The implementation of the Gofer functional programming system

Mark P. Jones

Research Report YALEU/DCS/RR-1030
May 1994

This is work is supported in part by Darpa N00014-91-J-4043.

The implementation of the Gofer functional programming system

Mark P. Jones
Yale University, Department of Computer Science,
P.O. Box 208285, New Haven, CT 06520-8285.
jones-mark@cs.yale.edu

Research Report YALEU/DCS/RR-1030 May 1994

Abstract

The Gofer system is a functional programming environment for a small, Haskell-like language. Supporting a wide range of different machines, including home computers, the system is widely used, both for teaching and research.

This report describes the main ideas and techniques used in the implementation of Gofer. This information will be particularly useful for work using Gofer as a platform to explore the use of new language features or primitives. It should also be of interest to those curious to see how the general techniques of functional programming language compilation are adapted to a simple, but practical, implementation.

Introduction

The Gofer system is a functional programming environment for a small, Haskell-like language. The language can be characterized by its support for lazy evaluation and higher-order functions, and a static type system that includes both polymorphism and overloading. First released in September 1991, Gofer is widely used, both for education and research. Judging from comments from its users, there are two main reasons for Gofer's popularity. First, that it runs on a wide range of machines, including small home computers. Second, particularly on more powerful systems, the interpreter provides a fast, interactive development environment, avoiding the need for lengthy recompilation.

While the user documentation [24] and C source code for Gofer have always been included in public distributions, there was no substantial effort to provide details about the implementation. At the time, this seemed unnecessary; although it introduced some new ideas, most of the techniques used were believed to be standard and well-known.

In retrospect, user feedback during the past few years suggests that some of these ideas may not be as well-known as we had expected. There has also been quite a lot of interest in modifying the Gofer system, from simple tasks such as the addition of new primitives, to more sophisticated experiments in language design. While this information can, in principle, be gleaned by a careful study of the source code, it is certainly not the most convenient form of documentation!

This report is intended as a guide for those interested in the inner workings of Gofer. Its aim is to explain the original design goals, the overall structure, the datatypes used, and the way that the different pieces fit together. It does not attempt to cover every technical detail of the system and should be viewed as an introduction, not a replacement, for studies of the source code. Nor is this report intended as a tutorial on 'the implementation of functional languages'; indeed, Simon Peyton Jones' book by that name [48] was a constant companion during the development of Gofer, and remains an almost certain prerequisite to this report.

Throughout this report, we assume familiarity with the Gofer environment as described in the user documentation [24, 25, 32], with the Haskell functional programming language on which much of the language design is based [16], and with the C programming language in which the system is implemented.

Outline of report

We begin in Section 1, outlining the principal design goals for the development of the Gofer system. Section 2 summarizes the main features of the system from the user's perspective, highlighting some of the decisions made in the design of the standard user interface. In Section 3, we begin a more detailed study of the current implementation of Gofer. Starting with an overview of the whole system, we progress through

detailed descriptions of the most important components including storage management (Section 4), lexical analysis and parsing (Section 5), static analysis (Section 6), type checking (Section 7), compilation to supercombinators (Section 8), and program execution (Section 9). Section 10 discusses some of the ways that functional programming has been used and influenced the development of Gofer, even though the system was actually written in imperative C. The Gofer compiler, a simple-minded translator producing executable C versions of Gofer programs, is described in Section 11. Finally, Section 12 offers some reflections on the current implementation and some areas for future development.

1 Motivation and design goals

The Gofer system was initially developed as a vehicle for personal research. The name Gofer[1] is derived from the claim that functional languages are supposedly 'Good for equational reasoning', a reflection on the fact that the system was originally planned as a tool for machine-assisted equational reasoning. The initial design included a small interpreter intended only for simple calculations, but this soon became the main focus of the project, motivated in particular by the desire to investigate a new implementation technique for the Haskell overloading mechanism [21].

The system was designed with some specific goals in mind:

- **Compatibility**: The language design should be closely based on the definition of Haskell, an emerging and freely available standard for non-strict functional programming languages [16]. Of course, Haskell was also a natural choice for work on type class overloading and guaranteed an initial supply of examples on which to base any comparisons. Finally, it seemed sensible to avoid the notoriously difficult process of language design, concentrating instead on specific elements of a carefully developed system.
- **Extensibility**: The system should provide a good foundation for further work and experimentation.
- **Portability**: The system should be usable on a wide range of machines. A very specific constraint was that it should be able to run the system on an 8MHz 8086 based PC with 640K RAM running MS-DOS; all of the original development work was carried out on this machine. Such systems are infamous for their awkward 16 bit segmented architecture that limits the size of individual blocks of data to 64K. Although the Gofer system runs on many different machines without such restrictions, this has continued to be an important influence on the basic design.

Note that, although speed of execution is an important part of making an implementation usable, we did not consider this as a primary design goal.

Looking back, we believe that we have been reasonably successful in meeting these goals. Initially based on version 1.0 of the Haskell report [17], Gofer has been modified to track subsequent versions of Haskell. As a result, there is a large class of programs that can be run using either Haskell or Gofer without requiring changes to the source code. Unfortunately, there are some differences, most significantly in the treatment of type class overloading. It is easier now to realize that the goal of compatibility stands in direct conflict with the technical ideas that motivated its development. We still believe that the Gofer approach to type classes has some important advantages, but the incompatibility with Haskell, although minor, has become a little frustrating. The differences are also something of an embarrassment, given that one of the goals in the design of Haskell was to 'reduce unnecessary diversity in functional programming languages'.

Gofer has also proved to be useful in subsequent work, and some of these developments have found their way into general releases of the Gofer system. Examples of this include:

- A simple-minded compiler, translating Gofer programs to C to enable the development of standalone applications written in Gofer.
- An system of *constructor classes*, extending the notion of Haskell type classes, to allow more sophisticated forms of user-defined overloading [33].

Others, for example, the specialized partial evaluator described in [31], have yet to be included in the standard distribution.

Finally, the Gofer system has proved to be fairly portable and includes support for a wide variety of different machine and operating system environments. In a few places, the source code depends on assumptions about the compiler used to build Gofer

[1] We were not aware of the similarly named gopher program—used to access information over the Internet—until some time after the first release of Gofer. Despite their very different application areas, the choice of names has been an unfortunate source of confusion, and may soon warrant the choice of a new name for Gofer.

which cannot be guaranteed by the C language standards. In this respect, we seem to have been fairly lucky that our non-portable assumptions happen to be true for the systems used. Despite the new features added since the first release, Gofer can still be used on the same small machines that it was originally developed for. However, most of the machines in use today, even home computers, are much more powerful. We may see significant benefits in future versions of the system once we are free of the constraints that the older machines impose.

The fact that Gofer can be used on small machines seems to be one of the main reasons for its widespread use. However, while some other functional language implementations do require more powerful machines, Gofer is by no means unique in providing a functional programming environment for smaller systems; Leroy's Caml Light [40], implementing a dialect of ML, is another well-known example.

2 A user's perspective

The Gofer system[2] provides an interpreter for an experimental language that is closely based on the definition of Haskell version 1.2 [16]. Notable features include:

- A purely functional language with non-strict semantics (lazy evaluation), higher-order functions, pattern matching, ...
- Facilities for defining new algebraic datatypes and type synonyms.
- A polymorphic type system with provision for user-defined overloading based on a system of type classes (see Section 7.5 for further details).
- Full Haskell expression and pattern syntax including lambda, case, conditional and let expressions, list comprehensions, operator sections, and wildcard, as and irrefutable patterns.
- A partial implementation of the Haskell I/O facilities, supporting simple text file manipulation and interactive programs.
- User documentation, sample programs and source code freely available by anonymous ftp.
- A relatively portable implementation runs on a wide range of computer systems including several smaller home/personal computers.

[2]This report is based on Gofer version 2.30, the current release at the time of writing. Most of the comments in this report are also true of earlier versions.

The only significant feature of Haskell that is not currently supported is the Haskell module system. On the other hand, Gofer also supports several experimental extensions of which constructor classes are perhaps the best known example. The latest release also includes prototype implementations of other recent proposals for extensions to Haskell, including monadic I/O [47] and lazy state threads [39].

In a typical Gofer session, the system behaves like an interactive calculator. First, the user enters an expression at the Gofer prompt, usually a ? character. After checking for errors, the interpreter evaluates the expression, lazily printing the result as it is produced, and then returning to the prompt for another expression:

```
? 2 * (3+4)
14
(5 reductions, 10 cells)
? sum [1..10]
55
(92 reductions, 132 cells)
?
```

The counts of `reductions` and `cells` give a rough indication of the amount of time and memory used by the calculation, respectively. These statistics can be useful in some situations for comparing the relative complexity of different algorithms. However, these messages can also be suppressed using a command line option to avoid unnecessary distractions.

The process described above has much in common with the standard read-eval-print loop used in many interactive programming environments, particularly those for languages like Lisp and Scheme. Gofer allows local definitions within an expression, for example:

```
? f 3 where f x = x*x - 3*x + 7
7
(7 reductions, 13 cells)
?
```

However, unlike most Lisp systems, it is not possible to define new global values or functions directly from within the interpreter. Instead, new definitions are entered using a text editor to produce a *script* file of definitions that can then be loaded into the interpreter. The strict separation between sessions and scripts was a conscious design decision, inspired by [7] and [58], reflecting the way we expect the system to be used in practice. In particular, this approach allows a programmer to work by switching between scripts and sessions, adding new definitions, testing,

and making further changes as necessary. In addition, script files provide a consistent view of the current program, allowing program development to be spread across different Gofer sessions, but also avoiding the complexities (and confusions) of incremental compilation and type checking.

The user interface is carefully designed to support this style of program development. For example, if an error is detected while loading a script file, the user can enter the command `:e`, an abbreviation for `:edit`, to start up a text editor on the file containing the error at the approximate line position where the problem was detected. Once the error has been corrected, the user saves the file and exits the editor, returning to the interpreter, which automatically reloads the script.

The following example shows how the `:load` command is used to load the definitions in the script file `example.gs` into the interpreter:

```
? :load example.gs
Reading script file "example.gs":

Gofer session for:
/Gofer/Standard.prelude
example.gs
?
```

Notice that the list of files displayed here includes not only `example.gs`, but also a *prelude* file called `/Gofer/Standard.prelude`. This is a script of standard definitions that are loaded at the beginning of every Gofer session. For example, the prelude typically includes the definition of standard arithmetic and list processing functions. The interpreter allows users to develop special versions of the prelude to suit particular requirements. For example, this feature has been used to support experimental preludes using constructor classes, or to provide closer compatibility with Haskell, or with a textbook such as [7] in an introductory course on functional programming. On the other hand, the system is carefully designed to ensure that the prelude file will only be loaded once when the interpreter is started. This is intended to discourage users from the temptation of modifying prelude definitions; since these definitions are usually shared by many different programs, it is bad practice to modify them for the purposes of one particular program. The preferred method for writing new prelude files is to build the new definitions on top of a minimal prelude; the new definitions should only be moved into a stand-alone prelude when the development is complete. It is entirely desirable for this process to be a little awkward; new preludes should only be used as a last resort since they are inherently non-standard.

Although Gofer does not include a module system, it is often convenient to split large programs into several different script files. This allows different components of a program to be developed independently, and perhaps reused in later programs. The Gofer system supports this by allowing the user to specify a list of script files, each of which is loaded, stacked on top of the definitions in the preceding files. For example, the following command might be used to load three script files into the interpreter:

```
? :load ex1.gs ex2.gs ex3.gs
```

Since the files are loaded in strict order, it is possible for `ex2.gs` to include references to definitions in `ex1.gs`, but not to those in `ex3.gs`. In a similar way, if the definitions in `ex2.gs` are changed, then both `ex2.gs` and `ex3.gs` will be reloaded, since the definitions in the latter might potentially be modified in light of changes in the former. A more sophisticated dependency analysis could be used to avoid reloading `ex3.gs` in cases where it would not be affected by any changes in `ex2.gs`. However, in practice, the reloading of script files is usually fast enough that this is not a concern.

For convenience, Gofer allows the list of files required by a particular program to be recorded in a *project file*. A command of the form:

```
? :project program.prj
```

can be used to read the file names listed in `program.prj` and load the corresponding script files into the interpreter.

It is important to mention that the original Gofer user interface was closely modeled on the 'scrolling' interface of Orwell [58] which provided my own introduction to functional programming. For example, following Orwell, early versions of Gofer displayed lines of dots to indicate progress during various stages of reading script files[3]. The command syntax of Gofer, using a leading colon character, `:`, was also modeled on that of the editor `vi`, versions of which are available for many of the systems on which Gofer is used.

In comparison with popular programs on some machines, the user interface of Gofer is very primitive. Unfortunately, there are still many different stan-

[3] This feature is still supported by the current release but requires a command line option, `+.`. It is no longer used as the default behaviour because it has a surprisingly significant slowing effect on the total time taken to process script files on some machines.

dards for constructing more sophisticated user interfaces and we are not aware of any standard toolkits offering the degree of portability required for the Gofer system. However, several people have developed graphical user interfaces for Gofer, targeted for specific machines. The most widely known example is MacGofer which extends the standard Gofer interpreter with a graphical user interface for Apple Macintosh computers. The MacGofer system has been developed by Kevin Hammond at the University of Glasgow.

3 Overall structure

After all of the background information given above, it is finally time to begin our tour through the Gofer system!

3.1 Implementation language

One of the first decisions to be made was the choice of implementation language. Given that Gofer was intended to run on a wide variety of machines, including small PCs with limited memory, the C language was an obvious candidate. Even so, the decision to use C was not easy; we believe that the development of Gofer would have been both easier, and less error-prone, had it been written in a strongly typed functional language which it accepts. This approach is often used by designers as a means of testing, and of demonstrating confidence, in the use of a new language for program development. For example, Chalmers LML [4], Standard ML of New Jersey [3] and Glasgow Haskell [12] are all able to compile their own source code. However, each of these systems requires substantial machine resources. In addition, we were also concerned about bootstrapping; each of the systems above relies on a compiler to achieve reasonable performance, while Gofer was originally conceived as an interpreter. Nevertheless, even though Gofer was written in an imperative language, it also shows strong influences from functional programming. This is discussed in a little more detail in Section 10.

Although compilers for ANSI C were available at the time, the Gofer code was written so that it could also be compiled with older K&R compilers [37]. The C preprocessor was used to allow the use of important features of ANSI C where possible. For example, a preprocessor macro, `Args(arglist)` is used to allow the use of function prototypes in declarations such as:

```
Void parseScript Args((String,Long));
```

For an ANSI C compiler, the `Args` macro includes the types of the arguments of `parseScript` in the declaration, allowing compile-time detection of some argument mismatch errors. For older compilers, the `Args` macro is defined to omit the list of argument types from the preprocessed version of the program.

