer (even the version for humble 8086 PCs will now support floating point). Thanks to Jeroen Fokker for this!
- I/O datatype definitions and otherwise symbol are now builtin to the Gofer system.
- Other minor tweaks and improvements.

1.2 Bug fixes

Nobody really likes to dwell on bugs, especially when they have been eliminated. But for those of you who want to know, here is a summary of the bugs discovered and fixed in Gofer 2.28:

- End of file does not imply end of line (only significant on certain systems... I has made an assumption which happens to hold under DOS and Unix, but was not true for other systems).
- Code generator produced incorrect code for some conditional expressions involving local variables (fairly obscure).
- Some conditional expressions entered into the interpreter were evaluated incorrectly, leading to unexpected evaluation errors.
- A small potential space leak concerned with saving the names of files passed to the editor from within Gofer was eliminated.
- A subtle bug, which only occurred when a garbage collection occurred in the middle of an attempt to update a cell with an indirection has been fixed.
- Fixing the definitions of the `div` and `quot` operators to agree with Haskell 1.2 (these had been changed in the transition from 1.1 to 1.2 without my noticing).
- Corrected bug in string matching code (part of the `:names` command) which previously allowed '`*e*p`' to match with '`negate`'!
- Nested comments were not always handled correctly when they occurred at the very end of a script file.

- Added new clauses to parser to improve and correct error messages produced by some examples.
- Other miscellaneous tweaks and fixes.

There are no other currently known bugs in Gofer. But someone is bound to find a new one within hours of the release of 2.28 if past experience is anything to go by. If that someone is you, please let me know!

2 User interface extensions

The user interface of the previous release has been extended a little to support a range of new features, intended to make the Gofer environment more convenient for program development. Further details are given in the following sections.

2.1 Customizing the Gofer system

Often there will be several people using Gofer on the same system. Not everyone will want to be using the system in the same way. For example, some users may wish to use their own version of the prelude or start the interpreter with particular command line options.

It has always been possible to do this by installing Gofer in an appropriate manner. But, having had more than a couple of enquiries about this, I wanted to take some time to spell the process out more clearly. The following description will be biased towards those people using Gofer on Unix-like systems, but the same basic principles can be applied with other operating systems too.

The Gofer interpreter and prelude files will typically be installed in a given directory, accessible to all users on the system. For the sake of this example, let's assume that this is `/usr/local/lib/Gofer`. Each user could take a copy of the Gofer interpreter into their own file space, but a much better option is for each user to use a short script file stored somewhere on their path. For example, the path on my Unix account includes a subdirectory called bin and I store the following script file 'gofer' in this directory:

```
#!/bin/sh
#
# A simple shell script to invoke the Gofer interpreter and set
# the path to the prelude file.  Ultimately, you might want to
# copy this file into your own bin directory so that you can record
# your favourite command line settings or use a different prelude
# file ...
#
GOFER=/usr/local/lib/Gofer/standard.prelude
export GOFER
exec /usr/local/lib/Gofer/gofer $*
```

I happen to use the standard prelude file and the default settings for all the command line options. If, for example, I wanted to use a different prelude file, a smaller heap and omit the printing of statistics about the number of reductions and cells used in an evaluation, I can modify the script to reflect this:

```
#!/bin/sh
#
```

```
# A modified version of the above script
#
GOFER=/usr/local/lib/Gofer/simple.prelude
export GOFER
exec /usr/local/lib/Gofer/gofer -h20000 -s $*
```

Of course, it is also possible to keep both of these short scripts in my bin directory, so that I have the choice of starting up Gofer in several different configurations, depending on the kind of work I'm going to be doing with it.

2.2 Command line options

Gofer 2.28 supports a number of options which can be set, either on the command line when Gofer interpreter is started, or using the `:set` command within in the interpreter. Using the `:set` command without any arguments produces a list of all the command line options available:

```
? :set
TOGGLES: groups begin with +/- to turn options on/off resp.
s    Print no. reductions/cells after eval
t    Print type after evaluation
d    Show dictionary values in output exprs
f    Terminate evaluation on first error
g    Print no. cells recovered after gc
c    Test conformality for pattern bindings
l    Literate scripts as default
e    Warn about errors in literate scripts
i    Apply fromInteger to integer literals
o    Optimise use of (&&) and (||)
u    Catch ambiguously typed top-level vars
.    Print dots to show progress
w    Always show which files loaded
1    Overload singleton list notation
k    Show kind errors in full

OTHER OPTIONS: (leading + or - makes no difference)
hnum Set heap size (cannot be changed within Gofer)
pstr Set prompt string to str
rstr Set repeat last expression string to str

Current settings: +sfceow1 -tdgliu.k -h100000 -p? -r$$
?
```

Most of these are the same as in the previous release of Gofer. The following sections outline the few changes that have been made. The '`1`' and '`k`' toggles are for use with constructor classes and will be described in Section 4.

2.2.1 Print dots to show progress

One of the first differences that you might notice when running the new version of Gofer is that the rows of dots printed when loading a script file:

```
? :l examples
Reading script file "examples":
```

```
Parsing.................................
Dependency analysis.....................
Type checking...........................
Compiling...............................

Gofer session for:
/usr/local/lib/Gofer/standard.prelude
examples
?
```

are no longer printed while script files are loaded. The rows of dots are useful for showing progress on slow machines (like the PC on which Gofer was originally developed) where it is reassuring to know that the system has not crashed, and is simply working its way through one particular phase of the system. However, on a faster system, the dots are not necessary and printing them can impose a surprising overhead on the time it takes to load files. As a default, Gofer now simply prints the names of each phase (Parsing, Dependency Analysis, Type checking and Compiling) and, when that phase is complete, backspaces over it to erase it from the screen. If you are fortunate enough to be using a fast machine, you may not always see the individual words as they flash past. After loading a file, your screen will typically look something like this:

```
? :l examples
Reading script file "examples":

Gofer session for:
/usr/local/lib/Gofer/standard.prelude
examples
?
```

On some systems, the use of backspace characters to erase a line may not work properly. One particular example of this occurs if you try to run Gofer from within emacs. In this case, you may prefer to use the original setting, printing the lines of dots by giving the command:

```
:set +.
```

The default setting is (as illustrated above, `:set -.`). In practice, you will probably want to include the appropriate setting for this option in your startup script (see Section 2.1).

2.2.2 Always show which files loaded

Some people may feel that the list of filenames printed by Gofer after successfully loading one or more script files is redundant. This is particularly likely if you are using the (usually default) `:set -.` option since the list of files loaded will probably still be on the screen. The list of filenames can be suppressed using the `:set -w` option as follows:

```
? :l examples
Reading script file "examples":

Gofer session for:
/usr/local/lib/Gofer/standard.prelude
```

```
examples
? :set -w
? :l examples
Reading script file "examples":
?
```

The default setting can be recovered using a `:set +w` command.

Note that you can also use the `:info` command (without any arguments) as described in Section 2.3.2 to find out the list of files loaded into the current Gofer session. This should be particularly useful if you choose the `:set -w` option.

2.2.3 Set repeat string

The previous expression entered into the Gofer system can be recalled as part of the next expression using the symbol `$$`:

```
? map (1+) [1..10]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
(101 reductions, 189 cells)
? filter even $$
[2, 4, 6, 8, 10]
(130 reductions, 215 cells)
?
```

This feature was provided and documented in the previous release of Gofer. However, it is possible that you may prefer to use a different character string. This is the purpose of the `-rstr` option which sets the repeat string to str. For example, user's of SML might be more comfortable using:

```
? :set -rit
? 6*7
42
(3 reductions, 7 cells)
? it + it
84
(4 reductions, 11 cells)
?
```

Another reason for making this change might be that you have a program which uses the symbol `$$` as an operator. Each occurrence of the `$$` symbol in a script file will be interpreted as the correct operator, whatever the value of the repeat string. But, if the default `:set -r$$` setting is used, any occurrence of `$$` in an expression entered directly to the evaluator will be taken as a reference to the previous expression.

Note that the repeat string must be either a valid Haskell identifier or symbol, although it will always be parsed as an identifier. If the repeat string is set to a value which is neither an identifier or symbol (for example, `:set -r0`) then the repeat last expression facility will be disabled altogether.

2.2.4 Other changes

Comparing the list of command line options in Section 2.2 with the list produced by previous versions of Gofer will reveal some other small differences not already mentioned above. The changes are as follows:

- The default setting for the `d` toggle (show dictionaries in output expressions) has been changed to off (`:set -d`). For a lot of people, the appearance of dictionary values was rather confusing and of little use. If you still want to see how dictionary values are used, you will need to do `:set +d` or add the `+d` argument to your startup script.

- The default setting for the `e` toggle (warn about errors in literate scripts) has been changed to `:set +e` for closer compatibility with the literate script convention outline in the Haskell report, version 1.2. In addition, the setting of the `l` toggle is now used only as a default if no particular type of script file is specified by the file extension of a give script. See Section 2.4 below for further details.

- The default setting for the `f` toggle (terminate evaluation on first error) has been changed to `:set +f`. The old setting of `:set -f` is, in my opinion, better for debugging purposes, but does not give the behaviour that those using Haskell might expect. This has caused a certain amount of confusion and was the motivation for this change.

- The following three command line options, provided in previous versions of Gofer, have now been removed:

```
TOGGLES:
a    Use any evidence, not nec. best
E    Fail silently if evidence not found

OTHER OPTIONS:
xnum Set maximum depth for evidence search
```

  These options were only ever used for my own research and were (intentionally) undocumented, so it seemed sensible to remove them from the distributed system. A quick patch to the source code and a recompilation is all that is necessary to reinstate these options; useful if somebody out there found out about these options and actually uses them (if you do, I'd love to know why!).

2.3 Commands

The full list of commands that can be used from within the Gofer interpreter are summarized using the command `:?` as follows:

```
? :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load scripts from specified files
:load               clear all files except prelude
:also <filenames>   read additional script files
:reload             repeat last load command
:project <filename> use project file
:edit <filename>    edit file
:edit               edit last file
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
```

```
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:find <name>        edit file containing definition of name
:!command           shell escape
:cd dir             change directory
:quit               exit Gofer interpreter
?
```

Almost all of these commands are the same as in the previous release. The only new features are listed in the following sections.