It turns out that many of the datatypes in Gofer are just synonyms for existing C types. For example, the types `Int`, `Text`, `Cell`, `List`, `Pair` and `Module` are just new names for the standard C type of integers, `int`. However, careful use of these different names provides valuable program documentation, suggesting how we intend different values to be treated. For example, the prototype for the `copy` function defined in `storage.c` is written as:

```
extern List copy Args((Int,Cell));
```

which gives a much clearer indication of what this function does (returning a list containing some fixed number of copies of a given `Cell`) than if we had written just:

```
extern int copy Args((int,int));
```

Of course, this looses the benefits of strong typing; for example, most C compilers will not report any errors for an expression like `copy(val,10)`, despite the fact that the arguments have been written in the wrong order. One way to restore strong typing would have been to introduce new types rather than synonyms. For example:

```
typedef struct {int cellValue;} Cell;
```

This might be sensible in C++, using `classes` instead of `structs`, but is not suitable for some older C compilers that prohibit the use of `struct` values as function arguments. In truth, the ability to make puns, implicitly treating values of one type as values of another, turned out to be quite convenient at several points in the code.

Another important use of the C preprocessor in the Gofer source is to support the display and handling of error messages. For example, the code:

```
ERROR(line) "Syntax error"
EEND;
```

can be used to display an error message about a syntax error in line `line` of a particular script file and to branch to the error handler. A small collection of macro definitions in the file `errors.h` provides additional facilities for including expressions and types in error messages. One advantage of this approach is

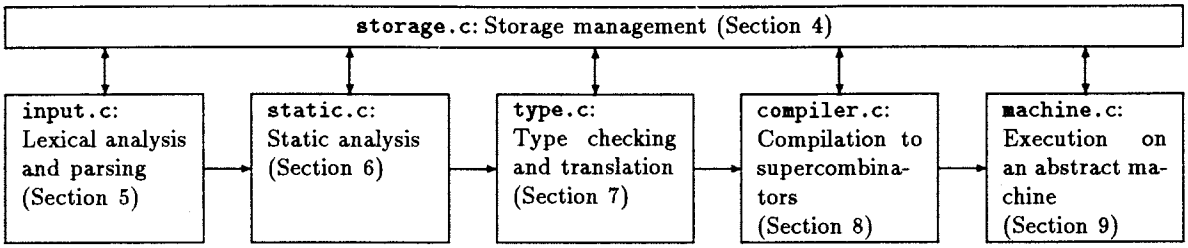


Figure 1: Main components of the Gofer system

that it makes it much easier to identify error messages in the source code, and helps to ensure that they are all handed in a uniform manner.

The reader may be surprised by some of the concessions that have been made to support older, and often obsolete, compilers and architectures. However, we have to recognize that many such systems are still in use, particularly as the personal computers on which Gofer is most widely used.

3.2 Components of the Gofer system

The main components in the Gofer system are illustrated by the diagram in Figure 1, which also corresponds to the way that input programs are processed, from lexical analysis to execution on the Gofer abstract machine. Each of these components is implemented by a corresponding C file, described in the following sections of the report, starting with `storage.c` which provides the storage management facilities used by all of the other components.

Each part of the system is responsible for initializing and maintaining the variables and data structures on which it depends. This is described by including a 'control function' in each component. For example, the control function for the type checker is called `typeChecker()`. This can be used, for example, to reset the type checker by a call of the form `typeChecker(RESET)`. Several other messages are understood by control functions, including `INSTALL`, to initialize local data structures, `MARK`, to request the component to mark local data structures in preparation for garbage collection, and `EXIT`, to perform any final tidying up necessary before exiting the interpreter. The `everybody()` function defined in `commonui.c` can be used to send a particular message to every component of the system. For example, one of the first things that the interpreter does is to call `everybody(INSTALL)`, and one of the last things it does before terminating is to call `everybody(EXIT)`.

There some other components of the Gofer system that are not included in Figure 1, including:

- `gofer.c`: The standard interpreter user interface. Includes code from `commonui.c` (parts of the user interface shared with the Gofer compiler, `gofc`, for example, command line processing) and from `machdep.c` (machine dependent code, for example, terminal I/O).
- `builtin.c`: Defines the built-in primitives, included from the file `prims.c`.
- `output.c`: Provides a pretty-printer for the internal representations of programs, expressions, types and kinds. These routines are mostly used to display fragments of expressions or types as part of error messages produced by the Gofer system. They have also been quite useful when debugging the interpreter itself!

We should also mention the files `gofc.c`, `cmachine.c` and `cbuiltin.c` which are variants of `gofer.c`, `machine.c` and `builtin.c`, respectively, used in the Gofer compiler, `gofc`. See Section 11 for further details.

4 Storage management

One of the most important components of the Gofer system is the storage management facilities provided by `storage.c` and the header file `storage.h`. Together, these define many of the datatypes and structures that are used in the other parts of the system. The most important data structures used in the Gofer system are as follows:

- The `Text` datatype, used to represent identifier names and text strings (Section 4.1).
- The `Cell` datatype, used as a representation for a wide variety of values, including programs and types (Section 4.2).

- The `Module` datatype, used to deal with programs that are built up by loading a sequence of script files (Section 4.3).
- The Gofer stack, used for a variety of purposes, including the execution of programs in the Gofer abstract machine (Section 4.4).

4.1 The `Text` datatype

Gofer programs usually contain many different character strings, including keywords, variable names and the text of string literals. A naive treatment of strings, allocating fresh storage for each string as it is encountered, has obvious disadvantages:

- Heap space may be wasted by keeping multiple copies of the same string, for example, a function name that is used throughout a source program.
- The memory allocators used in some C runtime systems perform poorly when burdened with the allocation of many, relatively small objects.
- Comparisons between strings are not atomic; several character comparisons are necessary to distinguish between strings with the same initial prefix. This is particularly important, for example, in code that searches a symbol table to find an entry corresponding to a particular string.

Gofer avoids these problems using a table of character strings and associating each string with its offset in this table; we use values of type `Text` to record these offsets. Strings are only added to the table the first time that they are encountered. Subsequent occurrences are given the same `Text` value as the first. Using `Text` values to represent strings avoids all of the problems described above, and ensures that two strings are equal if, and only if, the corresponding `Text` values are equal.

In the current implementation, the size of the character storage table is fixed at compile-time. With the limited memory of an older PC, there was little point in doing anything more sophisticated than this, particularly since a dynamically allocated table would have required a further indirection to access values in the table. However, a dynamically allocated table would be useful on more powerful systems, allowing the size of the character table to be increased without a complete recompilation of the Gofer system.

Two functions are provided to convert between the `String` and `Text` types:

```
extern String textToStr Args((Text));
extern Text   findText  Args((String));
```

The first of these simply returns a pointer to the string at the offset in the table determined by the `Text` value. The `findText` function is used to find the `Text` value corresponding to a given `String`, extending the character table if necessary. A collection of hash tables is used to speed the process of searching the character table for previous occurrences of a string. In the original implementation, a single hash table was used to give the starting point for a linear search through the character table. This was quickly changed, to use multiple hash tables, when we discovered that over 80% of the time required to load large programs was spent inside the `findText` function, searching for previous occurrences of strings! Where possible, the current implementation now uses 10 levels of hash tables before resorting to a linear search. For large programs, this can gave an immediate fivefold increase in compilation speed. The benefits for smaller programs are less significant, but still worthwhile.

Two further functions:

```
extern Text inventText     Args((Void));
extern Text inventDictText Args((Void));
```

are used to generate 'new' variables and dictionary parameters, respectively, in later stages of the compiler. They are implemented as functions that return values outside the range of offsets into the character table.

4.2 The `Cell` datatype

One of the advantages of programming in a language like Gofer is the use of automatic storage allocation and garbage collection. This frees the programmer from the need to specify when memory allocation is required, or when allocated sections of memory can be released for use in other parts of the program. Explicit memory management of this kind is a notorious source of errors in languages like C.

To support automatic storage management, the Gofer system includes an implementation of a garbage-collected heap with values of type `Cell` used as 'pointers' to objects in the heap. This turns out to be a very useful data structure and is used throughout the system, not just for the execution of Gofer programs. For example, the Gofer heap is also used to build parse trees for input programs, to store types for use in the type checker, and to implement the translation of functions to supercombinators.

In the following subsections we outline both the interpretation and implementation of `Cell` values. Be warned that any reasonably complete description of

`Cell` values is likely to be long and complicated. For a first reading, it may be best to simply scan the following sections, without attempting to absorb too many of the details.

4.2.1 Interpretation of `Cell` values

The `Cell` datatype includes representations for many different kinds of value. The following list describes some of the most important kinds of `Cell` value:

- **Primitive constants**: The `Cell` datatype includes representations for several different kinds of constant values, including integers (corresponding to the `int` type in the underlying C implementation), floating point numbers, characters and string literals.

- **Pairs**: Used to construct compound values, including pairs, trees and lists. A pair containing two given `Cell` values, `x` and `y`, can be obtained as the result of the function call `pair(x,y)`. Conversely, the two components of a pair `p` can be extracted using the expressions `fst(p)` and `snd(p)`.

  Pair cells are also used to represent function application in both the compile-time and run-time representation of Gofer programs. The `storage.h` header file includes the following macro definitions to support this use:

```
#define ap(f,x)     pair(f,x)
#define fun(c)      fst(c)
#define arg(c)      snd(c)
```

  Pairs are also used to build a representation for lists; `storage.h` contains the following macro definitions to describe the encoding of lists:

```
#define cons(x,xs)    pair(x,xs)
#define hd(xs)        fst(xs)
#define tl(xs)        snd(xs)
#define singleton(x) cons(x,NIL)
#define isNull(x)     ((x)==NIL)
```

  In words, a non-empty list is represented by a pair whose `fst` component stores the head of the list, with the tail of the list (i.e. the remaining elements) in the `snd` component. The special cell value `NIL` is used to represent the empty list (and in other parts of the implementation as a general purpose 'dummy' value). We refer to lists constructed in this manner as being 'internal' lists; values corresponding to the list datatype used by the Gofer program have a very different representation. For example, the fully evaluated Gofer list `[1..3]` is represented by:

```
ap(ap(nameCons,mkInt(1)),
   ap(ap(nameCons,mkInt(2)),
      ap(ap(nameCons,mkInt(3)),
         nameNil)))
```

  where `nameCons` and `nameNil` are constant values (see the description of name values below) corresponding to the Gofer constructor functions `(:)` and `[]`, respectively.

  Using three different names, for example, `pair`, `ap` and `cons`, for a single function may seem an unnecessary complication. However, in practice, careful use of these different names actually makes the code a little more readable.

- **Variable names**: There are several different kinds of variable names that can be represented as `Cell` values, including ordinary variables, constructor variables and dictionary variables. In each case, the expression `textOf(v)` gives the `Text` name for the variable represented by a `Cell` value `v`.

- **Constructor cells**: The `Cell` datatype includes representations for fragments of parse trees. For example, a list comprehension generator of the form `p<-exp` is represented by a `FROMQUAL` cell that contains a pair whose components are the pattern `p` and the expression `exp`. In later sections, particularly in grammars describing internal representations of Gofer programs, we will use the notation `FROMQUAL (p,exp)` to describe a `Cell` value that is constructed in this way.

- **Special cell values**: The `Cell` datatype includes a number of special values for certain constants. Examples of this include:

  - `ARROW`, `LIST` and `UNIT`, used to represent the function space `(->)`, list `[ ]` and unit `()` type constructors, respectively. `UNIT` is also used as the representation for the Gofer value `()`, the only value (other than $\perp$) of type `()`.
  - `WILDCARD`, representing the wildcard pattern `_`.
  - `STAR`, representing the kind of all types, an important concept in the treatment of constructor classes [33]. Function kinds, mapping constructors of kind `k1` to constructors of kind `k2`, are represented by `pair(k1,k2)`.

- **Tuples**: Used in the representation of tuple types and values. For example, `mkTuple(2)` is used in the construction of pairs. The `Cell` datatype actually includes values corresponding to nullary and unary tuples, `mkTuple(0)` and `mkTuple(1)`, respectively, but they are not used in the current implementation.

- **Offsets**: Used as place holders for type variables in a polymorphic type, or for bound variables in a supercombinator definition. See Sections 7 and 8 for more details.

- **Dictionaries and selectors**: Dictionaries are used to package groups of related values as a single unit. Selectors are used to extract the components of a dictionary. This is described in more detail in Section 7.5.

- **Type constructors**: Used to represent Gofer type constructors introduced by `type` or `data` declarations, or as primitive built-in datatypes such as `Bool`, `Int` and `Char`. The details about the type constructor represented by a `Cell` value `tc` are recorded in the C structure `tycon(tc)`. For example:

  - `tycon(tc).text` gives the name of `tc` as a `Text` value.
  - `tycon(tc).kind` gives the kind of `tc`, as described in [33].
  - `tycon(tc).what` contains a code indicating whether the type constructor was defined by a `data` definition (`DATATYPE`), a `type` synonym (`SYNONYM`), or a restricted `type` synonym (`RESTRICTSYN`).
  - `tycon(tc).defn` contains the expansion of a type synonym or the list of constructor functions for an algebraic datatype.

  Note that `tycon()` is a macro, not a function returning a `struct` as the syntax might suggest.

- **Names**: Used to represent named values that are needed for the execution of Gofer programs. This includes user-defined values, constructor functions, member functions, primitives and new supercombinators introduced by the compiler. Like type constructors, the details corresponding to a `Cell` value `n` representing a name are held in a C structure `name(n)`, including the fields:

  - `name(n).text`: the `Text` name of `n`.
  - `name(n).arity`: the number of arguments that the function expects. An arity of zero indicates a constant applicative form (CAF) as described in [48].
  - `name(n).type`: the type of `n`, if known.
  - `name(n).defn`: used to distinguish different kinds of name. The value of this field may indicate a constructor function `CFUN` or a member function `MFUN`. For user-defined functions, this field is also used to record dependency information during compilation. It is also used to save the result of the first (and thereafter, only) evaluation of a CAF in `machine.c`.
  - `name(n).code`: the starting address for the code used to implement a supercombinator. See Section 9 for more details.
  - `name(n).primDef`: a pointer the implementation of a primitive function (or zero, if the name does not correspond to a primitive).

  As in the case of type constructors, `name()` is a macro, not a function returning a `struct`.

- **Type classes**: Used to represent individual type (or constructor) classes. Once again, if `c` is a `Cell` value representing a class then there is a structure `class(c)` that includes fields:

  - `class(c).text`: the `Text` name of `c`.
  - `class(c).arity`: the number of arguments that the class takes.
  - `class(c).sig`: the kind of constructor expected for each parameter.
  - `class(c).supers`: the list of superclass constraints.
  - `class(c).members`: the list of member functions for `c`.
  - `class(c).defaults`: the list of default definitions for each member function (if any).
  - `class(c).instances`: the list of instance declarations for class `c`.
  - `class(c).dictIndex`: a pointer to the index of dictionary values for the class.

  The use of these values is described in more detail in Section 7.5.

- **Type class instances**: Like type constructors and names, the details about the instances of a type class are kept in structures indexed by `Cell` values. For example, if `in` is the `Cell` value corresponding to the instance declaration

Value	whatIs code	predicate	selector	constructor(s)
Integers	INTCELL	isInt	intOf	mkInt
Floating point	FLOATCELL	isFloat	floatOf	mkFloat
Characters	CHARCELL	isChar	charOf	mkChar
Pairs	AP	isPair	fst, snd	pair
Tuples	TUPLE	isTuple	tupleOf	mkTuple
Offsets	OFFSET	isOffset	offsetOf	mkOffset
Selectors	SELECT	isSelect	selectOf	mkSelect
Dictionaries	DICTCELL	—	dictOf	mkDict
Type constructors	TYCON	isTycon	tycon	mkTycon, newTycon
Names	NAME	isName	name	mkName, newName
Type classes	CLASS	isClass	class	mkClass, newClass
Class instances	INSTANCE	isInst	inst	mkInst, newInst

Figure 2: Predicates, selectors and constructors for Cell values

```
instance (Eq a) => Eq [a] where
   [] == []  =  True
   ...
```

then the structure inst(in) includes fields:

- inst(in).cl: The class that the instance applies to; in this case, Eq.
- inst(in).head: The form of instance specified; in this case, Eq [a].
- inst(in).specifics: The context part of the declaration; in this case, a singleton list containing the class constraint Eq a.
- inst(in).implements: The list of member function implementations; in this case, a singleton containing the name of the function implementing equality of lists.

Once again, the use of these values is described in more detail in Section 7.5.

Use of these different kinds of Cell values is supported by a small collection of utility functions, many of which are summarized in Figure 2, including:

- Predicates, to determine if a given Cell represents a particular kind of value.
- Selectors, to return the value associated corresponding to a particular kind of Cell. The fst, snd and name functions described above are all examples of this.
- Constructors, to obtain Cell values corresponding to particular kinds of value. The pair function described above is a simple example of this.

The whatIs function provides another way to determine what kind of value is represented by a given Cell. Figure 2 lists some of the codes returned by whatIs for frequently used Cell values; for example, a test of the form whatIs(c)==NAME is equivalent to isName(c), although the latter may be a little more efficient. The whatIs function is more useful as a means of testing for particular types of Cell that do not have a predefined predicate. For example, testing the condition whatIs(c)==DICTCELL is the easiest way to determine whether c represents a dictionary value. Another important application of whatIs is to provide a simple form of pattern matching; the following idiom is widely used in the implementation of Gofer, reducing the need for repeated inspection of Cell values:

```
switch (whatIs(e)) {
    case NAME     : ...
    case CHARCELL : ...
    ...
    default       : ...
}
```

Profiling experiments suggest that whatIs is one of the most frequently executed functions in the whole Gofer system, so it is certainly worth investing a little time to try and make it as efficient as possible.

4.2.2 Implementation of Cell values

Starting from the description of Cell values above, the most obvious implementation would be to use an algebraic data type, similar to those that a Gofer programmer would normally introduce using a data definition.

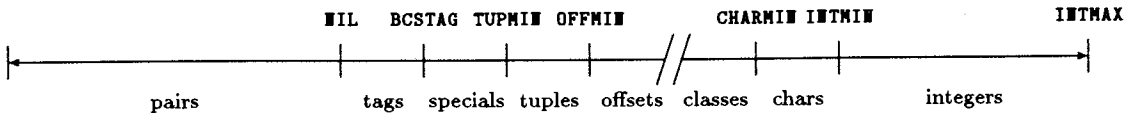


Figure 3: Implementation of `Cell` values as ranges of integers (not to scale).

In fact, the current implementation uses a rather more complicated scheme, encoding arbitrary `Cell` values as integers. The main idea is to use different ranges of integers to represent different kinds of value. Figure 3 illustrates part of the encoding used in the current implementation. This approach has some important benefits over the more direct implementation:

- It avoids the use of pointers in the representation of `Cell` values. This was an important consideration for the PC implementation which would have required 32 bit pointers and, as a result, doubled the storage requirements of a system using 16 bit integers.
- Occurrences of commonly used values, including names, type constructors and small integers, do not require any storage, regardless of the number of times that they are used during the compilation or execution of a program.

At the same time, this representation also has some significant disadvantages:

- The task of decoding `Cell` values can be a little expensive. The use of integers instead of pointers also carries some performance overheads, for much the same reason that an array access through a single pointer indirection is usually more efficient than using an integer index which involves additional, implicit pointer arithmetic.
- It is very difficult to change the mapping between integer ranges and different kinds of `Cell` value while the system is running. This might be important if we wanted to be able to modify the initial configuration, for example, to accommodate larger input programs.

The integer ranges used in the current implementation, and the values that they correspond to, are as described below. A long string of `#define` directives in the header file `storage.h` is used to ensure that none of these ranges overlap.

- Negative integers are used exclusively to represent pair values. This allows us to implement the test `isPair(c)` to determine if `c` represents a pair, as a comparison `c<0` that is very cheap on most machines. In theory, this limits the maximum number of pair cells in the heap to be equal to the number of negative integers (i.e. 2^{b-1} pairs where b is the number of bits in an integer). However, in practice, this doesn't cause any problems: Working with a 32 bit machine, the upper bound is much larger than any heap size that we are ever likely to use. On a 16 bit PC, the upper bound on heap size is much lower (a maximum of 32,768 pairs) but the PC architecture doesn't have enough memory to support a larger heap.

  The first and second components of each pair are stored in two arrays of `Cell`s called `heapFst` and `heapSnd` each of which contains `heapSize` elements. The use of two arrays rather than a single array containing both components of each pair was another concession to the PC architecture which limits the maximum size of an array (at least, if we hope to avoid expensive index calculations) to 64K. By using two arrays, the PC architecture can support a 128K heap containing the maximum of 32,768 pairs, each requiring 4 bytes. However, the decision to use two arrays may seem rather shortsighted because it makes the allocation of variable size heap objects much more difficult; this might be an obstacle to efficient implementation of arrays or records. In fact, as we describe in the next section, the constraints imposed by the garbage collector in the current design would still have made it almost impossible to support variable length allocation.

  Because pairs are represented by negative Cell values, we use two variables `heapTopFst` and `heapTopSnd` to point to the locations immediately above the two arrays (i.e. to `heapFst+heapSize` and `heapSnd+heapSize`, respectively). The `fst` and `snd` selectors are implemented by macros:

```
#define fst(c)  heapTopFst[c]
#define snd(c)  heapTopSnd[c]
```

so that the negative index values for c find their way back to the appropriate elements in the array. Since the Gofer heap is quite heavily used, it would probably be useful to store the variables **heapTopFst** and **heapTopSnd** in global registers.

The default heap size setting for a particular machine can be changed using a command line option, but the current implementation does not make any provision for expanding the heap while the interpreter is running. This might be useful, for example, to allow the user to request an increase in heap size, or even to permit automatic heap expansion if garbage collection fails to reclaim sufficient space. There is no reason why this feature could not be added but this would only be of use on larger systems; the default heap size settings for older PCs are already set to the maximum possible values.

- The zero value is used to represent the **NIL** value, making tests for the empty list very fast.

- **INDIRECT** and **INDIRECT1**, symbolic names for the integers 1 and 2, are used in the implementation of indirection nodes. See Section 9.1 for more details.

- Values in the range **[1..BCSTAG-1]** are used as tags in the **fst** component of a pair representing a 'boxed' value. The following table lists the boxed values used in the current implementation. The right hand column gives the type of value that is stored in the **snd** component of these boxed values:

VARIDCELL	Identifier variable	Text
VAROPCELL	Operator variable	Text
DICTVAR	Dictionary variable	Text
CONIDCELL	Identifier constructor	Text
CONOPCELL	Operator constructor	Text
STRCELL	String literal	Text
INTCELL	Integer literal	Int
ADDPAT	(v+k) pattern	Int
MULPAT	(c*v) pattern	Int
DICTCELL	Dictionary	Dict
FILECELL	Input file no.	Int
FLOATCELL	Floating pt	Float

  One of the main purposes of a tag value is to indicate that the **snd** component of a pair should be ignored by the garbage collector.

- Values in the range **[BCSTAG..SPECMIN-1]** are used as tags in boxed values whose **snd** component is another **Cell**. The **FROMQUAL** value is a simple example of this with a pair containing the pattern and expression for a generator as its **snd** component.

- Values in the range **[SPECMIN..TUPMIN-1]** are used for special cell values such as **UNIT**, **WILDCARD** and **STAR**.

- The first **NUM_TUPLES** values in **[TUPMIN..]** are used to represent tuple type and value constructors. The default setting of **NUM_TUPLES** in **prelude.h** allows for tuples with up to 100 components. In practice, it is very rare to find Gofer programs that use tuples with more than 6 components.

- The first **NUM_OFFSETS** values in **[OFFMIN..]** are used to represent offsets. The default setting of **NUM_OFFSETS** to **1024** in **prelude.h** allows for extensive use of pattern matching and local variables in function definitions, and for up to **1024** different polymorphic type variables in inferred types. Examples exceeding this limit are only usually of interest as pathological examples.

- The first **NUM_TYCON** values in **[TYCMIN..]** are used to represent type constructors. The **tycon** selector is implemented using the macro:

```
#define tycon(tc) tabTycon[(tc)-TYCMIN]
```

  where **tabTycon** is an array with **NUM_TYCON** structures containing the fields described in the previous section.

- The first **NUM_NAME** values in **[NAMEMIN..]** are used to represent names. The **name** selector is implemented using an array of structures in a similar way to the **tycon** selector described above.

- The first **NUM_SELECTS** values in **[SELMIN..]** represent dictionary selectors.

- The first **NUM_INSTS** values in **[INSTMIN..]** correspond to instances of a type class. The **inst** selector is implemented by using **Cell** values to index the elements in an array of structures.

- The first **NUM_CLASSES** values in **[CLASSMIN..]** are used to represent classes, with the **class** selector implemented in similar way to **tycon**, **name** and **inst**.

- The first **NUM_CHARS** values in **[CHARMIN..]** are used to represent character values. The default value of **NUM_CHARS** allows for 256 characters, enough to support the full character set of most current machines.

- The remaining values not covered by the above, all in the range `[INTMIN..INTMAX]` are used to represent 'small' integers as unboxed values. In practice, the `INTMIN` value used here tends to be quite small, while `INTMAX` is the largest positive integer value. As a result, the range of small integers is actually quite large, not much less than the range of integers that can be represented using $b - 1$ bits, assuming b bits in every `Cell` value. The midpoint of this range is used as a representation for zero:

```
#define INTZERO (INTMIN/2 + INTMAX/2)
```

  This makes it possible to encode many integer values as `Cells` without requiring any additional storage. Integer values that cannot be represented in this way must be stored as boxed values, i.e. as a pair with first component `INTCELL` and a second component containing the integer.

  Since many programs use only small integer values, this representation gives a noticeable reduction in the amount of space used by programs. On the other hand, simple arithmetic operations on integers are more expensive to implement because they have to be written to deal with both boxed and unboxed integer values. In an interpreter like Gofer there are many more significant performance overheads to worry about. However, this representation would not be suitable for other systems where execution speed is an issue.

4.2.3 Garbage collection

We have already described how new pair cells are allocated by calling the `pair()` function. But what happens when all of the `heapSize` pairs in the heap have been allocated? Like many other systems, Gofer relies on the assumption that, by the time this happens, many of the pairs allocated earlier in the computation will no longer be required. A *garbage collector* is used to determine which parts of the heap can be reused, linking them together as a list of free pairs called the `freeList`. New pair cells can then be allocated by removing an element from the free list. When the free list becomes empty, the garbage collector is called once again, to build a new free list, and the allocator continues as before. Although it is possible for the heap to become full, most of the time, the garbage collector helps to maintain the illusion that there is no limit on the number of pairs that can be allocated.

Automatic garbage collection is particularly convenient because it frees the programmer from the responsibility of deciding when a particular piece of storage can safely be reused, and avoids hard to locate errors that can occur when a programmer makes the wrong decision. Indeed, it has even been argued that the use of garbage collection can be faster than other forms of memory allocation [2], although we would not attempt to make any such claims for the particular garbage collection system used in Gofer.

Although there is no need for the programmer to specify when a particular section of storage should be reused, it is important to ensure that the garbage collector will be able to identify all of the parts of the heap that are still in use. This is achieved by listing a collection of *roots*, i.e. pointers to data structures in the heap that must be preserved by the garbage collector. Roots come from several different sources. For example, the `type` and `defn` fields in the structure describing a `Name` value, local variables used to hold intermediate results, and global variables pointing to list of definitions in a particular input program, must all be preserved by the garbage collector. If they are not, the program will behave badly when the parts of the heap that these variables point to are overwritten with new values. Each component of the system includes a control function (see Section 3.2) that responds to the `MARK` message by marking all of the roots relevant to that part of the system. The garbage collector uses this by calling `everybody(MARK)`. Any part of the heap that is not marked will be reused. For example, it is important to realize that, if a component is modified to use a new global variable, `newList`, pointing to a list of values in the heap, then the control function for that component must be modified to include the line:

```
mark(newList);
```

It is also sensible to ensure that `newList` is initialized and reset to a suitable value (usually `NIL`) in response to the `INSTALL` and `RESET` messages, respectively. Failure to do this may mean that the storage pointed to by `newList` is retained for longer than it is actually required.

Everything that we have described so far is relatively straightforward. We have hinted at some specifics of the algorithm used to allocate new pairs, and to collect unused cells using the `freeList`; this is an example of a *mark-scan* garbage collector. However, compared with the garbage collection algorithms used in other systems, our mark-scan collector has some serious disadvantages:

- The time taken to build the new `freeList` after

all the cells have been marked is proportional to the total size of the heap, not to the size of the heap that is actually in use, which is often much smaller.

- Allocating new pairs from a free list (including a test to make sure that the list is non-empty) is expensive, compared with alternative schemes where new pairs are allocated from a contiguous array of cells, with a heap pointer that identifies the next free heap location. Testing for heap overflow requires a comparison of the heap pointer with a pointer to the end of the heap. In addition, it is often possible to allocate several new pairs at a time, using only a single heap overflow test.

- An additional benefit of the approach described above is that it allows the allocation of different sized heap objects. For example, to allocate an array of 100 cells, we simply need to increment the heap pointer by 100, checking first for overflow of course. In theory, a mark-scan collector could also be used to allocate variable sized heap objects. However, in practice, this often fails because the free space may become fragmented, making it inefficient, and sometimes impossible, to allocate a block of memory of the right size, even when the total amount of free memory is large enough.

These problems can be avoided by using a garbage collector that is allowed to move objects during garbage collection, typically using a *compaction* algorithm, that avoids any heap fragmentation. For this to work correctly, we need to to ensure that the garbage collector will update any pointers to the old position of an object to reflect its new position.

Unfortunately, it is not possible to ensure this property for the Gofer garbage collector; to allow the heap to be used in arbitrary C functions in the system, the garbage collector also uses the C calling stack as a source of roots. To see why this is useful, consider the following function, taken from **storage.c**, which can be used to allocate a list containing a given number of copies of a particular **Cell** value:

```
List copy(n,x)
Int  n;
Cell x; {
    List xs = NIL;
    while (0<n--)
        xs = cons(x,xs);
    return xs;
}
```

Suppose now that a garbage collection occurs part way through this routine, for example, when only half of the required list has been constructed. Clearly, it is important to ensure that the current value of **xs** is preserved by the garbage collector. Similarly, the value of the **x** parameter may need to preserved by the garbage collector; in this particular case, this is only necessary if the garbage collector is invoked on the first call to the **cons** function—after that, the value pointed to by **x** would have been included, and hence marked, as part of the data structure pointed to by **xs**. The easiest way for the garbage collector to find these values is to look for them on the C calling stack. Of course, the C stack also contains several other kinds of values including, for example, other function parameters like **n**, temporary values and function return addresses. When the garbage collector encounters a value on the C stack that looks like a reference to a part of the heap[4], it must treat that value as a root for the garbage collector. On the other hand, it should not attempt to change the value on the stack, allowing for the possibility that it might actually be some other kind of value which just happens to fall in the same range as a heap reference. For example, modifying a return address is likely to cause the program to branch to the wrong location at some later stage, with unpredictable, and possibly disastrous effects.