2.3.1 Shell escapes

The shell escape command `:!` is used to enable you to run other programs from within the Gofer interpreter. For example, on a Unix system, you can print a list of all the files in the current directory by typing:

```
? :!ls
<list of all files in current directory gets printed here>
?
```

The same thing can be achieved on a PC running DOS by typing:

```
? :!dir
<list of all files in current directory gets printed here>
?
```

This is the same as in previous releases of Gofer; the only difference is that there is no longer any need to type a space between the `:!` command and the shell command that follows it. In fact, there is no longer any need to type the leading colon either. Thus the two commands above could equally well have been entered as:

```
!ls
!dir
```

To start a new shell from within Gofer, you can use the command `:!` or the abbreviated form `!` – in Unix and DOS you can return to the Gofer system by entering the shell command '`exit`'. This is likely to be different if you use Gofer on other systems.

2.3.2 Information about named values

The `:info` command is a new feature which is useful for obtaining information about the values currently loaded into a Gofer session. It can be used to display information about all kinds of different values including:

- Datatypes: The name of the datatype and a list of its associated constructor functions is printed:

  ```
  ? :info Request
  -- type constructor
  data Request
  ```

```
-- constructors:
ReadFile :: String -> Request
WriteFile :: String -> String -> Request
AppendFile :: String -> String -> Request
ReadChan :: String -> Request
AppendChan :: String -> String -> Request
Echo :: Bool -> Request
GetArgs :: Request
GetProgName :: Request
GetEnv :: String -> Request
```

- Type synonyms: Prints the name and expansion of the synonym:

```
? :info Dialogue
-- type constructor
type Dialogue = [Response] -> [Request]
```

  If the type synonym is restricted (see Section 3.1) then the expansion is not included in the output:

```
? :info Stack
-- type constructor
type Stack a = <restricted>
```

- Type classes: Lists the type class name, superclasses, member functions and instances:

```
? :info Eq
-- type class
class Eq a where
    (==) :: Eq a => a -> a -> Bool
    (/=) :: Eq a => a -> a -> Bool

-- instances:
instance Eq ()
instance Eq Int
instance Eq Float
instance Eq Char
instance Eq a => Eq [a]
instance (Eq a, Eq b) => Eq (a,b)
instance Eq Bool
```

  Note that the member functions listed for the class include the class predicate as part of the type; the output is not intended to be thought of as a syntactically valid class declaration.

  Overlapping instance declarations (see Section 3.2) are listed in increasing order of generality.

- Other values: for example, named functions and individual constructor and member functions:

```
? :info map : <=
map :: (a -> b) -> [a] -> [b]
```

```
(:) :: a -> [a] -> [a]   -- data constructor

(<=) :: Ord a => a -> a -> Bool   -- class member
```

As the last example shows, the `:info` command can take several arguments and prints out information about each in turn. A warning message is displayed if there are no known references to an argument:

```
? :info (:)
Unknown reference `(:)'
```

This illustrates that the arguments are treated as textual names for operators, not syntactic expressions (for example, identifiers). The type of the `(:)` operator can be obtained by giving the command `:info :` as above. There is no provision for including wildcard characters of any form in the arguments of :info commands.

If a particular argument can be interpreted as, for example, a constructor function, or a type constructor depending on context, both possibilities are displayed. For example, loading a program containing the definition:

```
data Set a = Set [a]
```

We obtain:

```
? :info Set
-- type constructor
data Set a

-- constructors:
Set :: [a] -> Set a

Set :: [a] -> Set a   -- data constructor
```

If no arguments are supplied to `:info`, a list of all the script files currently loaded into the interpreter will be displayed:

```
? :info

Gofer session for:
/usr/local/lib/Gofer/standard.prelude
examples
```

3 Literate scripts

Support for literate scripts – files in which program lines begin with a '>' character and all other lines are treated as comments – was provided in previous versions of Gofer. The command line option `:set +l` was used to force Gofer to treat each input file as a literate script, while `:set -l` (the default) was used to treat each input file as a standard script of definitions.

In practice, this turned out to be somewhat inconvenient, particularly when loading combinations of files, some as literate scripts, some without. For example, quite a few people kept two versions of the prelude, one as a literate script, one not, so that they wouldn't have to fiddle with the settings or using the :set commands to load files.

Gofer version 2.28 now uses a more sophisticated scheme to determine how an input script file should be treated, based on the use of file extensions. More specifically, any script file with a name ending in one of the following suffixes:

```
.hs     .has    .gs     .gof    .prelude
```

will always be loaded as a normal (i.e. non-literate) script file, regardless of the setting of the l command line option. In a similar way, files with names ending in one of the following suffixes:

```
.lgs    .lhs    .verb   .lit
```

will always be treated as literate scripts. The command line option l is only used for files with names not ending in one of the above suffixes.

For example, the commands:

```
:set -l
:load prog1.gs prog2 prog3.lgs
```

will load `prog1.gs` and `prog2` as non-literate scripts, and then load `prog3.lhs` as a literate script.

3.1 Prelude files

The Gofer system comes with a standard prelude, and a small number of alternative preludes. These have always been there, but a lot of people don't seem to have noticed these, so I thought I'd say a few words about the different preludes included with Gofer: Remember that you can always change the prelude you are using by setting the GOFER environment variable or by modifying a startup script as described in Section 2.1:

standard.prelude The standard Gofer prelude, using type classes and providing the familiar range of operators and functions.

nofloat.prelude A simplified version of the standard.prelude which does not include any floating point operators. This is likely to be of most use for those using Gofer on PCs where memory is at a premium; compiling a version of the interpreter (or compiler runtime library) without floating point support can give an important saving.

simple.prelude A prelude file based on the standard prelude but without type classes. Let me emphasize that point: *you can use Gofer without having to learn about type classes.* Some people seem to take to the use of type classes right from the beginning. For those that have problems understanding the technical details or even the motivation, the `simple.prelude` can be used to get you familiar with the syntax of the language and the basic principles. Then you can move up to the `standard.prelude` when you're ready. The principle differences can be described by listing the types of commonly used operators in the `simple.prelude`:

```
(==) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(<)  :: a -> a -> Bool
(>=) :: a -> a -> Bool
(>)  :: a -> a -> Bool
```

```
(/=) :: a -> a -> Bool
show :: a -> String
(+)  :: Int -> Int -> Int
(-)  :: Int -> Int -> Int
(*)  :: Int -> Int -> Int
(/)  :: Int -> Int -> Int
```

The resulting language is closer to the system in Bird and Wadler (and can be made closer still by editing the `simple.prelude` to use `zipwith` instead of `zipWith` etc.

cc.prelude An extended version of the `standard.prelude` including support for a number of useful constructor classes. Most of the examples and applications described in Section 4 are based on this prelude.

min.prelude A minimal prelude file. If you really want to build a very small prelude for a particular application, start with this and add the extra things that you need.

As you can see, the standard extension for prelude files is `.prelude` and any file ending with this suffix will be read as a non-literate script (as described in Section 2.4). Note that, even if you are using a computer where the full name of a prelude file is not stored (for example, on a DOS machine the `standard.prelude` file becomes `STANDARD.PRE`) you should still specify the prelude file by its full name to ensure that the Gofer system treats it correctly as a prelude file.

You are also free to construct your own prelude files, typically by modifying one of the supplied preludes described above. Anyone who created prelude files for use with previous releases of Gofer will need to edit these files to ensure that they will work correctly. Note in particular that there is no longer any need to include definitions of the I/O datatypes in programs. Furthermore, the error function should now be bound to the primitive `primError` rather than using the old definition of `error s | False = error s`.

4 Language differences

This section outlines a number of small differences and extensions to the language used by Gofer. These features are not included in the definition of Haskell, so you shouldn't be thinking that programs written using these features can ultimately be used with a full Haskell system. The use of constructor classes – a more substantial change is described in Section 4.

4.1 Restricted type synonyms

Gofer 2.28 supports a form of restricted type synonym that can be used to restrict the expansion of the synonym to a particular set of functions. Outside of the selected group of functions, the synonym constructor behaves like a standard datatype. More precisely, a restricted type synonym definition is a top level declaration of the form:

```
type T a1 ... am = rhs in f1, ..., fn
```

where `T` is the name of the restricted type synonym constructor and rhs is a type expression typically involving some of the (distinct) type variables `a1`, ..., `am`. The same kind of restrictions that apply to normal type synonym declarations are also

applied here. The major difference is that the expansion of the type synonym can only be used within the binding group of one of the functions `f1`, ..., `fn` (all of which must be defined by top-level definitions in the file containing the restricted type synonym definition). In the definition of any other function, the type constructor `T` is treated as if it had been introduced by a definition of the form:

```
data T a1 ... am = ...
```

The original motivation for restricted type synonyms came from my work with constructor classes as described in Section 4 and you will several examples of this in the `ccexamples.gs` file in the `demos/Ccexamples` directory of the standard distribution. For a simpler example, consider the following definition of a datatype of stacks in terms of the standard list type:

```
type Stack a = [a] in emptyStack, push, pop, top, isEmpty
```

The definitions for the five functions named here are as follows:

```
emptyStack :: Stack a
emptyStack  = []

push       :: a -> Stack a -> Stack a
push        = (:)

pop        :: Stack a -> Stack a
pop []      = error "pop: empty stack"
pop (_:xs)  = xs

top        :: Stack a -> a
top []      = error "top: empty stack"
top (x:_)   = x

isEmpty    :: Stack a -> Bool
isEmpty     = null
```

The type signatures here are particularly important. For example, since `emptyStack` is mentioned in the definition of the restricted type synonym Stack, the definition of `emptyStack` is type correct. The declared type for `emptyStack` is `Stack a` which can be expanded to `[a]`, agreeing with the type for the empty list `[]`. However, in an expression outside the binding group of these functions, the `Stack a` type is quite distinct from the `[a]` type:

```
? emptyStack ++ [1]
ERROR: Type error in application
*** expression     : emptyStack ++ [1]
*** term           : emptyStack
*** type           : Stack a
*** does not match : [Int]
```

The 'binding group' of a value refers to the set of values whose definitions are in the same mutually recursive group of bindings. In particular, this does not extend to the type class system so we can define instances such as:

```
instance Eq a => Eq (Stack a) where
    s1 == s2 | isEmpty s1 = isEmpty s2
             | isEmpty s2 = isEmpty s1
             | otherwise  = top s1 == top s2 && pop s1 == pop s2
```