In this particular example, it would have been possible to change the definition of **copy()** to ensure that there were enough free cells available before attempting to construct the result list, avoiding the chance of a garbage collection during the construction of the result list. This is not a general solution; in some cases, it is impossible, or at best, very expensive, to calculate an upper bound on the amount of storage that a particular C function will allocate. Furthermore, it requires the programmer to take more responsibility[5] for storage allocation, complicating the code that has to be written.

There are other alternatives, using extra code in the definition of C functions, for example, to label intermediate values, to avoid the need for garbage collection off the C stack. However, with many of these schemes, we soon find ourselves writing the C equivalent of G-code, a low-level assembly language (see Section 9.2) to ensure proper marking of roots. This

[4] The predicate **isGenPair()** is used as a more accurate alternative to **isPair()** to test for genuine pair cell values, i.e. excluding any **Cell** values less than **-heapSize**.

[5] It's not that we don't trust the programmer (in this case, one and the same person!); we simply recognize that it is all too easy to make errors in memory allocation, and often very difficult to detect and correct such mistakes.

would not be so bad if the `copy()` function was one of the most complex functions that we had to deal with; unfortunately, it is one of the very simplest.

As if the limitations of the garbage collector that we have already discussed are not enough, there are some further problems that we should mention.

- First, the garbage collector is *conservative*, in the sense that it may preserve parts of the heap that are not actually needed [59, 9]. This happens when the garbage collector misinterprets a value on the C stack as a pointer into the heap. In some situations, this causes a space-leak, preserving large data structures long beyond the point where they are actually needed. Fortunately, this does not seem to cause too many problems in practice, except on some occasions when the heap is already almost full to capacity.

- Another major problem that we have skimmed over until now is that accessing values from the C stack is inherently unportable, and generally speaking, very poor C programming practice. The standard version of the garbage collector relies on the assumption that the C stack can be treated as a contiguous array of equally sized `Cell` values. This is true for many systems, from small PCs to powerful workstations, but there are also some examples for which these assumptions are not valid. For example, in a port of the Gofer system to run under Acorn's RISCOS operating system, Bryan Scattergood had to modify the garbage collector to deal with a stack implemented by a linked list of frames. As another example, in the development of MacGofer, Kevin Hammond had to modify other parts of the system to accommodate the use of an 80-bit (10 byte) representation for floating point values that does not fit directly into the 4 byte units used for `Cell` values. As another illustration of the kind of portability problems that we have run into, it is common for optimizing C compilers to place temporary values in registers. In some cases, and despite our efforts to avoid the problem, the values in registers are hidden from the garbage collector, even when they should be treated as roots. For example, on a Sun workstation, the `compiler.c` part of the Gofer system, which makes heavy use of the garbage collected heap, is usually compiled without optimization to avoid exactly this kind of problem.

In conclusion, the garbage collection system is an enormous pain, and a significant source of portability problems. More than any other part of the system, it demonstrates some of the compromises that have been necessary in the development of Gofer. Happily, in practice, the garbage collector is usually very reliable. For example, we have run large Gofer programs taking several days to execute and requiring several tens of thousands of garbage collections without any apparent problems. In light of the discussion above, we consider ourselves very fortunate that the garbage collector works as well as it does! The current implementation allows us almost transparent use of the heap in C function definitions. Without this, we believe that it would have been significantly more difficult to write and maintain large sections of the system, particularly the supercombinator compiler in `compiler.c`. At the same time, the garbage collector does still cause some problems, in particular, prohibiting useful extensions such as the allocation of variable sized blocks of memory. Perhaps these problems can be avoided in future versions of the system by switching to a language that includes built-in support for garbage collection. Alternatively, it might be possible to use a language, for example C++, allowing special treatment of `Cell` values to ensure proper garbage collection; it remains to see what kind of performance overheads this approach might carry.

4.3 The `Module` datatype

As we have already described in Section 2, the Gofer system allows the user to load a sequence of script files, each layered on top of the definitions in previously loaded files. Apart from `Cell` values which have been described in detail in the previous section, most other values, for example, `Name`s, `Class`es and `Inst`s, are allocated from fixed tables with 'high water mark' values, for example, `nameHw`, `classHw`, and `instHw`, respectively, pointing to the next free position in each table.

The `Module` datatype is used to record the number of script files that have currently been loaded. More importantly, for each script file loaded, the system maintains a table recording the values of the various high water mark variables immediately before each file was loaded. These values are saved by a call to the function:

```
Module startNewModule  Args((Void));
```

which returns a module number for the new script file. Later, the system can restore the storage allocation to its position immediately before this module was loaded by a call to the function:

```
Void   dropModulesFrom Args((Module));
```

In addition to resetting the values of the various high water marks, this function also requires some small adjustments to some additional data structures, such as the hash table used to locate **Name** values. Together, these functions provide a simple checkpointing mechanism. For example, by calling **startNewModule()** before reading a new script file, or before reading an expression to be evaluated, it is easy to restore the original state if an error occurs, or once the expression has been evaluated, respectively.

A more sophisticated scheme for the allocation and recovery of values from tables would be necessary if the system did not assume a strict dependency between the script files in the order that they are loaded.

4.4 The Gofer stack

Several parts of the Gofer system make use of a stack of **Cell** values:

- The parser uses a stack to record intermediate values corresponding to fragments of input programs (Section 5.3).
- The algorithm for calculating strongly connected components of a program dependency graph during static analysis uses a stack to record nodes in the graph that have already been visited (Section 6.2).
- The Gofer abstract machine uses a stack to hold intermediate values and function arguments during program execution (Section 9).

All of these applications are implemented using the same stack data structure, with the most important operators as follows:

```
Void clearStack Args((Void));
Void push       Args((Cell));
Cell pop        Args((Void));
Cell top        Args((Void));
Cell pushed     Args((Int));
```

The use of these functions should be obvious from their names and types, except perhaps the last; **pushed(n)** returns the value **n** positions from the top of the stack. The expression **top()** is equivalent to **pushed(0)**. In fact, these operators are implemented by preprocessor macros rather than function calls. Note that all of the values on the stack are treated as roots for the garbage collector; this is important to ensure that the stack is preserved if a garbage collection occurs during parsing, static analysis or program execution.

5 Lexical analysis and parsing

The **input.c** program is used to read and parse both the definitions in Gofer scripts and the commands and expressions that are entered into the interpreter. Combining a hand-written lexical analyzer and a **yacc** generated parser [19], this part of the system draws on standard techniques of compiler construction.

5.1 Lexical analysis

Most of the code for lexical analysis is included in the file **input.c**. A two character lookahead, represented by the two variables **c0** and **c1** is sufficient to identify all of the tokens used by the Gofer system. At each stage, the current position in the input text is maintained in the **row** and **column** variables. Column positions are used chiefly to keep track of indentation which is significant in the implementation of the layout rule. Row values, corresponding to line numbers in Gofer programs, are recorded at various points in the parsed form of input expressions so that errors detected at later stages in the system can be referred back to the appropriate point in the source code.

The lexical analyzer can be initialized to take its input from the console, a project file or a Gofer script file using one of the three functions:

```
Void consoleInput Args((String));
Void projInput    Args((String));
Void fileInput    Args((String, Long));
```

For console input, the **String** argument is used as the Gofer prompt. In the remaining two cases, the **String** argument is the name of the file to be read. The second argument to **fileInput** gives the length of the input file (if known) and is used to determine the portion of the file that has been read at any point during parsing.

The lexical analyzer is able to read Gofer script files using both the standard and literate styles. In the latter case, program lines in input files must begin with a '>' character in the first column; all other lines are treated as comments. This convention, described more fully in [16, Appendix c], is widely used because it allows program text to be freely mixed with its documentation. The ability to read literate scripts directly without preprocessing does not significantly increase the complexity of the lexical analyzer.

For each token encountered, the lexical analyzer returns a code to the parser to indicate what kind of token has been read. For example, **NUMLIT** for numeric literals, **VARID** for a variable name and **COCO** for a **::**

symbol. Distinguishing between different types of token is all that the parser needs to process the input, but many of these different kinds of tokens have corresponding attributes that are needed in later stages; obvious examples include the value of a numeric literal or the name of a variable, both of which can be represented as `Cells`. These additional properties are passed to the parser in the variable `yylval`, the standard convention for `yacc`-generated parsers. In some cases, notably for the `=` and `::` symbols, the value returned in `yylval` is the line number on which the symbol occurs. These line numbers are used in later stages of the system to locate the (approximate) source of an error.

5.2 The Gofer grammar

The grammar for Gofer programs and expressions is defined by the file `parser.y`, used as input to `yacc` to produce `parser.c` which is `#included` as part of `input.c`. For convenience, the grammar includes productions for the module headers, including import and export declarations, that are used in full Haskell programs. This means that it is often possible to use a file containing one or more Haskell module definitions as input to Gofer without modification. However, other than checking for syntax errors, these module headers are completely ignored.

The Gofer grammar is closely based on the definition of Haskell, although it is a little more liberal in some respects. As a simple example, patterns in Gofer programs are actually parsed as expressions. This helps to avoid unnecessary conflicts and ambiguities in the grammar. Later, in `static.c`, static checks are used to ensure that the expressions parsed are valid patterns. A similar technique is used to parse certain forms of type expression. The following examples show that several tokens of lookahead may sometimes be necessary to determine whether a given type expression includes a (type class) context:

```
T a -> T b    vs.   C a => T a
(T a, T b)    vs.   (C a, C b) => a -> b.
```

Since `yacc` grammars only allow a single lookahead token, we deal with this problem by parsing the context part of a type expression as if it were just a type. Later, if a `=>` symbol is detected, we use a simple static check to ensure that the value parsed as a type can instead be treated as a context. For example, the expression `Int => Bool` would be accepted by the parser, but rejected by the static check. On the other hand, the current grammar will accept a type expression of the form `((Eq a)) => a -> Bool`, even though a strict adherence to the Haskell grammar would prohibit the repeated parentheses around the predicate `Eq a`.

There are also some small differences in the interpretation of certain expressions. For example, in Haskell, the two expressions:

```
let (n+1) = 43 in n
let n+1   = 43 in n
```

have different meanings; the first evaluates to `42` while the local definition in the second introduces a new value for the `(+)` operator, not for `n`, and hence returns whatever value was bound to `n` in the enclosing environment (or a compile-time error if no such binding exists). In Gofer, the second interpretation is used for both expressions; `(n+k)` patterns cannot be used at the top-level of a pattern binding, with or without the parentheses.

Another interesting aspect of the Gofer parser is the way that expressions involving infix operators are treated. The Haskell grammar [16, Appendix B] defines the syntax for infix operators using a complex family of productions indexed by precedence and fixity values. Implementing this directly as a `yacc` grammar would cause a significant increase in the size of the grammar. Instead, the Gofer parser reads expressions involving infix operators as a sequence of expressions, separated by operator symbols, without any further processing. When the whole sequence has been read, it is passed as an argument to the `tidyInfix` function at the end of `parser.y` that uses a simple shift-reduce parser to determine the correct interpretation of the expression. In a system supporting Haskell-style `import` declarations, this tidying process could profitably be delayed until the interface files for imported modules have been read, providing the fixities for imported infix operators. Apart from simplifying the `yacc` grammar, this approach would also make it very easy to extend the range of precedence values that can be assigned to infix operators. Following Haskell, the current implementation allows only single digit precedences.

Another small difference between the Haskell and Gofer grammars is in the treatment of the prefix unary minus operator. The definition of Haskell, requires unary minus to be treated as having a precedence value of 6, the same precedence used for binary addition and subtraction. In Gofer, unary minus binds more tightly than any infix operator, but less tightly than function application. This rarely causes any difficulties in practice, although we will probably change the Gofer grammar at some point in the future to be consistent with Haskell.

5.3 The shadow stack

During parsing, the Gofer system is constantly using the heap to build expression trees corresponding to parsed fragments of the source program. It is important to ensure that all of these intermediate values are preserved if a garbage collection occurs during parsing. In fact, parsers generated by `yacc` already maintain a stack of such values for internal use; if we could make the values on the `yacc` stack known to the garbage collector, then we wouldn't have to worry about loosing the parse trees for intermediate fragments during garbage collections.

Unfortunately, because the `yacc` stack is intended only for internal use, there is no officially documented way to access these values. Instead, using some insight into the way that `yacc` generated parsers work, the current implementation uses the Gofer stack to simulate the state of the internal parser stack. This duplication of effort has obvious disadvantages, but does at least ensure that the garbage collector will preserve intermediate values during parsing in a relatively portable manner.

We refer to this process as *shadowing* the `yacc` stack. The first step in the shadowing process is to ensure that the lexical analyzer pushes the appropriate value onto the Gofer stack every time it encounters a token in the input file. Most of the remaining work is taken care of by the `gcShadow()` function defined in `parser.y`. This is used in productions such as:

```
ctype : ctype atype {$$=gc2(ap($1,$2));}
```

An expression of the form `gc2(e)` is just an abbreviation for `gcShadow(2,e)`, the purpose of which is to remove the top two elements of the stack and replace them with the new expression `e`. In fact, the definition of `gcShadow` is a little more complicated since we need to take account of situations where the stack already contains an extra lookahead token. We refer the reader to `parser.y` for more details.

5.4 Implementing the layout rule

The Haskell layout rule, also adopted by Gofer, allows a programmer to use layout and indentation to reflect the structure of a program in a concise and natural manner. In effect, the layout rule works by automatically inserting `{`, `;` and `}` tokens at certain points in the input stream. For example, the case expression:

```
case expr of True  -> branch1
             False -> branch2
```

is transformed, by the insertion of these additional tokens, to:

```
case expr of { True  -> branch1;
               False -> branch2
             }
```

We refer the reader to [24, Chapter 13] and [16, Section 1.5] for a more complete description of the layout rule and its use.

The implementation of the layout rule is a little tricky and relies quite heavily on the error recovery features of `yacc` and on some mildly complicated interactions between the lexer and parser. The details are a little too technical for this report and we refer the reader to the code for `input.c` and `parser.y`, in particular, the productions for `close` and `close1`, for more information.

5.5 Abstract syntax for parsed values

To explain the role of the different components in the Gofer system, we will describe the internal representation of Gofer programs at each stage using a sequence of simple grammars. The purpose of the lexical analyzer and parser is to translate input programs in the concrete syntax of Gofer into the first of these abstract syntaxes.