As a convenience, Gofer allows the type signatures of functions mentioned in the type synonym declaration to be specified within the definition rather than in a different point in the script. Thus the example above could equally well have been written as:

```
type Stack a = [a] in
    emptyStack :: Stack a,
    push       :: a -> Stack a -> Stack a,
    pop        :: Stack a -> Stack a,
    top        :: Stack a -> a,
    isEmpty    :: Stack a -> Bool

emptyStack  = []

push        = (:)

pop []      = error "pop: empty stack"
pop (_:xs)  = xs

top []      = error "top: empty stack"
top (x:_)   = x

isEmpty     = null
```

However, the first form is necessary when you want to define two or more restricted type synonyms simultaneously. For example:

```
type Pointer = Int in allocate, deref, assign
type Heap a  = [a] in newHeap, allocate, deref, assign
newHeap  :: Heap a
allocate :: Heap a -> (Heap a, Pointer)
deref    :: Heap a -> Pointer -> a
assign   :: Heap a -> Pointer -> a -> Heap a
etc ...
```

The use of restricted type synonyms doesn't quite provide proper abstract data types. For example, if you try:

```
? push 1 emptyStack
[1]
(5 reductions, 11 cells)
```

then the structure of the stack as a list of values is revealed by the printing mechanism. This happens because Gofer uses the `show'` function to print out a value (in this case of type `Stack Int`) which looks inside the structure of the object to see how it is represented. This happens to be most convenient for use in an interpreter as an aid to debugging. For the purists (and the preservation of abstraction), Gofer could be modified to apply the (overloaded) show function to printed values. This would force the programmer to define the way in which stack values are printed (distinct from lists) and preserve the abstraction. Without having set up this machinery, we get:

```
? show (push 1 emptyStack)
ERROR: Cannot derive instance in expression
*** Expression        : show (push 1 emptyStack)
*** Required instance : Text (Stack Int)
```

The Gofer compiler described in Section 5 does not implement show' and hence enforces the abstraction.

4.2 Overlapping instance declarations

This section describes a somewhat technical extension, aimed at those who work with type classes. Many readers may prefer to skip to the next section at this point.

The definition of Haskell and previous versions of Gofer insist that no two instance declarations for a given class may contain overlapping predicates. Thus the declarations:

```
class CX a where c :: a -> Int

instance CX (a,Int) where c (x,y) = y
instance CX (Int,a) where c (x,y) = x
```

are not allowed because the two predicates overlap:

```
ERROR "misctest" (line 346): Overlapping instances for class "CX"
*** This instance   : CX (Int,a)
*** Overlaps with   : CX (a,Int)
*** Common instance : CX (Int,Int)
```

As the error message indicates, given an expression `c (1,2)` it is not clear whether we should use the first or the second instance declarations to evaluate this, with potentially different results, `2` or `1` respectively.

On the other hand, there are cases where this sort of thing might be quite reasonable. For example, the standard function show prints lists of characters as strings, but any other kind of list is printed using the `[...]` notation with the items separated by commas:

```
? show "Hello"
"Hello"
? show [True,False,True]
[True,False,True]
? show [1..10]
[1,2,3,4,5,6,7,8,9,10]
?
```

Haskell deals with this by an encoding using the showList function, but a more obvious approach might be to define two instances:

```
instance Text a => Text [a] where ... print using [ ... ] notation
instance Text [Char] where ...        print as string
```

Other examples might include providing optimized versions of primitives for particular frequently use operators, or providing a default behaviour as in:

```
class Eq a where (==) = error "no definition of equality specified"
```

Haskell requires the context of an overloaded function to be reduced to a form where the only predicates that it contains are of the form `C a`. This means that the inferred type of an object may be simplified before the full type of that object is known. For example, we might define a function:

```
f x = show [x,x]
```

The inferred type in Haskell is `f :: Text a => a -> String` and the decision about which of the two instance declarations above should be used has already been forced on us. To see this, note that `f 'a'` would evaluate to the string `"['a', 'a']"`. But if we allowed the second instance declaration above to be used, show `['a', 'a']` would evaluate to `"aa"`. This breaks a fundamental property of the language where we expect to be able to replace one subexpression with another equal term and obtain the same result.

In Gofer, the type system is a little different and the inferred type is `f :: Text [a] => a -> String`. The decision about which instance declaration to use is postponed until the type assigned to `'a'` is known. Thus both `f 'a'` and `show ['a', 'a']` evaluate to `"aa"` without any contradiction.

Although the type system in Gofer has always been able to support the use of certain overlapping instance declarations, previous versions of the system imposed stronger static restrictions which prohibited their use. Gofer 2.28 relaxes these restrictions by allowing a program to contain overlapping instance declarations so long as:

- One of the instance predicates being declared is a substitution instance of the other. Thus:

  ```
  instance Eq [Char] where ...    -- OK
  instance Eq a => Eq [a] where ...
  ```

  is permitted because the second predicate, `Eq [a]`, is more general than the first, `Eq [Char]`, which can be obtained by substituting Char for the type variable `a`. However, the example at the beginning of this section:

  ```
  instance CX (a,Int) where ...   -- ILLEGAL
  instance CX (Int,a) where ...
  ```

  is not allowed since neither `(a,Int)` or `(Int,a)` is a substitution instance of the other (even though they have a common instance `(Int,Int)`).

- The two instances declared are not identical. This rules out examples like:

  ```
  instance Eq Char where ...    -- ILLEGAL
  instance Eq Char where ...
  ```

The features described here are added principally for experimentation. I have some particular applications that I want to try out (which is why I actually implemented these ideas) but I would also be very interested to hear from anyone else that makes use of this extension.

4.3 Parsing Haskell syntax

From correspondence that I have received, quite a few people use Gofer to develop programs which, ultimately, will be compiled and executed using a Haskell system. Although the syntax of the two languages is quite similar, it has been necessary to comment out module headers and other constructs in Haskell programs before they could be used with previous version of Gofer.

The new version of the Gofer system is now able to parse these additional constructs (and will generate an error message if a syntax error occurs). However: *no attempt*

is made to interpret or use the information provided by these additional constructs. This feature is provided purely for the convenience of those people using Gofer and Haskell in the manner described above. Gofer does not currently support any notion of modules beyond the use of separate script files.

The following changes have been made:

- The identifiers:

```
deriving    default     module      interface
import      renaming    hiding      to
```

  are now reserved words in Gofer. Any program that uses one of these as an identifier with an older version of Gofer will need to be modified to use a different name instead.

- Module headers and import declarations may be included in a Gofer program using the syntax set out in version 1.2 of the Haskell report. Several modules may be included in a single file (but of course, Gofer makes no distinction between the sections of code appearing in different 'modules').

- Datatype definitions may include `deriving` clauses such as:

```
data Maybe a = Just a | Nothing deriving (Eq, Text)
```

  although no derived instances will actually be generated. If you need these facilities, you might consider writing out the instances of the type classes concerned yourself in a separate file which can be loaded when you run your program with Gofer, but which are omitted when you compile it with a proper Haskell system.

- Programs may include default declarations, although, once again, these are ignored; for example, there is no restriction on the forms of type that can be included in a default declaration, nor will an error occur if a single module includes multiple default declarations.

4.4 Local definitions in comprehensions

We all make mistakes. The syntax for Gofer currently permits a local definition to appear in a list comprehension (and indeed, in the monad comprehensions described in the next section):

```
[ (x,y) | x <- xs, y = f x, p y ]
```

This example is implemented by translating it to something equivalent to:

```
map h xs where h []     = []
               h (x:xs) = let y = f x
                          in  if p y then (x,y) : h xs
                                     else h xs
```

It is cumbersome to rewrite this using list comprehensions without local definitions:

```
concat [ let y = f x in [ (x,y) | p y ] | x <- xs ]
```

so we might resort to the 'hack' of writing:

```
[ y | x <- xs, y <- [f x], p y ]
```

which works (but doesn't extend to recursive bindings, and is really an inappropriate use for a list; a list is used to represent a sequence of zero or more objects, so using a list when you know that there is always going to be exactly one element seems unnecessary). So, to summarize, I still think that local definitions can be useful in comprehensions.

So where is the mistake I mentioned? The problem is with the *syntax*. First, it is rather easy to confuse the comprehension above with the comprehension:

```
[ (x,y) | x <- xs, y == f x, p y ],
```

leading to errors which are hard to detect. The second is that the syntax is too restrictive; you can only give relatively simple local declarations – mutually recursive definitions and function bindings are not permitted.

Gofer 2.28 now supports a new syntax for local definitions in comprehensions. The old syntax is still supported, for compatibility with previous releases, but will be deleted in the next public release (assuming I remember). Local declarations can now be included in a comprehension using a qualifier of the form `let { decls }`. So the comprehension at the beginning of this section can also be written:

```
[ (x,y) | x <- xs, let {y = f x}, p y ]
```

Note that the braces cannot usually be omitted in Gofer due to an undocumented extension to the syntax of Gofer function declarations. The braces would not be needed if this syntax were added to a standard Haskell system.

This extension means that it is now possible to write comprehensions such as:

```
[ (x,y,z) | x <- xs, let { y   = f x z;
                           z   = g x y;
                           f n = h n [] }, p x y z ]
```

Once again, this is still an experimental feature. I suspect it will be of most use to anyone making substantial use of monad comprehensions as described in the next section.

5 Constructor classes

[This is a long section; if you are not interested in experimenting with Gofer's system of constructor classes, you can skip straight ahead to the next section without missing anything. Of course, if you don't know what a constructor class is, you might want to read at least some of this section before you can make that decision.]

One of the biggest changes in Gofer version 2.28 is the provision of support for constructor classes. This section provides an overview of constructor classes which should hopefully, in conjunction with the example supplied with the full distribution, be enough to get you started. More technical details about constructor classes can be obtained by contacting me.

Some of the following introduction here (particularly sections 4.1 and 4.2) may seem somewhat familiar to those of you have already read one of the papers that I have written on the subject although I have added some more information about the Gofer implementation.

Others may find that this section of the documentation seems rather technical; try not to be put off at first sight. Looking through the examples and the documentation, you may find it is easier to understand than you expect!

A final comment before starting: there is, as yet, no strong consensus on the names and syntax that would be best for monad operations, comprehensions etc. If you have any opinions, or proposals which differ from what you see here, please let me know... I'd be very interested to hear other people's opinions on this.