The main result of parsing a Gofer program is a list of equations mixed with type signature declarations. Using the grammar in Figure 4, this can be described as a list of the form [*Eqn*]. Note that, following the Gofer syntax for lists, we use the notation [...] to indicate a list of values, not an optional item as in some BNF-style notations. The *Line* type used here indicates an integer value corresponding to a line number in the source program. We have used the names `VAR` and `CON` to indicate variables and constructor identifiers respectively. In fact, the parser and lexer actually go a step further, making a distinction between the case when an identifier is written with applicative/prefix syntax (`VARID/CONID`) and the case when infix syntax is used (`VAROP/CONOP`).

There are two useful observations to make about this grammar. First, as suggested in earlier comments, the grammar does not distinguish between expressions and patterns; for example, the fact that `WILDCARD` is not permitted in an expression or that `LETREC` clauses cannot appear in patterns is not captured by the grammar. These errors will not be detected until later, during the static analysis in `static.c` (Section 6). Second, the grammar is quite complex; while it omits much that is purely syntactic such as comments and parentheses used for grouping,

```
Eqn     ::= SIGDECL (Line, [Var], SigType)
          | (Expr, Rhs)

Expr    ::= VAR Text
          | CON Text
          | AP (Expr, Expr)
          | Const
          | COND (Expr, Expr, Expr)
          | FINLIST [Expr]
          | LETREC ([Eqn], Expr)
          | LAMBDA Alt
          | COMP Comp
          | RUNST Expr
          | ESIGN (Expr, SigType)
          | CASE (Expr, [(Pat, Rhs)])
          | ASPAT (Var, Pat)
          | LAZYPAT Pat
          | WILDCARD

Const   ::= UNIT
          | TUPLE Int
          | STRCELL Text
          | CHARCELL Char
          | FLOATCELL Float
          | INTCELL Int

Comp    ::= (Expr, [Qual])

Qual    ::= FROMQUAL (Pat, Expr)
          | QWHERE [Eqn]
          | BOOLQUAL Expr

Pat     ::= Expr

Alt     ::= ([Pat], Rhs)

Rhs     ::= GUARDED [(Line, (Expr, Expr))]
          | LETREC ([Eqn], Rhs)
          | (Line, Expr)

SigType ::= QUAL ([Pred], Type)
          | Type

Type    ::= ARROW
          | TUPLE
          | UNIT
          | LIST
          | VARID
          | CONID
          | AP (Type, Type)

Pred    ::= AP (CONID, Type)
          | AP (Pred, Type)
```

Figure 4: Abstract syntax for parsed terms and types

it is still quite close to the input language. Although it makes the implementation a little more complicated, keeping the input in this form makes it easier to give accurate error messages during static analysis and type checking. That said, there are a few examples where the special syntax used in Gofer programs is not preserved. The following table shows the translations used for the special syntax for right sections, unary minus and arithmetic sequences:

Source	Translation
`('op' a)`	`flip a`
`(- a)`	`negate a`
`[a..b]`	`enumFromTo a b`
`[a,b..]`	`enumFromThen a b`
`[a..]`	`enumFrom a`
`[a,b..c]`	`enumFromThenTo a b c`

Apart from the list of equations and type signatures that are returned when the end of the input file is encountered, the parser also processes several other kinds of declaration:

- **type** and **data** definitions: For each type constructor definition, the **tyconDefn()** function is used to allocate a new **Tycon** and to save the details for further processing during static analysis.
- **class** and **instance** declarations: In a similar way, new **Class** and **Inst** values are allocated using **classDefn()** and **instDefn()** to record the details for each **class** and **instance** declaration in the input program, respectively.
- Primitive declarations: Bindings of variable names to internal primitives are gathered together in a list **primDefns** during parsing using the **primDefn()** function. Later, as part of the static analysis carried out in **static.c**, new **Name** values are allocated for each named primitive.
- Fixity declarations: Precedence values and associativities for infix operator symbols are entered into the tables used by the expression parser as soon as the fixity declarations are encountered.

Note that these declarations can only appear at the top-level of an input program.

6 Static analysis

The main purpose of the code in **static.c** is to carry out static checks and analyzes on parsed programs and expressions before they are passed on to the type checker. Most of these tasks are so mundane that we

often don't think about them explicitly when writing programs for Gofer. For example, consider a program containing a datatype definition of the form:

```
data T a b c = C | D t1 | E t2 t3
```

where `t1`, `t2` and `t3` are some type expressions. To ensure that this definition is valid we need to carry out the following tests:

- Check that there is no previous definition for `T`, either as a type constructor or as a type class.
- Check the format of the left hand side. The parser ensures that the arguments to `T` are simple type variables, but an additional check is needed to ensure that there are no repeated variables.
- Check that the type expressions appearing on the right hand side are well-formed. In particular, we need to ensure that:
  - The only type variables involved are those on the left hand side of the definition.
  - All of the type constructors referred to on the right hand side are defined somewhere in the current program.
  - The types on the right hand side are well-kinded. For example, this prevents any attempts to supply a type constructor with too many arguments. This requires a form of *kind inference* and will be described in Section 7.7.
- Add new `Name` values with suitable types and arities for each of the constructor functions `C`, `D` and `E` defined by the right hand side. In addition, we need to ensure that there are no previous definitions for these functions in another datatype definition.

Definitions for type synonyms, classes, instance, primitive, function and operator fixities are also subjected to static checks of a similar nature. Verifying these conditions for input programs allows the system to give early detection of simple program errors. In addition, it simplifies the code in later sections of the compiler. For example, there is no need to deal with unbound variables during type checking or code generation.

The Gofer system allows the definitions in a script to be placed in any order[6]. As a result, many of these static checks must be delayed until the whole script file has been parsed.

Although the full list of static checks is quite long, most of them quite easy to implement. For the remainder of this section, we will concentrate on two particular features; the translation of parsed terms and types to a form that is suitable for the type checker, and the use of dependency analysis. Strictly speaking, the kind inference mechanisms used in Gofer are also part of the static analysis. However, the implementation of these functions in `kind.c` shares some code and data structures with the main type checker and, therefore, we postpone further discussion of this to Section 7.7.

6.1 Translation of parsed values

In addition to the checks described above, the static analysis component of the Gofer system also translates parsed types and declarations into a slightly different form. Most of the changes are motivated by the need to avoid unnecessary work in later stages. For example, identifiers corresponding to constants such as `Name`s, `Tycon`s or `Class`es in input terms are replaced with the corresponding values in translated terms, avoiding the need for further symbol table lookups.

6.1.1 Translation of parsed types

Most of the work of the static analysis routines is accomplished by 'walking' the structure of input terms, types and declarations. For example, the analysis of a parsed type expression produces a result of the form described by the grammar in Figure 5. Notice that occurrences of `CONID` cells in the input grammar have now been replaced by the `Class` or `Tycon` values that they refer to. In a similar way, the type variables represented by `VARID` cells in parsed types are replaced by numbered `Offset`s. The presence of type variables, signaling a polymorphic type, is represented by `POLYTYPE` values, with the *Sig* field used to record the kinds of polymorphic type variables[7].

A full description of the representation of polymorphic types requires a fairly good understanding of the technical issues discussed in [33]. We will illustrate the main ideas by describing the representation of the Gofer type `a -> m a`. Following the Haskell convention that type variables are implicitly bound by an outermost universal quantifier, this corresponds to

[6] The placement of fixity declarations *does* have an effect on the way that programs are parsed. Also, the ordering of definitions does affect the order in which errors are reported to the user. However, it does not have any effect on the semantics of error-free programs.

[7] During the preliminary stages of static analysis, the *Sig* field is used to record the number of type variables, but not their kinds.

```
SigType   ::=  POLYTYPE (Sig, Type)
           |   QualType

QualType  ::=  QUAL ([Pred], Type)
           |   Type

Type      ::=  ARROW
           |   TUPLE
           |   UNIT
           |   LIST
           |   OFFSET Int
           |   TYCON Tycon
           |   AP (Type, Type)

Pred      ::=  AP (Class, Type)
           |   AP (Pred, Type)
```

Figure 5: Representation of types after static analysis

the following type in the underlying formal system:

$$\forall a^*.\forall m^{*\to *}.a \to m\ a.$$

The annotations on the variables a and m specify the kind of constructors that they represent. Treating $\forall$ as a form of λ-binding, this type can be thought of as a function mapping a constructor a of kind $*$ and a constructor m of kind $* \to *$ to a constructor $a \to m\ a$, also of kind $*$. With this interpretation, we choose the kind $* \to (* \to *) \to *$ as a signature for the type `a -> m a`. The full representation of this type can be constructed using the expression:

```
ap(POLYTYPE,
   pair(pair(STAR,           /* Sig  */
             pair(pair(STAR,STAR),
                  STAR)),
        ap(ap(ARROW,         /* Type */
              mkOffset(0)),
           ap(mkOffset(1),
              mkOffset(0)))))
```

Note the use of `mkOffset(0)` and `mkOffset(1)` corresponding to the variables `a` and `m`, respectively. We will see later that this representation makes it particularly easy to instantiate a polymorphic type with new constructor variables of the appropriate kinds.

6.1.2 Translation of parsed equations

Definitions of variables and functions, represented by lists of equations in the output of the parser, are also translated during static analysis. The main change is to group equations into *bindings* each of which takes one of two possible forms:

- A function or variable binding groups together all the equations for a single variable and is represented by a value of the form:

  $$(Var, (Type, [Alt]))$$

  The *Type* value in a function binding is used to record any explicitly declared type for the function named by the *Var* field. If there is no explicit type declaration, a `NIL` value is used in its place. The list of *Alt*ernatives in a binding is obtained by combining the defining equations for the corresponding variable in the original script. Recall that each *Alt* is a pair containing a list of patterns and a right hand side expression. As part of the conversion from equations to bindings, static checks ensure that all of the equations for a particular variable are grouped together in the source file and that the number of argument patterns is the same for all equations (referred to as the *arity* of the variable).

- A pattern binding is used to represent a group of variables that are defined by a single equation whose left hand side is a pattern. Such bindings are represented as values of the form:

  $$([Var], ([Type], (Pat, Rhs)))$$

  The variables defined by a pattern binding are listed in the first component with a corresponding list of explicitly declared types (or `NIL` values, as necessary) in the second. Differing slightly from the definition of Haskell, the Gofer system will reject any pattern bindings that do not define any variables. This was a conscious design decision, intended to prohibit silly pattern bindings like `True = False`; without any variables on the left hand side, there is no way to force evaluation, and hence detect the failure of such pattern matches.

Given an arbitrary binding `b`, the Gofer implementation uses the test `isVar(fst(b))` to distinguish between function and pattern bindings. The description of bindings using nested pairs makes it a little easier to explain how the explicit type information in a binding can be discarded after type checking, for example, by mapping a function binding to a value of the form $(Var, [Alt])$. Of course, since all Gofer data structures are built up using the primitive pairing constructor, this is just a matter of presentation.

```
Binding  ::=  (Var, (Type, [Alt]))
          |   ([Var], ([Type], (Pat,Rhs)))

Expr     ::=  ...
          |   LETREC ([[Binding]],Expr)
          |   ...
          |   NAME Name

Qual     ::=  FROMQUAL (Pat,Expr)
          |   QWHERE [[Binding]]
          |   BOOLQUAL Expr

Rhs      ::=  GUARDED [(Line,(Expr,Expr))]
          |   LETREC [[Binding]] Rhs
          |   (Line, Expr)
```

Figure 6: Grammar of terms after static analysis

Figure 6 summarizes the main changes in the form of parsed terms and declarations. Notice the use of lists of lists of bindings in local definitions, rather than the lists of equations in the original parsed form. This is a result of the dependency analysis, described below. A somewhat smaller change is the introduction of `Name` values, used as replacements for variables that refer to constructor functions, class member functions, or user defined functions in previously loaded script files.

6.2 Dependency analysis

As we have already mentioned, Gofer does not place any restrictions in the ordering of definitions in a script file. However, for the benefit of other parts of the system, particularly kind and type inference, the Gofer system uses information gathered during static analysis to sort these definitions in order of dependency.

For example, given a list of bindings bs, the dependency analysis is used to produce a list of lists of bindings, $[bs_1, \ldots, bs_n]$ such that:

- $bs_1, \ldots, bs_n$ includes exactly the same collection of bindings as bs.
- The bindings in each bs_i are mutually recursive.
- For any variable f defined in bs and referenced in bs_i, the definition of f is included in bs_j for some $j \leq i$.

In practical terms, this means that an expression of the form:

```
let bs in expr
```

can be rewritten using a sequence of nested `let` expressions:

```
let bs1 in
⋮
let bsn in expr
```

The principal motivation for dependency analysis is to enhance polymorphism. For example, without dependency analysis, the type checker would reject the following expression because the use of a single binding group forces a more restrictive type for `id` than we might expect:

```
let  id x   = x
     succ y = id y + 1
in   id True
```

A proper understanding of this relies on the fact that, in the pure Hindley/Milner type system, all calls to functions within a single binding group are assigned the same monomorphic type [1]. Dependency analysis is discussed in more detail in [48, Section 6.2.8] and in [16, Section 4.5.1]. A secondary reason for using dependency analysis in the current implementation is to make Gofer programs easier for the type checker to digest; as we describe in the next section, the Gofer type checker works best when a program can be split into small binding groups.

Dependency analysis can be implemented by finding the strongly-connected components for the dependency graph of a given program and using a topological sort to arrange them in the desired order. In Gofer, the dependency analysis is implemented using a small modified version of a standard algorithm that automatically produces the strongly-connected components in the required order [51, 6]. The core of the algorithm is contained in the file `scc.c`. This file is set up to allow different instances of the dependency analysis algorithm to be generated using the C preprocessor and `#include`ing several copies of the code. Templates in C++, generics in Ada, or functors in Standard ML would have provided more elegant ways to deal with this in other languages.

Dependency analysis is also applied to group mutually recursive class and type definitions together in dependency order prior to kind inference. In previous versions of the Gofer system, up to and including version 2.28b, separate dependency analyses were used for type and class definitions. With an eye to providing better compatibility with Haskell in future releases, the implementation in version 2.30 accommodates arbitrary recursions between class and type definitions, combining the two dependency analyses

in one. Strictly speaking, there is no need for a dependency analysis of this form because the current version of Gofer does not support polymorphic kinds. Nevertheless, we anticipate that this may be a useful addition at some point in the future. In the meantime, we still benefit from the ability to split the kind inference of the declarations in a program into smaller, more manageable, pieces.

7 Type checking/inference

The main task of the type checker is to ensure that every expression, and indeed, every definition in a Gofer program has a type. Sections of a program that cannot be assigned a type are treated as errors. This can often help to detect and locate coding errors. Furthermore, if a given program type checks without producing any error messages, then we can be sure that its execution "will not go wrong" and that run-time type checks can be omitted from the compiled version of the program. Perhaps more importantly, types are useful as a means of describing, documenting, and reasoning about the way that objects are used. This reflects a general philosophy that types are a valuable tool in software development.

In practice, it is not usually necessary to include explicit type information in most parts of a Gofer program. This is possible because the system is able to infer the missing type information. The Gofer type checker has two main roles:

- To calculate types for all of the functions and variables defined in a script file and to ensure that the results are consistent with any explicit type declarations included in the source program.
- To add extra parameters to functions with overloaded types, to be used to pass dictionary values in the implementation of type class overloading.