5.1 An overloaded map function

Many functional programs use the `map` function to apply a function to each of the elements in a given list. The type and definition of this function as given in the Gofer standard prelude are as follows:

```
map          ::  (a -> b) -> ([a] -> [b])
map f []      =  []
map f (x:xs)  =  f x : map f xs
```

It is well known that the `map` function satisfies the familiar laws:

```
map id         = id
map f . map g  =  map (f . g)
```

A category theorist will recognize these observations as indicating that there is a functor from types to types whose object part maps any given type a to the list type `[a]` and whose arrow part maps each function `f::a -> b` to the function `map f :: [a] -> [b]`. A functional programmer will recognize that similar constructions are also used with a wide range of other data types, as illustrated by the following examples:

```
data Tree a  =  Leaf a  |  Tree a :^: Tree a

mapTree            :: (a -> b) -> (Tree a -> Tree b)
mapTree f (Leaf x)  = Leaf (f x)
mapTree f (l :^: r) = mapTree f l :^: mapTree f r

data Maybe a =  Just a  |  Nothing

mapMaybe           :: (a -> b) -> (Maybe a -> Maybe b)
mapMaybe f (Just x) = Just (f x)
mapMaybe f Nothing  = Nothing
```

Each of these functions has a similar type to that of the original `map` and also satisfies the functor laws given above. With this in mind, it seems a shame that we have to use different names for each of these variants. A more attractive solution would allow the use of a single name `map`, relying on the types of the objects involved to determine which particular version of the `map` function is required in a given situation. For example, it is clear that `map (1+) [1,2,3]` should be a list, calculated using the original `map` function on lists, while `map (1+) (Just 1)` should evaluate to `Just 2` using `mapMaybe`.

Unfortunately, in a language using standard Hindley/Milner type inference, there is no way to assign a type to the `map` function that would allow it to be used in this way. Furthermore, even if typing were not an issue, use of the map function would be rather limited unless some additional mechanism was provided to allow

the definition to be extended to include new datatypes perhaps distributed across a number of distinct program files.

5.1.1 An attempt to define map using type classes

The ability to use a single function symbol with an interpretation that depends on the type of its arguments is commonly known as overloading. In Gofer, overloading is implemented using type classes – which can be thought of as sets of types. For example, the `Eq` class defined by:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
```

(together with an appropriate set of instance declarations) is used to describe the set of types whose elements can be compared for equality. The standard prelude for Gofer includes integers, floating point numbers, characters, booleans, lists (in which the type of the members is also in `Eq`) and so forth. There is no need for all the definitions of equality to be combined in a single script file; new definitions of equality are typically included each time a new datatype is defined.

Functions such as `nub`, defined in the standard prelude as:

```
nub       :: Eq a => [a] -> [a]   -- remove duplicates from list
nub []     = []
nub (x:xs) = x : nub (filter (x/=) xs)
```

can be used with any choice of type for the type variable a so long as it is an instance of `Eq`. Only a single definition of the `nub` function is required.

Unfortunately, the system of type classes is not sufficiently powerful to give a satisfactory treatment for the map function; to do so would require a class `Map` and a type expression `m(t)` involving the type variable `t` such that `S = { m(t) | t is a member of Map }` includes (at least) the types:

```
{ (a -> b) -> ([a] -> [b]),
  (a -> b) -> (Tree a -> Tree b),
  (a -> b) -> (Maybe a -> Maybe b), ....
                                  | a and b arbitrary types }
```

The only possibility is to take `m(t) = t` and choose `Map` as the set of types `S` for which the map function is required:

```
class Map t where map :: t

instance Map ((a -> b) -> ([a] -> [b])) where ...
instance Map ((a -> b) -> (Tree a -> Tree b)) where ...
instance Map ((a -> b) -> (Maybe a -> Maybe b)) where ...
```

This syntax is permitted in Gofer (but not in Haskell) but it does not give a sufficiently accurate characterization of the type of map to be of much use. For example, the principal type of `\i j -> map j . map i` is:

```
(Map (a -> c -> e), Map (b -> e -> d)) => a -> b -> c -> d
```

(`a` and `b` are the types of `i` and `j` respectively). This is complicated and does not enforce the condition that `i` and `j` have function types. Furthermore, the

type is ambiguous (the type variable `e` does not appear to the right of the `=>` symbol or in the assumptions). Under these conditions, we cannot guarantee a well-defined semantics for this expression. Other attempts to define the map function, for example using multiple parameter type classes, have also failed for essentially the same reasons.

5.1.2 A solution using constructor classes

A much better approach is to notice that each of the types for which the map function is required is of the form:

```
(a -> b) -> (f a -> f b).
```

The variables a and b here represent arbitrary types while f ranges over the set of type constructors for which a suitable map function has been defined. In particular, we would expect to include the list constructor (which we write as `[]` in Gofer), `Tree` and `Maybe` as elements of this set. Motivated by our earlier comments we will call this set `Functor`. With only a small extension to the Gofer syntax for type classes this can be described by:

```
class Functor f where
    map :: (a -> b) -> (f a -> f b)

instance Functor [] where
    map f []        = []
    map f (x:xs)    = f x : map f xs

instance Functor Tree where
    map f (Leaf a)  = Leaf (f a)
    map f (l :^: r) = map f l :^: map f r

instance Functor Maybe where
    map f (Just x) = Just (f x)
    map f Nothing  = Nothing
```

`Functor` is our first example of a constructor class. The following extract illustrates how the definitions for `Functor` work in practice:

```
? map (1+) [1,2,3]
[2, 3, 4]
(15 reductions, 44 cells)
? map (1+) (Leaf 1 :^: Leaf 2)
Leaf 2 :^: Leaf 3
(10 reductions, 46 cells)
? map (1+) (Just 1)
Just 2
(4 reductions, 17 cells)
?
```

Furthermore, by specifying the type of map function more precisely, we avoid the ambiguity problems mentioned above. For example, the principal type of `\i j -> map j . map i` is simply:

```
Functor f => (a -> b) -> (b -> c) -> f a -> f c
```

which is not ambiguous, and makes the types of `i` and `j` as `(a -> b)` and `(b -> c)` respectively.

[You can try these examples yourself using the Gofer system. The first thing you need to do is start Gofer using the file `cc.prelude` instead of the usual Gofer `standard.prelude`. The `cc.prelude` includes the definition of the `functor` class and the instance for `Functor []`. The remaining two instance declarations are included (along with lots of other examples) in the file `ccexamples.gs` in the `demos/Ccexamples` subdirectory of the standard distribution.]

5.1.3 The kind system

Each instance of `Functor` can be thought of as a function from types to types. It would be nonsense to allow the type `Int` of integers to be an instance of `Functor`, since the type `(a -> b) ->(Int a -> Int b)` is obviously not well-formed. To avoid unwanted cases like this, we have to ensure that all of the elements in any given class are of the same kind.

To do this, we formalize the notion of kind, writing `*` for the kind of all types and `k1 -> k2` for the kind of a constructor which takes something of kind `k1` and returns something of kind `k2`. This notion comes is motivated by some theoretical work by Henk Barendregt on the subject of 'Generalized type systems'; do not confuse this with the use of the symbol `*` in a certain well-known functional language where it represents a type variable. These things are completely different!

Rather than thinking only of types we work with constructors which include types as a special case. Constructors take the form:

```
Constructor  ::=  ConstructorConstant
              |   Constructor1 Constructor2
              |   variable
```

This corresponds very closely to the way that most type expressions are already written in Gofer. For example, Tree a is an application of the constructor constant `Tree` to the variable `a`. Gofer has some special syntax for tuple, list and function types. The corresponding constructors can also be written directly in Gofer. For example:

```
a -> b   =  (->) a b
[a]      =  [] a
(a,b)    =  (,) a b
(a,b,c)  =  (,,) a b c
etc ...
```

Each constructor constant has a corresponding kind. For example:

```
Int, Float, ()    ::  *
[], Tree, Maybe   ::  * -> *
(->), (,)         ::  * -> * -> *
(,,)              ::  * -> * -> * -> *
```

Applying one constructor `C :: k1 -> k2` to a construct `C' :: k1` gives a constructor expression `C C'` with kind `k2`. Notice that this is just the same sort of thing you would expect from applying a function of type `a -> b` to an value of type `b`; kinds really are very much like 'types for constructors'.

Instead of checking that type expressions contain the correct number of arguments for each type constructor, we need to check that any type expression has kind `*`. In

a similar way, all of the elements of a constructor class must have the same kind; for example, a constructor class constraint of the form `Functor f` is only valid if `f` is a constructor expression of kind `* -> *`. Note also that our system includes Gofer/Haskell type classes as a special case; a type class is simply a constructor class for which each instance has kind `*`. Multiple parameter classes can also be dealt with in the same way, using a tuple of kinds `(k1,...,kn)` to indicate the kind of constructors required for each argument.

The language of constructors is essentially a system of combinators without any reduction rules. As such, standard techniques can be used to infer the kinds of constructor variables, constructor constants introduced by new datatype definitions and the kind of the elements held in any particular constructor class. The important point is that there is no need – and indeed, in our current implementation, no opportunity – for the programmer to supply kind information explicitly. We regard this as a significant advantage since it means that the programmer can avoid much of the complexity that might otherwise result from the need to annotate type expressions with kinds.

5.2 Monads as an application of constructor classes

Motivated by the work of Moggi and Spivey, Wadler has proposed a style of functional programming based on the use of monads. While the theory of monads had already been widely studied in the context of abstract category theory, Wadler introduced the idea that monads could be used as a practical method for modeling so-called 'impure' features in a purely functional programming language.

The examples in this and following sections illustrate that the use of constructor classes can be particularly convenient for programming in this style. You will also find a lot more examples prepared for use with Gofer in the file `ccexamples` in the `demos/Ccexamples` subdirectory of the standard distribution.

5.2.1 A framework for programming with monads

The basic motivation for the use of monads is the need to distinguish between computations and the values that they produce. If `m` is a monad then an object of type `(m a)` represents a computation which is expected to produce a value of type `a`. These types reflect the fact that the use of particular programming language features in a given calculation is a property of the computation itself and not of the result that it produces.

Taking the approach outlined by Wadler in his paper 'The Essence of Functional Programming' (POPL '92), we introduce a constructor class of monads using the definition:

```
class Functor m => Monad m where
    result    :: a -> m a
    join      :: m (m a) -> m a
    bind      :: m a -> (a -> m b) -> m b

    join x     = bind x id
    x `bind` f = join (map f x)
```

The expression `Functor m => Monad m` defines `Monad` as a subclass of `Functor` ensuring that, for any given monad, there will also be a corresponding instance of the overloaded `map` function. The use of a hierarchy of classes enables us to capture

the fact that not every instance of Functor can be treated as an instance of `Monad` in any natural way.

[If you are familiar with either my previous papers or Wadler's writings on the use of monads, you might notice that the declaration above uses the name 'result' in place of 'return' or 'unit' that have been previously used for the same thing. The latter two choices have been used elsewhere for rather different purposes, and there is currently no clear picture of which names should be used. The identifier 'result' is the latest in a long line of attempts to find a name which both conveys the appropriate meaning and is not already in use for other applications.]

By including default definitions for bind and join we only need to give a definition for one of these (in addition to a definition for result) to completely define an instance of `Monad`. This is often quite convenient. On the other hand, it would be an error to omit definitions for both operators since the default definitions are clearly circular. We should also mention that the member functions in an instance of `Monad` are expected to satisfy a number of laws which are not reflected in the class definition above.

The following declaration defines the standard monad structure for the list constructor `[]` which can be used to describe computations producing multiple results, corresponding to a simple form of non-determinism:

```
instance Monad [] where
    result x        = [x]
    []     `bind` f = []
    (x:xs) `bind` f = f x ++ (xs `bind` f)
```

As a second example, the monad structure for the `Maybe` datatype, which might be used to describe computations which fail to produce any value at all if an error condition occurs, can be described by:

```
instance Monad Maybe where
    result x         = Just x
    Just x  `bind` f = f x
    Nothing `bind` f = Nothing
```

Another interesting use of monads is to model programs that make use of an internal state. Computations of this kind can be represented by functions of type `s-> (a,s)` (often referred to as state transformers) mapping an initial state to a pair containing the result and final state. In order to get this into the appropriate form for the Gofer system of constructor classes, we introduce a new datatype:

```
data State s a = ST (s -> (a,s))
```

The functor and monad structures for state transformers are as follows:

```
instance Functor (State s) where
    map f (ST st) = ST (\s -> let (x,s') = st s in (f x, s'))

instance Monad (State s) where
    result x      = ST (\s -> (x,s))
    ST m `bind` f = ST (\s -> let (x,s') = m s
                                  ST f'  = f x
                              in  f' s')
```

Notice that the `State` constructor has kind `* -> * -> *` and that the declarations above define `State s` as a monad and functor for any state type `s` (and hence

`State s` has kind `* -> *` as required for an instance of these classes). There is no need to assume a fixed state type.

¿From a user's point of view, the most interesting properties of a monad are described, not by the `result`, `bind` and `join` operators, but by the additional operations that it supports. The following examples are often useful when working with state monads. The first can be used to 'run' a program given an initial state and discarding the final state, while the second might be used to implement an integer counter in a `State Int` monad:

```
startingWith          :: State s a -> s -> a
ST m `startingWith` s0 = result where (result,_) = m s0
incr :: State Int Int
incr  = ST (\s -> (s,s+1))
```

To illustrate the use of state monads, consider the task of labeling each of the nodes in a binary tree with distinct integer values. One simple definition is:

```
label     :: Tree a -> Tree (a,Int)
label tree = fst (lab tree 0)
 where lab (Leaf n)  c  =  (Leaf (n,c), c+1)
       lab (l :^: r) c  =  (l' :^: r', c'')
                           where (l',c')  = lab l c
                                 (r',c'') = lab r c'
```

This uses an explicit counter (represented by the second parameter to `lab`) and great care must be taken to ensure that the appropriate counter value is used in each part of the program; simple errors, such as writing `c` in place of `c'` in the last line, are easily made but can be hard to detect.

An alternative definition, using a state monad and following the layout suggested in Wadler's POPL paper, can be written as follows:

```
label     :: Tree a -> Tree (a,Int)
label tree = lab tree `startingWith` 0
 where lab (Leaf n)  = incr                  `bind` \c ->
                       result (Leaf (n,c))
       lab (l :^: r) = lab l                 `bind` \l' ->
                       lab r                 `bind` \r' ->
                       result (l' :^: r')
```

While this program is perhaps a little longer than the previous version, the use of monad operations ensures that the correct counter value is passed from one part of the program to the next. There is no need to mention explicitly that a state monad is required: The use of `startingWith` and the initial value `0` (or indeed, the use of `incr` on its own) are sufficient to determine the monad `State Int` needed for the bind and result operators. It is not necessary to distinguish between different versions of the monad operators bind, result and join or to rely on explicit type declarations.

5.2.2 Monad comprehensions

Several functional programming languages provide support for list comprehensions, enabling some common forms of computation with lists to be written in a concise form resembling the standard syntax for set comprehensions in mathematics. In his paper 'Comprehending Monads' (ACM Lisp and Functional Programming, 1990),

Wadler made the observation that the comprehension notation can be generalized to arbitrary monads, of which the list constructor is just one special case.

In Wadler's notation, a monad comprehension is written using the syntax of a list comprehension but with a superscript to indicate the monad in which the comprehension is to be interpreted. This is a little awkward and makes the notation less powerful than might be hoped since each comprehension is restricted to a particular monad. Using the overloaded operators described in the previous section, Gofer provides a more flexible form of monad comprehension which relies on overloading rather than superscripts. At the time of writing, this is the only concrete implementation of monad comprehensions known to us.

In our system, a monad comprehension is an expression of the form `[e | qs ]` where `e` is an expression and `gs` is a list of generators of the form `p <- exp`. As a special case, if `gs` is empty then the comprehension `[ e | qs ]` is written as `[ e ]`. The implementation of monad comprehensions is based on the following translation of the comprehension notation in terms of the result and bind operators described in the previous section:

```
[ e ]                 = result e
[ e | p <- exp, qs ] = exp `bind` \p -> [ e | qs ]
```

In this notation, the label function from the previous section can be rewritten as:

```
label     :: Tree a -> Tree (a,Int)
label tree = lab tree `startingWith` 0
 where lab (Leaf n)  = [ Leaf (n,c) | c <- incr ]
       lab (l :^: r) = [  l :^: r   | l <- lab l, r <- lab r ]
```

Applying the translation rules for monad comprehensions to this definition yields the previous definition in terms of result and bind. The principal advantage of the comprehension syntax is that it is often more concise and, in the author's opinion, sometimes more attractive.

5.2.3 Monads with a zero

Assuming that you are familiar with Gofer's list comprehensions, you will know that it is also possible to include boolean guards in addition to generators in the definition of a list comprehension. Once again, Wadler showed that this was also possible in the more general setting of monad comprehensions, so long as we restrict such comprehensions to monads that include a special element zero satisfying a small number of laws. This can be dealt with in our framework by defining a subclass of `Monad`:

```
class Monad m => Monad0 m where
    zero   :: m a
```

For example, the `List` monad has the empty list as a zero element:

```
instance Monad0 [] where zero = []
```

Note that not there are also some monads which do not have a zero element and hence cannot be defined as instances of `Monad0`. The `State s` monads described in Section 4.2.1 are a simple example of this.

Working in a monad with a zero, a comprehension involving a boolean guard can be implemented using the translation:

```
[ e | guard, qs ]  =  if guard then [ e | qs ] else zero
```

Notice that, as far as the type system is concerned, the use of zero in the translation of a comprehension involving a guard automatically captures the restriction to monads with a zero:

```
? :t \x p -> [ x | p x ]
\x p -> [ x | p x ] :: Monad0 b => a -> (a -> Bool) -> b a
?
```

The inclusion of a zero element also allows a slightly different translation for generators in comprehensions:

```
[ e | p <- exp, qs ]  =  exp `bind` f
                         where  f p  =  [ e | qs ]
                                f _  =  zero
```

This corresponds directly to the semantics of standard Gofer list comprehensions, but only differs from the semantics of the translation given in the previous section when `p` is an irrefutable pattern; i.e. when `p` is a pattern which may not match the value (or values) generated by `exp`. You can see the difference by trying the following example in Gofer:

```
? [ x | [x] <- [[1],[],[2]]]
[1, 2]
(9 reductions, 31 cells)
? map (\[x] -> x) [[1],[],[2]]
[1,
Program error: {v157 []}
(8 reductions, 66 cells)
```

In order to retain compatibility with the standard list comprehension notation, Gofer always uses the second translation above for generators if the pattern `p` is refutable. This may sometimes give inferred types which are more restrictive than you expect. For example, tuples are not irrefutable patterns in Gofer or Haskell, and so the function:

```
? :t \xs -> [ x | (x,y) <- xs ]
\xs -> [ x | (x,y)<-xs ] :: Monad0 a => a (b,c) -> a b
?
```

is restricted to monads with a zero because the expanded translation above is used. You can always avoid this problem by using the lazy pattern construct (i.e. the tilde operator, `~p`) as in:

```
? :t \xs -> [ x | ~(x,y) <- xs ]
\xs -> [ x | ~(x,y)<-xs ] :: Monad a => a (b,c) -> a b
?
```

[At one stage, I was using a different form of brackets to represent monad comprehensions, implemented using the original translation to avoid changing the semantics of list comprehensions. But I finally decided that it would be better to use standard comprehension notation with lazy pattern annotations where necessary since this is less cumbersome than writing `\xs -> [| x | (x,y) <- xs |]` in place of the comprehension above. Please let me know what you think!]

5.2.4 Generic operations on monads

The combination of polymorphism and constructor classes in our system makes it possible to define generic functions which can be used on a wide range of different monads. A simple example of this is the 'Kleisli composition' for an arbitrary monad, similar to the usual composition of functions except that it also takes care of 'side effects'. The general definition is as follows:

```
(@@)   :: Monad m => (a -> m b) -> (c -> m a) -> (c -> m b)
f @@ g  = join . map f . g
```

For example, in a monad of the form `State s`, the expression `f @@ g` denotes a state transformer in which the final state of the computation associated with `g` is used as the initial state for the computation associated with `f`. More precisely, for this particular kind of monad, the general definition given above is equivalent to:

```
(@@)  :: (b -> State s c) -> (a -> State s b) -> (a -> State s c)
f @@ g = \a -> STM (\s0 -> let ST g'  = g a
                               (b,s1) = g' s0
                               ST f'  = f b
                               (c,s2) = f' s1
                           in  (c,s2))
```

The biggest advantage of the generic definition is that there is no need to construct new definitions of `(@@)` for every different monad. On the other hand, if specific definitions were required for some instances, perhaps in the interests of efficiency, we could simply include `(@@)` as a member function of `Monad` and use the generic definition as a default implementation.

Generic operations can also be defined using the comprehension notation:

```
mapl             :: Monad m => (a -> m b) -> ([a] -> m [b])
mapl f []         = [ [] ]
mapl f (x:xs)     = [ y:ys | y <- f x, ys <- mapl f xs ]
```

This is the same as mapping a function down the elements of a list using the normal map function except that, in the presence of side effects, the order in which the applications are carried out is important. For `mapl`, we start on the left (i.e. the front of the list) and work towards the right. There is a corresponding dual which works in the reverse direction:

```
mapr             :: Monad m => (a -> m b) -> ([a] -> m [b])
mapr f []         = [ [] ]
mapr f (x:xs)     = [ y:ys | ys <- mapr f xs, y <- f x ]
```

These general functions have applications in several kinds of monad with examples involving state and output.

The comprehension notation can also be used to define a generalization of Haskell's filter function which works in an arbitrary monad with a zero:

```
filter           :: Monad0 m => (a -> Bool) -> m a -> m a
filter p xs       = [ x | x<-xs, p x ]
```

There are many other general purpose functions that can be defined in the current framework and used in arbitrary monads. To give you some further examples, here are generalized versions of the foldl and foldr functions which work in an arbitrary monad:

```
mfoldl           :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
mfoldl f a []     = result a
mfoldl f a (x:xs) = f a x `bind` (\fax -> mfoldl f fax xs)

mfoldr           :: Monad m => (a -> b -> m b) -> b -> [a] -> m b
mfoldr f a []     = result a
mfoldr f a (x:xs) = mfoldr f a xs `bind` (\y -> f x y)
```

[Generalizing these definitions (and those of `mapl`, `mapr`) to work with a second arbitrary monad (in place of the list monad) is left as an entertaining exercise for the reader :–)]

As a final example, here is a definition of a 'while' loop for an arbitrary monad:

```
while    :: Monad m => m Bool -> m b -> m ()
while c s = c  `bind` \b ->
            if b then s         `bind` \x ->
                      while c s
                 else result ()
```

5.2.5 A family of state monads

We have already described the use of monads to model programs with state using the `State` datatype in Section 4.2.1. The essential property of any such monad is the ability to update the state and we might therefore consider a more general class of state monads given by:

```
class Monad (m s) => StateMonad m s where
    update :: (s -> s) -> m s s
    set    :: s -> m s s
    fetch  :: m s s
    set new = update (\old -> new)
    fetch   = update id
```

An expression of the form `update f` denotes the computation which updates the state using `f` and result the old state as its result. For example, the `incr` function described above can be defined as:

```
incr :: StateMonad m Int => m Int Int
incr  = update (1+)
```

in this more general setting. The class declaration above also includes set and fetch functions which set the state to a particular value or return its value. These are easily defined in terms of the update function as illustrated by the default definitions.

The `StateMonad` class has two parameters; the first should be a constructor of kind `(* -> * -> *)` while the second gives the state type (of kind `*`); both are needed to specify the type of update. The implementation of update for a monad of the form `State s` is straightforward and provides us with our first instance of the `StateMonad` class:

```
instance StateMonad State s where
    update f = ST (\s -> (s, f s))
```

A rather more interesting family of state monads can be described using the following datatype definition:

```
data STM m s a = STM (s -> m (a,s)) -- a more sophisticated example,
                                    -- where the state monad is
                                    -- parameterized by a second,
                                    -- arbitrary monad.
```

Note that the first parameter to `StateM` has kind `(* -> *)`, a significant extension from Haskell (and previous versions of Gofer) where all of the arguments to a type constructor must be types. This is another benefit of the kind system.

The functor and monad structure of a `StateM m s` constructor are given by:

```
instance Monad m => Functor (STM m s) where
    map f (STM xs) = STM (\s -> [ (f x, s') | ~(x,s') <- xs s ])

instance Monad m => Monad (STM m s) where
    result x        = STM (\s -> result (x,s))
    STM xs `bind` f = STM (\s -> xs s `bind` (\(x,s') ->
                                 let STM f' = f x
                                 in  f' s'))
```

Note the condition that `m` is an instance of `Monad` in each of these definitions. If we hadn't used the lazy pattern construct `~(x,s')` in the instance of `Functor`, it would have been necessary to strengthen this further to instances of `Monad0` – i.e. monads with a zero.

The definition of `StateM m` as an instance of `StateMonad` is also straightforward:

```
instance StateMonad (STM m) s where
    update f = STM (\s -> result (s, f s))
```

The following two functions are also useful for work with `STM m s` monads. The first, protect, allows an arbitrary computation to be embedded in a state based computation without access to the state. The second, execute, is similar to the `startingWith` function in Section 4.2.1, running a state based computation with a given initial state and returning a computation as the result.

```
protect          :: Monad m => m a -> STM m s a
protect m         = STM (\s -> [ (x,s) | x<-m ])

execute          :: Monad m => s -> STM m s a -> m a
execute s (STM f) = [ x | ~(x,s') <- f s ]
```

Support for monads like `StateM m s` seems to be an important step towards solving the problem of constructing monads by combining features from simpler monads, in this case combining the use of state with the features of an arbitrary monad `m`. I hope that the system of constructor classes in Gofer will be a useful tool for people working in this area.

5.2.6 Monads and substitution

The previous sections have concentrated on the use of monads to describe computations. Monads also have a useful interpretation as a general approach to substitution. This in turn provides another application for constructor classes.

Taking a fairly general approach, a substitution can be considered as a function `s::v-> t w` where the types `v` and `w` represent sets of variables and the type `t a` represents a set of terms, typically involving elements of type `a`. If `t` is a monad

and `x::t v`, then `x 'bind' s` gives the result of applying the substitution `s` to the term `x` by replacing each occurrence of a variable `v` in `x` with the corresponding term `s v` in the result. For example:

```
instance Monad Tree where
    result             = Leaf
    Leaf x    'bind' f = f x
    (l :^: r) 'bind' f = (l 'bind' f) :^: (r 'bind' f)
```

With this interpretation in mind, the Kleisli composition `(@@)` in Section 4.2.4 is just the standard way of composing substitutions, while the result function corresponds to a null substitution. The fact that `(@@)` is associative with result as both a left and right identity follows from the standard algebraic properties of a monad.

5.3 Constructor classes in Gofer

The previous two sections should have given you some ideas about the motivation and use for constructor classes. It remains to say a few words about the way that constructor classes fit into the general Gofer framework. In practice, this means giving a more detailed description of the way that the kind system works.

5.3.1 Kind errors and the k command line option

As has already been mentioned, Gofer 2.28 uses kind information to check that type expressions are well-formed rather than simply checking that each type constructor is applied to an appropriate number of arguments. For example, having defined a tree datatype:

```
data Tree a = Leaf a | Tree a :^: Tree a
```

the following definition will be rejected as an error:

```
type Example  =  Tree Int Bool
```

as follows:

```
ERROR "file" (line 42): Illegal type "Tree Int Bool" in
                        constructor application
```

The problem here is that the `Tree` constructor has kind `* -> *` so that it expects to take one argument (a type) and deliver a type as the result. On the other hand, in the definition of Example, the `Tree` constructor is treated as having (at least) two arguments; i.e. as having a kind of the form `(* -> * -> k)` for some kind `k`. Rather than confuse a user who is not familiar with the use of kinds, Gofer normally just prints an error message like the one above for examples like this.

If you would like Gofer to give a more detailed description of the problem, you can use the `:set +k` command line option as follows:

```
? :set +k
? :r
Reading script file "file":

ERROR "file" (line 42): Kind error in constructor application
*** expression     : Tree Int Bool
```

```
*** constructor    : Tree
*** kind           : * -> *
*** does not match : * -> a -> b
```

When the `k` command line option has been selected, the `:info` command described in Section 2.3.2 also includes kind information about the kinds of type constructors defined in a program. For example, given the definition of `Tree` above and the datatypes:

```
data STM m s x = STM (s -> m (s, x))
data Queue a   = Empty | a :< Queue a | Queue a :> a
```

The `:info` command gives the following kinds (editing the output to remove details about constructor functions for each datatype):

```
? :info Tree STM Queue
-- type constructor with kind * -> *
data Tree a
-- type constructor with kind (* -> *) -> * -> * -> *
data STM a b c

-- type constructor with kind * -> *
data Queue a
```

In addition to calculating a kind of each type constructor introduced in a datatype declaration, Gofer also determines a kind for each constructor defined by means of a type synonym. For example, the following definitions:

```
type Subst m v = v -> m v
type Compose f g x = f (g x)
type Pointer a = Int
type Apply f x = f x
type Fusion f g x = f x (g x)
type Const x y   = x
```

are treated as having kinds:

```
? :info Subst Compose Pointer Apply Fusion Const
-- type constructor with kind (* -> *) -> * -> *
type Subst a b = b -> a b

-- type constructor with kind (* -> *) -> (* -> *) -> * -> *
type Compose a b c = a (b c)

-- type constructor with kind * -> *
type Pointer a = Int

-- type constructor with kind (* -> *) -> * -> *
type Apply a b = a b

-- type constructor with kind (* -> * -> *) -> (* -> *) -> * -> *
type Fusion a b c = a c (b c)

-- type constructor with kind * -> * -> *
type Const a b = a
```

Note however type synonyms are only used as abbreviations for other type expressions. It is not permitted to use a type synonym constructor in a type expression without giving the correct number of arguments.