The first of these can be seen as another part of the static analysis described in the previous section, while the second is the first step in a sequence of program transformations that are needed to convert input code to executable programs. Throughout this report, we persist with the standard practice of referring to this component of the Gofer system as a 'type checker'. In reality, checking plays a secondary role to the more important task of type inference.

The type checker is the largest and most complicated component of the Gofer system. Of course, this is not really too surprising, given that one of the earliest motivations for the development of Gofer was to explore extensions to the type system. The type checker is implemented by the code in `type.c`, but also includes additional sections from the files:

- `subst.c`: Deals with the representation of substitutions and with the unification of types and kinds.
- `kind.c`: Provides functions for kind inference and for checking equalities between kinds during type checking.
- `preds.c`: Provides support for the implementation of type class overloading, including the simplification of predicate sets and the construction of dictionaries.

The Gofer type checker builds on a large body of theoretical work, most of which we will not attempt to describe here. In particular, we will assume some familiarity with the basics of standard type inference algorithms [41, 11] and with the extensions of this work to qualified types [26, 30, 29] and to constructor classes [33], on which the type checker depends for its theoretical basis. To give some historical insight to these references, it took approximately three months to develop the first version of Gofer version 2.xx that was capable of running small programs and included a Hindley/Milner style type checker. As we started to add the extra mechanisms required to support type classes, it became clear that we did not have sufficient understanding of the underlying theory to complete the implementation. A further three months study, resulting in the work described in [26] and later summarized in [30], was necessary before we were able to complete the type checker. In conclusion, you should not expect to understand the full details of the implementation of the Gofer type checker unless you are prepared to spend a considerable amount of time studying and examining it. The information in this section provides only a brief introduction, not a complete description.

Since this section is quite long, we will start with a brief outline of its contents. We begin with a description of the representations used for the 'current substitution' and typing assumptions in Sections 7.1 and 7.2, respectively, and show how they are used to implement generalization in Section 7.3. The main type checking algorithm is described in Section 7.4, while the implementation of overloading is covered by Section 7.5. This includes details about the construction and use of dictionary values. Section 7.6 describes the representation of type checked programs. Kind inference, strictly speaking a part of the static analysis, but implemented using the same ideas as

the type checker, is discussed in Section 7.7. Finally, Section 7.8 suggests some simple experiments for exploring the workings of the type checker.

7.1 The 'current substitution'

We have already described the representation of polymorphic, and possibly qualified, types using the grammar of Figure 5. In particular, polymorphic types usually include `Offset` values corresponding to type variables in the original type expression. For example, a representation for the type of the identity function, $\forall a^*.a \rightarrow a$, can be constructed using the expression:

```
ap(POLYTYPE,
   pair(pair(STAR,STAR),
        ap(ap(ARROW,
              mkOffset(0)),
              mkOffset(0))))
```

If the identity function is applied to a value of type `Int`, then we need to instantiate this type, replacing each occurrence of `mkOffset(0)` with `typeInt`, the `Tycon` representing the type of integers, to obtain:

```
ap(ap(ARROW,typeInt),typeInt)
```

Rather than copying the structure of a polymorphic type expression like this every time it is instantiated, the Gofer type checker adopts a different representation for monomorphic types combining a *skeleton* for the type with an *offset value*. The skeleton captures the basic structure of the type, without specifying what types, if any, the variables that it contains are bound to. For example, the same skeleton,

```
ap(ap(ARROW,mkOffset(0)),mkOffset(0))
```

is used for all instances of the identity function. Notice that this skeleton is taken directly from the original polymorphic type; there is no need for copying. Since every instance uses the same skeleton, we describe this approach as *structure sharing*. The same terminology is used for similar purposes, in the implementation of logic programming languages [10].

The second component in the representation of a monomorphic type is an offset into an array of type variables, each of which is a structure of the form:

```
typedef struct {
    Type bound; /* Skeleton */
    Int  offs;  /* Offset   */
    Kind kind;  /* Kind     */
} Tyvar;
```

This array is described as the *current substitution* since it can be viewed as a mapping from type variables to type expressions. The first two components of a type variable contain another skeleton/offset pair, representing the type that the variable is bound to. Unbound type variables are represented by type variables in which the `bound` field is `NIL`. The third component specifies the kind of the variable; this is set using the information in the *Sig* field of the representation of a polymorphic type. Although we describe the workings of the type checker in terms of type variables and type expressions, we should remember that the system of constructor classes presented in [33] also allows variables and constructors with kinds other than $*$, the kind of types.

For the example above, we would expect the type of the identity function, instantiated bee applied to integer values, to be represented by the skeleton above, combined with an offset value pointing to a type variable with `bound = typeInt` and `kind = STAR`. The value of `offs` does not matter in this case because there are no unbound type variables in the `bound` type.

Type variables in the current substitution are referred to by their position in the array. The expression `tyvar(n)` produces a pointer to the `n`th element of the current substitution. Polymorphic types containing more than one type variable are instantiated using consecutive type variables in the current substitution. For example, if the offset value for the representation of a particular type is `o`, then each occurrence of a cell of the form `mkOffset(i)` in the corresponding skeleton is interpreted as a reference to the type variable pointed to by `tyvar(o+i)`.

In practice, polymorphic types are usually instantiated in several steps, the first of which is to extend the current substitution with the required number of unbound type variables of the appropriate kinds. This process is implemented by the `instantiate()` function which, given a type `t`, returns a skeleton, an offset value corresponding to the first new type variable, and a list of predicates (or rather, skeleton predicates) for overloaded functions in the global variables:

```
Type typeIs;    /* Skeleton   */
Int  typeOff;   /* Offset     */
List predsAre;  /* Predicates */
```

Later, when the function is applied to arguments of a particular type, a *unification* algorithm is used to match the expected argument type against the type of the value that it is actually applied to, binding these variables to appropriate types as necessary.

The representation of the current substitution described in this section makes it possible to instantiate polymorphic types with new type variables very efficiently, without any copying. The biggest problem with this approach is that the maximum number of type variables that can be allocated is limited by the size of the array used to represent the current substitution. For many programs, the number of type variables required to type check each top-level binding group is well within the default setting. However, there are some programs where more type variables are required. Some of these are pathological examples, often used to demonstrate that the complexity of Hindley/Milner typing is exponential in the worst case by defining terms with outrageously high degrees of polymorphism. More important, programs with very large binding groups require large numbers of type variables. The most common source of programs causing this kind of problem are the machine generated parsers produced by systems like Ratatosk [43]. In a change from earlier versions of Gofer, the current distribution allows the size of the current substitution to increase dynamically during type checking (within the limits of available memory, of course) to accommodate such programs.

7.2 Representing assumptions

Another important data structure used by the type checker is a collection of assumptions about the types of the variables that appear in a term. One obvious way to represent type assumptions is to use a list of pairs, each of which gives the name and type of a particular variable. In fact, it turns out to be more convenient to split the type assumptions into two groups:

- λ-bound variables, e.g. variables bound in λ-expressions, as function arguments or in list comprehension generators. These variables can only be used at a single, monomorphic and non-overloaded type. For example, the following definition is not permitted because the λ-bound variable `i` is applied to values of two distinct types on the right hand side:

  ```
  funny i = (i True, i 'a')
  ```

- Let-bound variables, i.e. variables bound by top-level or local definitions. The Hindley-Milner type system used in Gofer allows let-bound variables to be assigned polymorphic types. For example, the let-bound variable `id` in the expression:

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

  has type $\forall a^*.a \rightarrow a$, and hence can be applied to different types of value, for example, booleans and characters, without causing a type error. Note however that, all occurrences of a let-bound variable in its definition must have the same monomorphic type. Hence the following example is *not* permitted:

  ```
  strange  :: a -> Bool
  strange x = strange [x]
  ```

  This restriction is necessary to avoid some of the problems with *polymorphic recursion* and type inference, as described in [1]. However, it is quite likely that future versions of both Haskell and Gofer will be extended to permit a weaker form of polymorphic recursion, requiring an explicit type signature for any let-bound variable that is used in this way.

These two types of variable can be mixed in a single expression by interleaving uses of one with those of the other. For example, in the function definition:

```
member x xs = any isx xs
              where isx y = x==y
```

the variables `member` and `isx` are let-bound, while `x`, `xs` and `y` are λ-bound.

At any stage during type checking, the current assumptions are represented by two lists of lists of (variable, type) pairs:

```
List defnBounds;   /* let-bound    */
List varsBounds;   /* lambda-bound */
```

Using lists of lists like this makes it easy to interleave assumptions about let-bound and λ-bound variables. In fact, these lists are used rather more like a stack with the first elements in each containing assumptions about the most recently bound variables at each stage. For example, during type checking of the expression `x==y` in the definition of `member` above, these lists would be of the form:

```
defnBounds = [[("isx",it)],
              [("member",mt)]]
varsBounds = [[("y",yt)],
              [("xs",xst), ("x",xt)]]
```

for some types, `it`, `mt`, `yt`, `xst` and `xt`. The types bound in assumptions take one of two forms; either a polymorphic type scheme, or a monomorphic type expression, often just a single integer referring to a particular type variable in the current substitution.

The reason for distinguishing between different kinds of variables in the representation of assumptions is that let-bound variables require some special treatment to deal with overloading (recall that λ-bound variables cannot have overloaded types). This will be described in more detail in Section 7.5.2.

The only variables that appear in the assumptions in **defnBounds** and **varsBounds** are those which are bound locally in the current (or enclosing) binding group. The types of for globally defined values, for example, constructor functions, member functions or functions from definitions in earlier binding groups, perhaps in another script file, can be obtained from the **type** field in the corresponding **Name** value. For example, the types of both **any** and **(==)** in the definition of **member** above will be obtained in this way.

In addition to assumptions about the types of bound variables, it is also necessary to keep a list of predicates to capture any class constraints that are required for overloaded functions. The global variable:

```
List preds;
```

is used to store these predicates as a list of triples, each of which contains:

- The skeleton for a predicate, i.e. an expression of the form `C t1 ... tn` where `C` is an `n`-parameter class and `t1`, ..., `tn` are skeletons for each parameter.
- An offset, used to map **Offset** values in the skeleton to particular type variables in the current substitution.
- An expression that can be used to obtain a dictionary for the given predicate. Initially, this expression will be a newly generated dictionary variable which will be inserted at the appropriate point in the translated expression. However, it may be overwritten later if it turns out that the dictionary required can be obtained as a subcomponent of some other dictionary value.

For example, consider the expression **x==y** in the definition above. The type of the **(==)** symbol is:

```
Eq a => a -> a -> Bool
```

When this type is instantiated, we allocate a new type variable **beta** in the current substitution, generate a new dictionary variable **dv** and add the triple **(Eq a, beta, dv)** to **preds**. At the same time, the call to **(==)** in the original version of the program is replaced by **(==) dv**, taking the dictionary parameter as an extra argument.

7.3 Implementing generalization

The description of the current substitution in Section 7.1 focussed on the way that polymorphic types are instantiated. The reverse process, generalizing a monomorphic type to determine the most general type possible, is also an important operation in a type inference system. The generalization of a qualified type ρ in the presence of an assumption set A is represented by the notation $Gen(A, \rho)$ in [29]. Some authors describe this as the *closure* of a type with respect to a particular set of assumptions.

There are two steps in the calculation of the generalization of a type; first we need to mark 'fixed type variables', i.e. those appearing in the current assumption set, then we need to make a copy of the type to be generalized, replacing type variables that are not fixed with generic type variables, represented by **Offset** values.

The marking and copying of types are implemented by functions in **subst.c**, using the **offs** field of each unbound type variable to distinguish between:

- **FIXED_TYVAR**: a type variable that appears in the assumption set.
- **UNUSED_GENERIC**: a type variable, not appearing free in the assumption set, that has not been used as a generic variable.
- **GENERIC+n**: a type variable, not appearing free in the assumption set, that has previously been encountered as a generic variable, represented by **mkOffset(n)**.

The generalization of a type expression, described by a skeleton and offset value, is usually calculated in the following manner. First, the **offs** field for every unbound type variable is set to **UNUSED_GENERIC** using the function **clearMarks()**. All of the types in the current assumption set are marked using the functions **markTyvar()** and **markType()**. Finally, the type to be generalized is copied using the **copyType()** function. Each occurrence of a **FIXED_TYVAR** is represented by the integer value for the variable in the current substitution. Each time a **UNUSED_GENERIC** variable is encountered, the **offs** field is changed to indicate a new generic variable, **GENERIC+n**, and the offset **mkOffset(n)** is returned as the result of this, and any subsequent copies of that variable.

The full generalization operation is implemented by the function **generalise()** in **type.c**, adding any type class constraints and kind annotations necessary for the full representation of a polymorphic type.

7.4 Basic typechecking

The `typeExpr()` function lies at the very heart of the type checker. Taking a line number `l` (for use in error diagnostics, should a problem be detected) and an expression `e` as arguments, `typeExpr` calculates the type of `e`, represented by a (skeleton,offset) pair in the variables `typeIs` and `typeOff` introduced in Section 7.1. In fact, as part of the implementation of overloading, `typeExpr` also returns a translation of the input expression `e` that includes extra dictionary values and parameters.

There are quite a few cases to consider, depending on the form of the expression `e`, and we will only consider a few examples here. One of the simplest cases deals with character constants, represented by `CHARCELL` values:

```
inferType(typeChar,0);
```

The `typeChar` variable used here corresponds to the `Char` type constructor, and is initialized during `typeChecker(INSTALL)`. In fact, the `inferType()` function is a simple macro and the code above expands to:

```
typeIs  = typeChar;
typeOff = 0;
```

Not surprisingly, most of the other cases are more complicated than this! For example, the code for conditionals is as follows:

```
Int beta = newTyvars(1);
check(l,fst3(snd(e)),e,cond,typeBool,0);
check(l,snd3(snd(e)),e,cond,var,beta);
check(l,thd3(snd(e)),e,cond,var,beta);
tyvarType(beta);
```

The first step here is to allocate a new type variable, `beta`, to hold the type of the expression, coinciding with the type of the expression in both the true and false branches of the conditional. The `check()` macro in the next three lines is used to calculate the type of a particular expression, generating an error message if the type obtained cannot be unified with the type in the last two arguments. If the two types cannot be unified, the values in parameters `l` (a line number), `e` (the enclosing expression), and `cond` (the text string, `"conditional"`) are included in the error message displayed by the system to help the programmer locate the source of the problem. The three uses of `check` ensure that the test part of the conditional is a boolean value, and that the two branches have the same type by unifying the types of each with the variable `beta`.

As a final example, the following code is used to type check an expression `e` of the form `LETREC (bs,m)` representing a local definition:

```
enterBindings();
mapProc(typeBindings,fst(snd(e)));
snd(snd(e)) = typeExpr(l,snd(snd(e)));
leaveBindings();
```

The `enterBindings()` call in the first line pushes `NIL` values onto the front of the lists `defnBounds` and `varsBounds`, in preparation for the addition of both let- and λ-bound variables in the typing of the bindings in `bs`. These assumptions are discarded by the call to `leaveBindings()` in the last line, once the type of `e` has been calculated. The intermediate lines are used to type check each group of bindings in `bs`, and then find the type of `m` in the resulting context. As the third line suggests, the translation of `e` is obtained by destructively updating the original expression.