```
? undefined :: Const Int
ERROR: Wrong number of arguments for type synonym "Const"
```

Assuming that you are familiar with polymorphic functions in Gofer, you might be wondering why some of the kinds given for the type synonyms above are not also polymorphic in some sense. After all, the standard prelude function const, is defined by

```
const x y = x
```

with type `a -> b -> a`, which looks very similar to the definition of the `Const` type synonym above, except that the kinds of the two arguments have both been fixed as `*`. In fact, the right hand side of a type synonym declaration is always required to have kind `*`, so this would mean that the most general kind that could be assigned to the `Const` constructor would be `* -> a -> *`.

Gofer does not currently support the use of polymorphic kinds (let's call them polykinds from now on). First of all, it is not clear what practical applications polykinds might offer (I have yet to find an example where they are useful). Furthermore, some of the deeper theoretical issues about type inference and related topics have not yet been studied and I suspect that polykinds would introduce significant complications without any significant benefits.

The current approach is to replace any unknown part of an inferred kind with the kind `*`. Any polymorphism in the kind of a constructor corresponds much more closely to the idea of a value that is not actually used at all than in the language of normal expressions and their types so this is unlikely to cause any problems. And of course, in Haskell and previous versions of Gofer, any variable used in a type expression was assumed to be a type variable with kind `*`, so all of the kinds above are consistent with this interpretation.

The rest of this section is likely to get a bit hairy. Read on at your peril, or skip to the start of Section 4.3.2. Only those with a strong interest in the type theory and pragmatics of constructor classes will miss anything.

The same approach is used to determine the kinds of constructor variables in type expressions. In theory, this can sometimes lead to problems. In practice, this only happens in very contrived examples and I doubt that any problems will occur for serious applications. The following example illustrates the kind of 'problem' that can occur. Suppose that we use a script containing the definitions:

```
undefined :: a              -- the 'bottom' value
undefined  = undefined

strange   :: f Tree -> f a
strange    = undefined
```

The type signature for the 'strange' function is indeed very strange; the constructor variables `f` and `a` have kinds `(* -> *) -> *` and `(* -> *)` respectively. What's more, the type is very restrictive. Without including additional primitive constructs in the language, I very much doubt that you will be able to find an alternative definition for strange which is not semantically equivalent to the definition above. And of course, the definition above doesn't really have any practical applications anyway. [In case you don't get my point, I'm trying to show that this really is a

very contrived example.] I would be very surprised to see a genuine example of a polymorphic operator which involves constructor variables of higher kinds in a non-trivial way that does not also include overloading constraints as part of the type. For example, it is not at all difficult to think of an interesting value of type `Monad m => a -> m a`, but much harder to think of something with type `a -> m a` (remember this means for all `a` and for all `m`).

The definitions of undefined and strange above will be accepted by the Gofer system as will the following definition:

```
contrived = strange undefined
```

The type of contrived will now be `f a` where `f :: (* -> *) -> *` and `a :: (* -> *)`. However, if we modify the definition of contrived to include a type signature:

```
contrived :: f a
contrived  = strange undefined
```

then we get a type checking error:

```
? :l file
Reading script file "file":
Type checking
ERROR "file" (line 24): Type error in function binding
*** term           : contrived
*** type           : a b
*** does not match : c d
*** because        : constructor variable kinds do not match
```

The problem is that for the declared type signature, the variables `f` and `a` are treated as having kinds `(* -> *)` and `*` respectively. These do not agree with the real kinds for these variables.

To summarize, what this all means is that it is possible to define values whose principal types cannot be expressed within the language of Gofer types in the current implementation. The values defined can actually be used within a program, but it would not, for example, be possible to allow such values to be exported from a module in a Haskell system unless kind annotations were added to the inferred types.

5.3.2 The kind of values in a constructor class

The previous section indicated that, if the `:set +k` command line option has been set, the `:info` command will include information about the kinds of type constructor constants in its output. This will also cause the `:info` command to display information about the kinds of classes and constructor classes. Notice for example in the following how the output distinguishes between `Eq`, a type class, and `Functor`, a constructor class in which each instance has kind `(* -> *)`:

```
? :info Eq Functor
-- type class
class Eq a where
    (==) :: Eq a => a -> a -> Bool
    (/=) :: Eq a => a -> a -> Bool

-- instances:
```

```
instance Eq ()
...

-- constructor class with arity (* -> *)
class Functor a where
    map :: Functor a => (b -> c) -> a b -> a c

-- instances:
instance Functor []
...
```

5.3.3 Implementation of list comprehensions

The implementation of overloaded monad comprehensions is cute, but also has a couple of potential disadvantages. These are discussed in this section. As you will see, they really aren't very much to worry about.

First of all, the decision to overload the notation for singleton lists so that `[ exp ] == result exp` can sometimes cause a few surprises:

```
? map (1+) [1]
ERROR: Unresolved overloading
*** type        : Monad a => a Int
*** translation : map (1 +) [ 1 ]
```

Note that this will only occur if you are actually using a prelude which includes the definition of the `Monad` class given in Section 4.2 This can be solved using the command line toggle `:set -1` which forces any expression of the form `[ exp ]` to be treated as a singleton list rather than being interpreted in an arbitrary monad. You really have to write 'result' if you do want an arbitrary monad:

```
? :set -1
? map (1+) [1]
[2]
(7 reductions, 18 cells)
? map (1+) (result 1)
ERROR: Unresolved overloading
*** type        : Monad a => a Int
*** translation : map (1 +) (result 1)
```

This should probably be the default setting, but I have left things as they are for the time being, partly so that other people might get the chance to find out about this and decide what setting they think would be best. As usual, the default setting can be recovered using the `:set +1` command.

A second concern is that the implementation of list comprehensions may be less efficient in the presence of monad comprehensions. Gofer usually uses Wadler's 'optimal' translation for list comprehensions as described in Simon Peyton Jones book. In fact, this translation will always be used if either the prelude being used does not include the standard Monad class or the type system is able to guarantee that a given monad comprehension is actually a list comprehension.

If you use a prelude containing the `Monad` class, you may notice some small differences in performance in examples such as:

```
? [ x * x | x <- [1..10] ]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
(98 reductions, 203 cells)

? f [1..10] where f xs = [ x * x | x <- xs ]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
(139 reductions, 268 cells)
```

The second expression is a little more expensive since the local definition of f is polymorphic with `f :: (Num b, Monad a) => a b -> a b` and hence the implementation of the comprehension in `f` does not use the standard translation for lists. To be honest, the difference between these two functions really isn't anything to worry about in the context of an interpreter like Gofer. And of course, if you really want to avoid this problem, an explicit type signature will do the trick (as in other cases where overloading is involved):

```
? f [1..10] where f   :: Num b => [b] -> [b];
                  f xs = [ x * x | x <- xs ]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
(99 reductions, 205 cells)

? f [1..10] where f   :: [Int] -> [Int]
                  f xs = [ x * x | x <- xs ]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
(99 reductions, 203 cells)
```

As the last example shows, there is only one more reduction in this case and that is the reduction step that deals with the application of `f` to the argument list `[1..10]`.

6 GOFC, the Gofer compiler

This release of Gofer includes `gofc`, a 'compiler' for Gofer programs which translates a large class of Gofer programs into C code which can then be compiled and executed as a standalone application.

Before anybody gets too excited, there are a couple of points which I should mention straight away:

- To make use of `gofc`, you will need a C compiler. This is why I do not intend to distribute any binary versions of `gofc`; if you have the C compiler needed to compile the output of `gofc` then you should also be able to compile `gofc` from the sources.
- First of all, the Gofer compiler was written by modifying the Gofer interpreter. Most of the modifications and changes were made in just a few days. The compiler and interpreter still share a large proportion of code. As such, and in case it isn't obvious: *please do not* expect to gain the same kind of performance out of `gofc` as you would from one of the serious Haskell projects. A considerably greater amount of time and effort has gone into those systems.
- The compiler is actually over a year old, but this is the first time it has been released. Although I have worked with it a bit myself, it hasn't had half the amount of testing that Gofer user's have given the interpreter over the last year and a half. It may not be as reliable as the interpreter. If you have problems with a compiled program, try running it with the interpreter too just to check that you haven't found a potential bug in `gofc`.

That having been said, I hope that the Gofer compiler will be useful to many Gofer users. One possible advantage is that the executables may be smaller than with some other systems. And of course, the fact that gofc runs on some home computers may also be useful. Finally, gofc provides a simplified system for experimenting with the runtime details of an implementation. For example, the source code for the runtime system is set up in such a way as to make it possible to experiment with alternative garbage collection schemes.

6.1 Using gofc

Compiling a program with gofc is very much like starting up the Gofer interpreter. The compiler starts by reading the prelude and then loads the script files specified by the command line. These scripts must contain a definition for the value `main :: Dialogue` which will be the dialogue expression that is evaluated when the compiled program is executed.

For example, if the file `apr1.gs` contains the simple program:

```
main :: Dialogue
main  = appendChan "stdout" "Hello, world\n" exit done
```

then this can be compiled as:

```
machine% gofc apr1.gs
Gofer->C Version 1.01 (2.28)  Copyright (c) Mark P Jones 1992-1993

Reading script file "/usr/local/lib/Gofer/standard.prelude":
Reading script file "apr1.gs":

Writing C output file "apr1.c":
[Leaving Gofer->C]
machine%
```

The output is written to the file `apr1.c` – i.e. the name obtained by removing the `.gs` suffix and replacing it with a `.c` suffix. Other filename suffixes that are treated in a similar way are:

`.prj`	
`.gp`	for Gofer project files
`.prelude`	for Gofer prelude files
`.gof`	
`.gs`	for Gofer scripts
`.has`	
`.hs`	for Haskell scripts
`.lhs`	
`.lit`	
`.lgs`	
`.verb`	for literate scripts

If no recognized suffix is found then the name of the output file is obtained simply by appending the `.c` suffix to the input name.

For the benefit of those using Unix systems, let me point out that this could cause you problems if you are not careful; if you take an input file called '`prog`' and compile it to '`prog.c`' using `gofc`, make sure that you do not compile the C program in such a way that the output is also called '`prog`' since this will overwrite your original

source code! For this reason, I would always suggest using file extensions such as the `.gs` example above if you are using `gofc`.

If you run gofc with multiple script files, then the name of the output file is based on the last script file to be loaded. For example, the command '`gofc prog1.gs prog2.gs`' produces an output file '`prog2.c`'.

Gofc also works with project files, using the name of the project file to determine the name of the output file. For example, the `miniProlog` interpreter can be compiled using:

```
machine% gofc + miniProlog
Gofer->C Version 1.01 (2.28)  Copyright (c) Mark P Jones 1992-1993

Reading script file "/usr/local/lib/Gofer/standard.prelude":
Reading script file "Parse":
Reading script file "Interact":
Reading script file "PrologData":
Reading script file "Subst":
Reading script file "StackEngine":
Reading script file "Main":

Writing C output file "miniProlog.c":
[Leaving Gofer->C]
machine%
```

This is another case where it might well have been sensible to have used a `.prj` or `.gp` for the project file `miniProlog` since compiling the C code in `miniProlog.c` to a file named '`miniProlog`' will overwrite the project file! Choose filenames with care!

You can also specify Gofer command line options as part of the command line used to run `gofc`. Think of it like this; use exactly the same command line to start Gofc as you would have done to start Gofer (ok, replacing the command '`gofer`' with '`gofc`') so that you could start your program immediately by evaluating the main expression. To summarize what happens next:

- Gofc will load the prelude file. Do not worry if the prelude (or indeed, later files) contain lots of definitions that your program will not actually use; only definitions which are actually required to evaluate the main expression will be included in the output file.
- Gofc will load the script files specified. If an error is found then an error message will be printed and the compilation will be aborted. You would probably be sensible to run your program through the interpreter first to tidy up any errors and avoid this problem.
- Gofc will look for a definition of '`main`' and check that it has type `Dialogue`. You will get an error if an appropriate main value cannot be found.
- Gofc determines the appropriate name for the output file.
- Gofc checks to make sure that you haven't used a primitive function that is not supported by the runtime system (see Section 5.2 for more details).
- Gofc outputs a C version of the program in the output file.

Once you have compiled the Gofer program to C, you need to compile the C code to build the executable application program. This will vary from one system to another and is documented elsewhere.

6.2 Primitive operations

The Gofer compiler accepts the same source language as the interpreter. However, there is a small collection of Gofer primitives which are only implemented in the interpreter. The most likely omission that you will notice is the `primPrint` function which is used to define the `show'` function in the standard prelude. Omitting this function is not an indication of laziness on my part; it is impossible to implement `primPrint` in the current runtime system because there is insufficient type information available at program runtime.

For example, if you try to compile the program:

```
main :: Dialogue
main  = appendChan "stdout" (show' 42) exit done
```

the compiler will respond with the error message:

```
ERROR: Primitive function primPrint is not
       supported by the gofc runtime system
       (used in the definition of show')
Aborting compilation
```

The solution is to use type classes. This is one of the reasons for including them in the language in the first place. This example can be compiled by changing the original program to:

```
main :: Dialogue
main  = appendChan "stdout" (show 42) exit done
```

(Remember that show is the overloaded function for converting values of any type a that is an instance of the Text class to a string value.)

6.3 Debugging output

Another potentially useful feature of `gofc` is it's ability to dump a listing of all the supercombinator definitions that are created by loading a particular combination of script files. For the time being, this is only useful for the purpose of debugging, but with only small modifications, it might be possible to use this as input to an alternative backend/code generator system (the format of the output combinators already uses explicit layout characters to make the task of parsing easier in an application like this).

To illustrate how this option might be used, suppose that we were working on a program containing the definition:

```
hidden xs = map (\[x] -> x) xs
```

and that somewhere during the execution of our program, this function is applied to a list value `[[1],[1,2]]`:

```
? hidden [[1],[1,2]]
[1,
Program error: {v132 [1, 2]}
(13 reductions, 75 cells)
```

The variable `v132` which appears here is the name used internally to represent the lambda expression in the definition of hidden. For this particular example, it

is fairly easy to work this out, but in general, it may not be so straightforward. Running the program through `gofc` and using the `+D` toggle as follows produces an output file containing Gofer SuperCombinators, hence the `.gsc` suffix:

```
machine% gofc +D file
Gofer->C Version 1.01 (2.28)  Copyright (c) Mark P Jones 1992-1993

[Writing supercombinators to "file.gsc"]
Reading script file "/usr/local/lib/Gofer/standard.prelude":
Reading script file "file":
[Leaving Gofer->C]
machine%
```

Note that there is no need in this situation for the files loaded to contain a definition for `main :: Dialogue`, although the compiler must be loaded using exactly the same prelude and order of files as in the original Gofer session to ensure that the same names are used. Scanning the output file, we find that the only mention of `v132` is in the definitions:

```
v132 o1 = case o1 of {
            (:) o3 o2 -> case o2 of {
                           [] -> o3;
                         }
          }

hidden o1 = map v132 o1;
```

This shows fairly clearly where the function `v132` comes from. Of course, this is far from perfect, but it might help someone to track down a bug that little bit faster one day. It's better than nothing.

Of course, the debugging output might also be of interest to anyone that wants to find out more about the implementation of Gofer and examine the supercombinator definitions generated when list comprehensions, overloading, local function definitions etc. have all been eliminated. For example, the standard prelude definitions of `map` and filter become:

```
map o2 o1 = case o1 of {
              [] -> [];
              (:) o4 o3 -> o2 o4 : map o2 o3;
            }

filter o2 o1 = case o1 of {
                 [] -> [];
                 (:) o4 o3 -> let { o5 = filter o2 o3;
                                  } in  | o2 o4 -> o4 : o5;
                                        | otherwise -> o5;
               }
```

This is one of the tools I'll be using if anyone ever reports another bug in the code generator...

7 Some history

Ever since the first version of Gofer was released I've had requests from Gofer users around the world asking how Gofer got its name and how it came into being. This

section is an attempt to try and answer those questions.

7.1 Why Gofer?

Everything has to have a name. You may type in an 'anonymous function' as a lambda expression but Gofer will still go ahead and give it a name. To tell the truth, I always intended the name 'Gofer' to be applied to my particular implementation of a functional programming environment, not to the language on which it is based. I wanted that to be an anonymous language. But common usage has given it the same name, Gofer.

If you take a look in a dictionary (as some puzzled Gofer users have) you'll find that 'gofer' means:

> an employee whose duties include running errands

(although you'd better choose a dictionary printed since the 70s for this). I'd not thought about this when I chose the name (and I would have used a lower case g instead of an upper case G if I had). In fact, Gofer was originally conceived as a system for machine assisted equational reasoning. One of the properties of functional languages that I find particularly attractive is that they are:

> **go**od **f**or **e**quational **r**easoning.

So now you know. The fact that you can also tell someone who is having a problem with their C program to "Gofer it!" (unsympathetic, I know) is nothing more than a coincidence. Fairly recently, somebody wrote to ask if Gofer stood for '**go**od **f**unctional programming **e**nvi**r**onment". I was flattered; I wish I'd thought of that one.

Some people have asked me why I didn't choose a title including the name 'Haskell', a language on which Gofer is very strongly based. There are two reasons for this. To start with, the original version of Gofer was based on a different syntax, Orwell + type classes. The Haskell influence only crept in when I started on version 2.xx. Secondly, it's only right to point out that there is quite a large gap between a system like Gofer and the full blown Haskell systems that have been developed. Using a name which doesn't involve 'Haskell' directly seemed the right thing to do. Some people tell me that it was a mistake. One of the objectives of Haskell was to create a standard language for non-strict functional programming. Gofer isn't intended as an alternative to Haskell and I hope it will continue to grow closer as time passes.

While I'm on the subject of names, I should also talk about an additional source of confusion that may sometimes crop up. While Gofer is a functional programming system, there is also a campus wide information system called 'Gopher' (sharing it's name with the North American rodents). I would guess that the latter has many more users than the former. So please be careful to spell Gofer with an 'f' not a 'ph' to try and minimize the confusion.

It has occurred to me that I should try and think of another name for Gofer to avoid the confusion with Gopher. I hope that won't be necessary, but if you have a really good suggestion, let me know! One possibility might be to call it 'Gordon'. The younger generation of brits might know what the connection is. Others may need to ask their children...

7.2 The history of Gofer

Here is a summary of the way that I first learnt about functional programming, and how it started me on the path to writing Gofer. This, slightly sentimental review

is mostly for my own entertainment. If you're the sort of person that likes to read the acknowledgments and bibliographic notes in a thesis: this is for you. If not, you can always stop reading :-)

My first exposure to lazy functional programming languages was using a language called 'Orwell' developed and used at the Programming Research Group in Oxford. I've been interested in using and implementing lazy functional programming languages ever since.

One of the properties of programming in Orwell that appealed to me was the ability to use equational reasoning – a very simple style of mathematical reasoning – to establish properties of programs and prove that they would behave in particular ways. Even more interesting, equational reasoning can be used to calculate efficient implementations of programs from a formal specification of what was intended.

Probably the first non-trivial functional program that I wrote was a simple Prolog interpreter. (This was originally written in Orwell and later transcribed to be compiled using the Chalmers Haskell B compiler, hbc. The remnants of this program live on in the mini Prolog interpreter that is included with the Gofer distribution and, I believe, with at least a couple of the big Haskell systems.) Using a sequence of something like a dozen or so transformations (most of which were fairly mundane), I discovered that I could turn a relatively abstract specification of a Prolog inference engine into a program that could be interpreted as the definition of a low level stack-based machine for executing Prolog queries. Indeed, I used the result as the core of a C implementation of mini Prolog.

The transformations themselves were simple enough but managing the complexity of the calculations was tough. It was not uncommon to find that some of the intermediate steps in a calculation would span more than 200 characters. Even with a relatively small number of transformation steps, carrying out proofs like this was both tedious and prone to mistakes. A natural application for a computer!

Here's an outline of what happened next:

eqr 1989. Eqr was a crude tool for machine assisted equational reasoning. It worked well enough for the job I had intended to use it for, but it also had a number of problems. I particularly missed the ability to use and record type information as part of an automated derivation.

1.xx 1990. Gofer 1.xx was intended to be the next step forward providing machine support for *typed* equational reasoning. It was based on Orwell syntax and was later extended to support Haskell style type classes. It had a lexer, parser, type checker and simple top-level interactive loop. It couldn't run programs or construct derivations.

2.xx January 1991. A complete rewrite. I remember those early days, several months passed before I ever got compile some of the earliest code. The emphasis switched to being able to run programs rather than derive them when I came up with a new implementation technique for type classes in February 1991. If I wanted to see it implemented, I was going to have to do it myself. Around about May, I realized I had something that might be useful to other people.

2.20 The first public release, announced in August 1991 and distributed shortly after that in September.

2.21 November 1991, providing a more comprehensive user interface, access to command line options and fixing a small number of embarrassing bugs in the original release.

2.23 August 1992, having been somewhat preoccupied with academic studies for some time, the main purpose of this release was to correct a number of minor bugs which had again been discovered, either by myself or by one or more of the many Gofer users out there.

2.28 January 1993. The most substantial update to Gofer since the original release. I had been doing a lot of work and experimentation with Gofer during the time between the release of versions 2.21 and 2.23, but I didn't have the time to get these extensions suitable for public distribution. By the time I came to release version 2.23, I also had several other distinct versions of Gofer (each derived from the source for version 2.21) including a compiler and a prototype implementation of constructor classes which was called 'ccgofer'. Work on version 2.28 started with efforts to merge these developments back into a single system (I was tired of trying to maintain several different versions, even though I was the only one using them). The rough outline of changes was as follows (with the corresponding version numbers for those who wonder why 2.28 follows 2.23):

2.24	enhancements and bug fixes
2.25	merging in support for the Gofer compiler
2.26	a reimplementation of constructor classes
2.27	reworked code generator and other minor fixes
2.28	preparation for public release