The most complicated part of the type checker is the task of finding the types of a group of mutually recursive definitions in a binding group. This is the purpose of the function `typeBindings()` in the code above. We will delay further discussion of this until Section 7.5.2.

7.5 Overloading

One of the most innovative features in the design of Haskell is its support for user-defined overloading based on the concept of *type classes*, introduced by Wadler and Blott [57]. Extending earlier work by Kaes [36], type classes were proposed as a general method of dealing with examples like equality and arithmetic functions that do not fit comfortably into a simple polymorphic type system, in contrast with the ad-hoc solutions adopted in earlier languages. Using the same ideas and notation, Gofer extends the basic system of type classes used in Haskell in a number of ways, for example, allowing multiple parameter classes, mutually recursive class definitions and constructor classes [33]. The Gofer type system also differs in subtle ways from that of Haskell in its treatment of type classes. This allows (in fact, requires) the use of arbitrary forms of class constraints in type expressions and leads to a particularly simple implementation. In addition, this permits further useful extensions such as the possibility of defining overlapping instances.

Several researchers, including this author, have investigated the theory and formal properties of type classes [8, 30, 29, 36, 44, 45, 50, 53] and there has

also been some experience with practical implementations [14, 28, 46, 5] and applications [27, 38, 13]. Since these topics are so well-documented, we will concentrate here only on the special features in the implementation of Gofer. We will also assume that the reader is familiar with the syntax and use of type classes in Gofer, as described in [24, Chapter 14].

Following the suggestions of Wadler and Blott [57], the implementation of overloading in Gofer makes heavy use of *dictionary* values. Roughly speaking, a dictionary is a tuple of values containing the implementation of overloaded functions corresponding to a particular instance of a class. As a simple example, consider the definition of the `Eq` class in the standard prelude:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y      = not (x == y)
```

and the instance declaration that makes `Int` an instance of this class:

```
instance Eq Int where
  (==) = primEqInt
```

The function `primEqInt` used here is a primitive function of type `Int -> Int -> Bool` that can be used to test two integers for equality. The default definition for the `(/=)` function will be used since no explicit definition is included in the instance declaration. These implementations of `(==)` and `(/=)` can be packaged up together as a dictionary, `eqInt`:

(==)	(/=)
primEqInt	defNeq eqInt

Every dictionary for an instance of the `Eq` class must contain (at least) implementations for the `(==)` and `(/=)` operators as its first and second components, respectively. In general, we will write `(#n d)` to denote the `n`th component of a dictionary `d`. Thus, if `d` is a dictionary for some instance `Eq a`, then the first component, `(#1 d)`, is an equality function of type `a -> a -> Bool`. We can use this to implement overloaded functions like:

```
member          :: Eq a => a -> [a] -> Bool
member x []      = False
member x (y:ys) = x==y || member x ys
```

by adding an extra dictionary parameter and replacing occurrences of the `(==)` operator with appropriate dictionary references:

```
member d x []  = False
member d x (y:ys)
       = (#1 d) x y || member d x ys
```

Note that the same definition works for any type of values, so long as we ensure that the first component of any dictionary passed to `member` includes the appropriate equality function as its first component.

The same ideas are used to handle default definitions, as in the case of the `(/=)` function for integers, implemented by `defNeq eqInt` in the dictionary above. The `defNeq` function referred to here is a general function derived from the default definition in the class declaration, which also uses an additional dictionary parameter to obtain the implementation of the `(==)` operator:

```
defNeq d x y = not ((#1 d) x y)
```

Thus, if `d` is a dictionary for `Eq a` containing a definition of the equality operator `(==)`, then `defNeq d` can be used as an implementation of `(/=)`.

If the type of an expression is known at compile-time, then we can use constant dictionary values rather than adding extra dictionary parameters. For example, the expression `(4 /= 5)` is implemented by translating it to `(#2 eqInt) 4 5` which can be evaluated as follows:

```
(#2 eqInt) 4 5 = defNeq eqInt 4 5
               = not ((#1 eqInt) 4 5)
               = not (primEqInt 4 5)
               = not False
               = True
```

Motivated by the examples above, we will split the remaining description of the implementation of overloading into two pieces:

- The construction of dictionaries using the information provided by class and instance declarations (Section 7.5.1).
- The translation of source programs to include extra parameters for dictionary values (Section 7.5.2).

7.5.1 Dictionary construction

To understand the process of dictionary construction, it is first necessary to explain the representation of dictionaries in a little more detail. In the general case, a dictionary value is represented by an array of `Cell` values of the form illustrated in Figure 7. The storage space for dictionary cells is allocated from

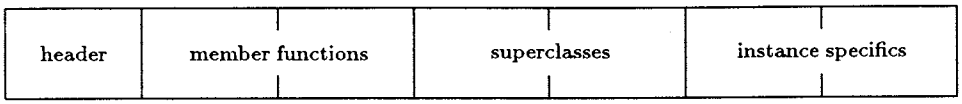


Figure 7: General layout of a dictionary

an array called `tabDict[]`, and individual dictionary values are identified by the position of their first (i.e. header) cell in the array. The nth element in the dictionary array is usually accessed using an expression of the form `dict(n)` which is defined as a preprocessor macro for `tabDict[n]`. Dictionaries can also be represented by values on the heap, storing the corresponding integer offset in the `snd` component of a pair whose `fst` component is `DICTCELL`.

As indicated by Figure 7, dictionary values can be split into four distinct sections:

- The header: The first slot (i.e. index 0) in a dictionary `d`, contains the `Cell` value corresponding to `d`. This fact can be used to distinguish dictionary headers from other cells in the dictionary array. However, its original purpose was to ensure that all heap references to a particular dictionary could share the same `DICTCELL` pair. When a dictionary is first created at some offset `n` in the dictionary array, its header field, `dict(n)` is initialized to `ap(DICTCELL,n)`. All subsequent references to the dictionary share the same value. This use of headers was originally motivated by concerns about the limited heap space in the original PC implementation of Gofer. In retrospect, although it does reduce heap use, the overall saving is probably quite small.

  The header is the only part of a dictionary that is not optional. As such, it plays a useful role by ensuring that the allocation of a new dictionary will always strictly reduce the amount of space remaining for subsequent dictionary allocation. This helps to detect programs that are not well-typed because they require an infinite collection of dictionaries; since the size of the dictionary is fixed at compile-time, only a finite number of dictionaries can be allocated before space is exhausted.

- The member functions: The implementations for each of the member functions for a particular class are stored at the beginning of the corresponding dictionaries, starting at index 1. The ordering of the member functions is loosely determined by the order that they are listed in the class declaration, and hence is the same for all instances of the class. If no explicit member function definition is included in a particular instance declaration then the default definition from the original class definition will be used instead. If there was no default definition either, then the member function slot will be filled with a function that signals a run-time error if it is ever used.

- Superclass dictionaries: As suggested by Wadler and Blott, superclasses can be implemented by storing their dictionaries as components of the dictionaries for immediate subclasses. The ordering of superclass dictionaries is again determined by the form of the class declaration, so that the position of a particular superclass dictionary is the same for all instances of a class. For example, the standard prelude includes a class `Ord` with six member functions whose definition begins:

  ```
  class Eq a => Ord a where
    ...
  ```

  If `d` is a dictionary for some instance `Ord a`, then `(#1 d)` through `(#6 d)` give the implementation for each of the member functions, while `(#7 d)` gives a dictionary for `Eq a`, and hence `(#1 (#7 d))` is an equality function for values of type `a`.

  Note that the superclass/subclass terminology used here is not really appropriate since it is possible to have mutually recursive classes each of which includes the other as a distinct 'superclass' (see [24, Chapter 14] for example). Nevertheless, we continue to use these terms in the same way that they are used in Haskell where a strict hierarchy is enforced.

- Instance specifics: Unlike the other parts of a dictionary, the format of the instance specifics section varies from one instance of a class to the next, depending on which instance declaration was used to construct the dictionary concerned. For example, the standard prelude contains the following definition for the equality on lists:

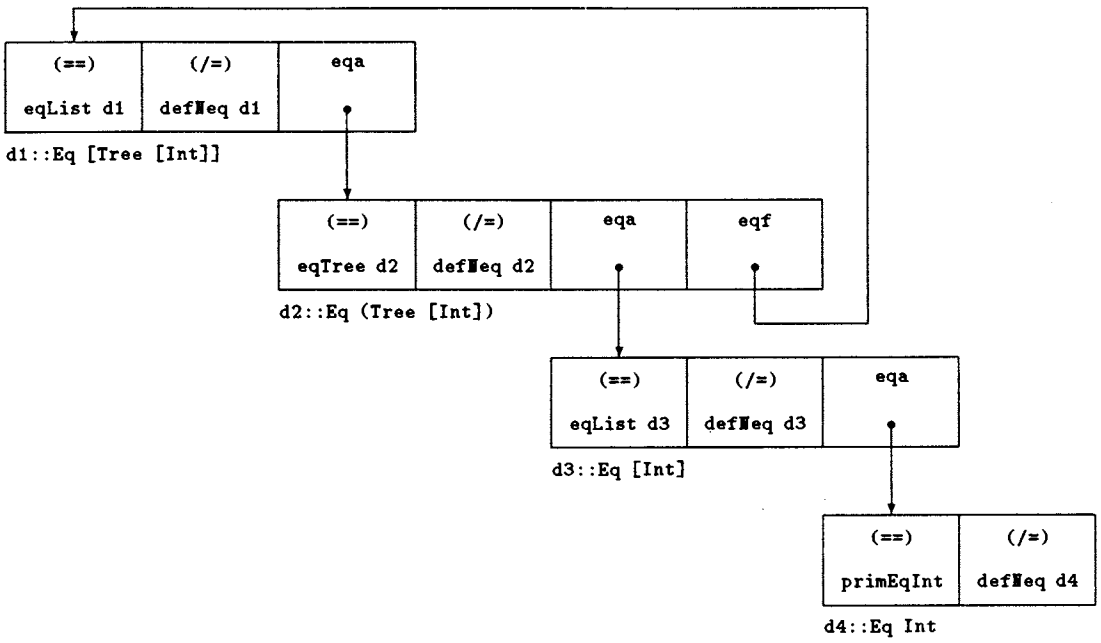


Figure 8: Dictionary structure required for instance `Eq (Tree [Int])`.

```
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x==y && y==ys
  _      == _      = False
```

The first line of the declaration indicates that the equality of lists of type `[a]` will be defined in terms of an equality on values of type `a`. If `d` is a dictionary for `Eq [a]`, it is convenient to store the dictionary for `Eq a` in `d`, for example, in `(#3 d)`. The equality on lists can now be implemented using the function:

```
eqList d []     []     = True
eqList d (x:xs) (y:ys)
   = (#1 (#3 d)) x y && eqList d xs ys
eqList d _      _      = False
```

As another example of the use of instance specifics, consider the definitions:

```
data Tree a = Node a [Tree a]

instance (Eq a, Eq [Tree a])
                 => Eq (Tree a) where
  Node x as == Node y bs
         = x==y && as==bs
```

In this case, two instance specifics will be required, one for each of the constraints in the context of the instance declaration. In particular, if the dictionaries for `Eq a` and `Eq [Tree a]` are stored in positions `#3` and `#4`, respectively, of a dictionary for `Eq (Tree a)`, then the equality on trees might be implemented by:

```
eqTree d (Node x as) (Node y bs)
       = (#1 (#3 d)) x  y  &&
         (#1 (#4 d)) as bs
```

Dictionary values are only constructed for class constraints without any free variables, i.e. for instances of a class in which all of the types and constructors involved are fully determined. For example, if a particular program requires the comparison of two trees of type `Tree [Int]`, the type checker will call the function `makeDict()` in `preds.c` to construct a suitable dictionary that can be inserted in the translated version of the program. It may also be necessary to invoke `makeDict()` recursively to fill the dictionary values in the superclass and instance specific sections of a dictionary. In this particular case, four dictionaries will be required, as illustrated in Figure 8. To avoid unnecessary clutter, dictionary header fields are not included in this diagram.

Note that all references to a particular instance of a given class share exactly the same dictionary, as in

the case of the recursion between dictionaries shown in Figure 8. This significantly reduces the amount of space required to store the set of dictionaries that are used in a particular program and avoids the problems of repeated construction discussed in [29, Chapter 6].

The sharing of dictionaries is achieved by implementing `makeDict()` as a kind of memo-function. For each class, there is a corresponding index mapping monotypes or constructors to dictionaries. By allocating the storage for a dictionary, and inserting a path to it in the index before attempting to initialize its components, we can avoid ever trying to build the same dictionary twice.

The index from constructors to dictionaries is implemented by breaking each type expression down into a string of single `Tycon` values (or special values like `ARROW`, `LIST`, or tuples), and using these strings to locate the required element in a tree of the form shown in Figure 9. For example, the constructor string cor-

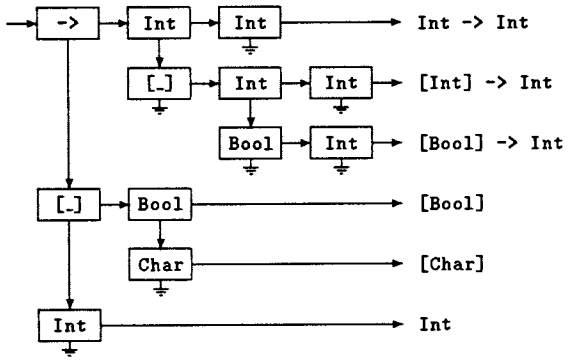


Figure 9: Indexing types by strings of constructors

responding to a type of the form `T t1 ... tn` starts with `T` and is followed by the strings for each of `t1` through `tn` in turn[8]. Each node in the index is labeled with a constructor and the dictionaries corresponding to individual types are located by traversing the index in the obvious way.

7.5.2 Translation of binding groups

The main purpose of this section is to describe how a group of mutually recursive bindings can be type checked, and to show how new dictionary parameters are introduced, as necessary, to implement overloading. This process is implemented by the `typeBindings()` function in `type.c`.

[8] In the current implementation, it is actually more convenient to take the arguments `t1` through `tn` in the reverse order, but the overall effect is the same.

The first task is to analyze the group of bindings to determine the correct set of typing rules. There are three cases:

- If the binding group contains any pattern bindings, or if it contains variable bindings (i.e. bindings in which the left hand side is a single variable) without any explicit type declarations, then the monomorphism restriction applies as described in [16, Section 4.5.4] and [24, Section 14.4.6]. The function `noOverloading()` will be used to type check the binding group in this case.
- If there are no pattern bindings, no variable bindings and no explicit type signatures, then the type checker infers the most general types possible for the bindings, possibly adding extra parameters for dictionaries, using the function `implicitTyping()`.
- In the remaining case, there are no pattern bindings but there are explicit type signatures and there may be some variable bindings. The function `explicitTyping()` will be used to infer types for the functions in the binding group, add dictionary parameters if necessary, and ensure that the inferred types match the declared types.

For the rest of this section, we will concentrate on the second case. The first is arguably a little easier, the third a little more complex because of the need to deal with user-supplied type constraints, but the basic principles are the same.

Consider a binding group of the form shown below with mutually recursive bindings for the variables $\mathtt{f}_1$, ..., $\mathtt{f}_m$ to bodies $\mathtt{e}_1$, ..., $\mathtt{e}_m$, with no explicit type signatures for any of these variables.

$$\begin{aligned} \mathtt{f}_1\ \mathtt{args}_1 &= \mathtt{e}_1 \\ &\vdots \\ \mathtt{f}_m\ \mathtt{args}_m &= \mathtt{e}_m \end{aligned}$$

The steps in calculating the types, and possibly translated definitions, for these variables are as follows:

- New type variables β_i are allocated to be used as the type for corresponding $\mathtt{f}_i$ values, and saved in the top level of `defnBounds`.
- Each of the bindings in the group is type checked. For each equation $\mathtt{f}_i\ \mathtt{args}_i = \mathtt{e}_i$, we calculate the types of both the left and right hand sides and unify the results to ensure that they are the same. In the process, dictionary variables will be added for each reference to an overloaded variable in $\mathtt{e}_i$, with corresponding triples added to `preds`, as described at the end of Section 7.2.

When type checking is complete, it may turn out that we need to add extra dictionary parameters for some $\mathtt{f}_i$ and hence we would need to traverse the bodies a second time, replacing each $\mathtt{f}_i$ with an expression of the form $\mathtt{f}_i\ \mathtt{dv}_1\ \ldots\ \mathtt{dv}_k$ for some dictionary parameters $\mathtt{dv}_1, \ldots, \mathtt{dv}_k$. In the worst case, with nested binding groups, repeated traversals like this could be very expensive. The type checker arranges for all references to a particular let-bound variable (i.e. a variable bound in `defnBounds`) to share exactly the same pair cell in the heap, that can be overwritten at some later point if extra parameters really are necessary. This is one of the reasons for distinguishing between let- and λ-bound variables in the representation of assumptions.

- Once all the bindings have been processed, the type checker marks all of the fixed variables in the current substitution (in the same way that is described in Section 7.3) and the `elimConstPreds()` function is used to deal with any class constraints that have been accumulated in `preds`. There are two ways to eliminate a triple `(pi,beta,v)` from `preds` at this stage:

  - If `pi` does not contain any free type variables, then we can use the function `makeDict()` described in Section 7.5.1 to construct the corresponding dictionary, and overwrite `v` with a pointer to it.
  - If the only free type variables in `pi` are all marked as fixed, then there is no point including it in the type of the $\mathtt{f}_i$ because we cannot generalize over any of the variables that it contains. Instead, the predicate is treated as a constraint on the environment that contains the binding group.

  These two cases correspond directly to the concepts of constant and locally constant overloading described in [30, 29].

- The remaining predicates in `preds` are simplified as much as possible to reduce the number of dictionary parameters that will be required. The simplification process eliminates duplicate predicates from `preds`. Predicates can also be eliminated if they can be derived either as superclasses or instance specifics from other members of `preds`. As described in [29, Section 8.2.3], we now consider the use of instance specifics in this way to be an error in the original design, and may change this behaviour in future releases of the system.

  The basic simplification algorithm, implemented by the function `simplify()` in `preds.c` can be described by the following pseudo-code:

```
simp ps = iterate s ps !! length ps
 where s []             = []
       s (p:ps)
         | entails ps p = ps
         | otherwise    = ps ++ [p]
```

  The main idea is to eliminate a predicate `p` if it is implied (i.e. entailed) by the remaining predicates `ps`. In addition, the simplification algorithm overwrites the dictionary variable `v` for any predicate that is removed with a suitable dictionary expression of the form `(#n d)`. In the case of duplicate predicates with corresponding dictionary parameters `v` and `v'`, the algorithm overwrites `v` with `(#0 v')`, rather than a copy of the `v'` cell. By sharing a single copy of `v'`, we avoid any problems if it also overwritten later in the simplification process.

- Finally, the dictionary variables in the remaining `preds` list are added as extra parameters to each of the variables in the binding group to obtain the translated versions of the original definitions. Similarly, for each i, we combine the predicate parts of the constraints in `preds` with the type bound to β_i in the current substitution and generalize (Section 7.3) to calculate the principal type for $\mathtt{f}_i$.

You are not alone if you consider this description of type checking a group of implicitly typed bindings to be rather daunting! A good part of the complexity here is caused by the need to deal with overloading, but even without that, there is still a lot of detail that does not show up in the simple rules used in many formal presentations of type inference. On the other hand, this level of complexity is not uncommon when dealing with complete languages, as in the static semantics for Haskell presented in [35] and the definition of ML in [42].

7.6 Abstract syntax for type checked programs

Figure 10 gives the grammar for the bindings that are produced as the results of type checking. The most important points here are:

- Type annotations are no longer included in *Bindings*.

```
Binding ::= (Var, [Alt])
          | ([Var], (Pat,Rhs))

Expr    ::= LETREC ([[Binding]],Expr)
          | COND (Expr,Expr,Expr)
          | AP (Expr,Expr)
          | Const
          | NAME Name
          | VAR Text
          | SELECT Int
          | DICTCELL Dict
          | FINLIST [Expr]
          | LISTCOMP Comp
          | MONADCOMP ((Expr, Expr), Comp)
          | RUNST Expr
          | CASE (Expr,[(Pat,Rhs)])
          | LAMBDA Alt
```

Figure 10: Grammar of type checked bindings

- Type annotations in expressions, for example, the term `(42 :: Int)`, represented using `ESIGN` values in previous stages, are not included in typed terms.
- Dictionary selector functions and dictionary constants are introduced using `SELECT` and `DICTCELL` values, respectively. Note that `SELECT n` is just the representation for the dictionary selector function `#n` in Section 7.5.
- Comprehensions, previously represented by `COMP` values, are split into two cases: `LISTCOMP` for list comprehensions and `MONADCOMP` for monad comprehensions. Note that the type checker will only produce `MONADCOMP` values if a standard prelude containing suitable definitions for the monad constructor classes is used. The representation of monad comprehensions includes two expressions for the appropriate `Monad` and `Monad0` dictionaries respectively. In the case where the comprehension can be interpreted over an arbitrary monad, the second of these dictionary expressions is replaced with `NIL`.

7.7 The kind system

Just as types can be used to classify values, kinds can be used to classify type constructors. The kind system in Gofer is used primarily to support the implementation of constructor classes described in [33] and has two main roles: to calculate suitable kinds for each of the data type constructors and classes in a program, and to use these kinds to ensure that only well-formed type and constructor expressions are used in Gofer programs. Without the complexities of polymorphism or overloading, the kind system is implemented as a greatly simplified version of the type system. Readers struggling to understand the basic features of the type checker may find that a study of the kind inference code in `kind.c` and of [33], provides a somewhat smoother introduction.

As we have already mentioned, kind checking can really be viewed as part of the static analysis of a program, rather than type checking. Nevertheless, we have included it in this section because of its similarity to the main type. More importantly, `kind.c` is included as part of `type.c` because it makes use of the same data structures to represent the 'current substitution' described in Section 7.1; the only difference is that the `bound` and `offs` fields in the current substitution are used to represent kinds, while the `kind` field is not actually used.

There are only two forms of kind expression, the kind $*$ denoting the collection of all types and represented by `STAR`, and function kinds of the form $\kappa_1 \rightarrow \kappa_2$ which are represented by pairs `pair(k1,k2)`, where `k1` and `k2` are the representations for κ_1 and κ_2, respectively. For the purposes of kind inference, we use `Offset` values to form skeleton kinds. Paired with an offset into the current substitution, these represent kinds that are initially unknown.

Further details about the kind system are given in [33] and the definitions in `kind.c` are a straightforward implementation of the ideas described there.

7.8 Experiments with `type.c`

To understand how a program works, it is often useful to be able to examine and study the results that it produces for different inputs. This is particularly important in the last three stages of the Gofer system, i.e. the type checker, compiler, and abstract machine support, where most of the effort in converting source programs to abstract machine code is concentrated. For each of these components, we will describe some simple ways to use or modify Gofer to gain some insight into the inner workings of the system. Most of these features were originally included in the Gofer source code as an aid to debugging.

The main source file for the type checker, `type.c` includes macro definitions for the symbols `DEBUG_TYPES` and `DEBUG_KINDS`. Normally, these are commented out and have no effect. Removing the comments from the first definition so that it reads:

```
#define DEBUG_TYPES
```

and rebuilding the system, we obtain a modified version of the interpreter that prints a lot of extra information during type checking. In particular, this includes details about:

- the allocation of new type variables,
- the results of unification,
- the introduction of new assumptions about bound variables, and
- the results of type checking individual expressions.

For large programs, this produces a considerable amount of output that should normally be redirected to a file so that it can be browsed and dissected using a text editor, or similar tools.

In a similar way, removing the comments from the second `#define` to obtain:

```
#define DEBUG_KINDS
```

yields a modified version of the interpreter that displays the inferred kinds of each datatype, type, and class in a given source program.

Finally, it is possible to get some insight into the construction and use of dictionary values by adding the line:

```
#define DEBUG_CODE
```

at the beginning of `type.c` and recompiling (unlike the previous examples, the standard distribution does not include a commented out version of this line).

The three flags described above can be used in any combination to select whatever information is of most interest. The format of the output is too complicated to explain here; you should expect to make frequent references to the Gofer source code during your first few experiments.

8 Compilation to supercombinators

Having dispensed with parsing, static analysis and type checking, `compiler.c` is used to translate Gofer programs into supercombinator definitions. The translation is implemented in four steps, described in the following subsections.

8.1 Program transformation

The first step in the compilation to supercombinators is to translate Gofer programs into a simpler language that provides only the core elements without some of the fancier features such as comprehensions, pattern bindings etc. One advantage of this is that it avoids further complexity in subsequent parts of the compiler. Switching to a simpler representation in earlier stages might also have eased the task of writing the static analysis and type checking components of the Gofer system. However, this would also have made it more difficult to give useful error messages, helping Gofer users locate the source of errors in their programs. Fortunately, errors of this kind cannot occur once a program has passed successfully through the type checker, leaving the compiler free to adopt different representations.

The reduction to the core language is carried out by a tree walk on the structure of type checked programs. The grammar of the output language is described in Figure 11. The main steps in the transformation are

Binding	::=	(*Var*, [*Alt*])
Rhs	::=	`GUARDED` [(*Expr*, *Expr*)]
	|	`LETREC` ([*Binding*], *Rhs*)
	|	*Expr*
Expr	::=	`LETREC` ([*Binding*], *Expr*)
	|	`COND` (*Expr*, *Expr*, *Expr*)
	|	`AP` (*Expr*, *Expr*)
	|	*Const*
	|	`NAME Name`
	|	`VAR Text`
	|	`SELECT Int`
	|	`DICTCELL Dict`

Figure 11: Grammar of translated bindings

as follows.

- Line numbers are removed from the representation of right hand sides (represented by the nonterminal *Rhs* in the grammars in this report). There is no need to retain this information in the compiler because we do not expect any further program errors to be detected[9].

[9] In retrospect, this may have been a little hasty; we have considered the possibility of modifying the pattern matching compiler, described in Section 8.2, to produce warning messages for non-exhaustive definitions. However, without the line numbers, it is difficult to relate such errors back to the input program.

- Local definitions, cast into lists of lists of bindings during dependency analysis, are flattened into simple LETREC values as suggested by the expansions given in Section 6.2.

- List and monad comprehensions are eliminated using the translations suggested by Wadler in [55] and [56], respectively.

- Case and lambda expressions are eliminated by translating them to equivalent expressions using local definitions. For example, the expression:

```
\f xs -> case xs of
           []     -> []
           (y:ys) -> f y : map f ys
```

would be translated as:

```
let g f xs
      = let h []     = []
            h (y:ys) = f y : map f ys
        in  h xs
in  g
```

Note that this requires the introduction of new variable names, in this case, g and h. In fact, the names that Gofer uses, generated by the inventText() function described in Section 4.1, are all printed as the letter v followed by a number; you will certainly have seen such variables as the result of this translation if you have ever entered a lambda expression without any arguments into the Gofer interpreter.

- Pattern bindings are reduced to simple variable bindings. For example,

```
let (x:xs) = e1 in  ...
```

is translated to:

```
let u            = conf e1
    conf u@(_:_) = u
    x            = head u
    xs           = tail u
in  ...
```

A couple of items here deserve further attention. First, the function conf is used to implement a conformality test, i.e. to ensure that the value of e1 matches the pattern (x:xs). Conformality tests are only used when the left hand side of a pattern binding contains an irrefutable pattern. In addition, Gofer provides a command line flag that can be used to suppress the introduction of conformality tests, with a small change in the semantics of pattern bindings. By default, conformality tests are used to ensure compatibility with Haskell. Notice also that new variables names, in this case u and conf, are generated as part of the transformation.

A second point is that, although we have used the functions head and tail in the code above, the compiler actually generates calls to an internal primitive function nameSel, sometimes printed _SEL. The definitions for x and xs become:

```
x  = _SEL (:) u 1
xs = _SEL (:) u 2
```

In general, _SEL c v n is used to extract the nth component in the value v, constructed using the constructor function c. There is no valid type for the _SEL function in the Gofer system. However, this does not cause any problems, since it is not introduced until after type checking is complete. If it were important, type safety could be restored by introducing special families of selectors for each datatype; indeed, this is almost what we have if we think of _SEL(:)1 as a name for the function \u -> _SEL (:) u 1.

8.2 Compilation of pattern matching

After the transformations described in the previous section, the only significant complication in the representation of Gofer programs is the use of patterns in function bindings. This section describes a program transformation, based closely on Wadler's description in [54], that reduces pattern matching in function definitions to a particularly simple form. For example, the standard map function, usually defined as:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

is translated to:

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

Given our description in the previous section about eliminating case expressions by translating them to function definitions, this may seem to be a step back in the wrong direction! However, it is important to realize that the pattern matching compiler will be used to translate arbitrary functions definitions. In

contrast, function bindings introduced as a result of **case** bindings in the original program are a special case because they always have exactly one argument.

Another important fact about the **case** expressions produced by the pattern matching compiler is that they do not use nested patterns. For example, the nested pattern binding in the source for the function definition:

```
firstDup (x:y:ys)
  = if x==y then x else firstDup (y:ys)
```

is translated to a nested pair of case expressions:

```
firstDup o1
  = case o1 of
      (o3:o2) -> case o2 of
                   (o5:o4) -> ...
```

The representation for the output of the pattern matching compiler is a little unusual because it does not include the binding occurrences of variables. For the examples above, we have used variable names **o1**, **o2**, ...for bound variable names. In fact, the actual representation used for the two examples above is somewhat closer to the following:

```
map {- arity 2 -}
 = case o1 of
     []  -> []
     (:) -> o2 o4 : map o2 o3

firstDup {- arity 1 -}
  = case o1 of
      (:) -> case o2 of
               (:) -> ...
```

The main idea here is to annotate function bindings with arities and to omit bound variables from pattern matching, leaving just the original constructor functions or constants. When required, the omitted argument variables can be inferred from context. The same idea is used for locally bound variables (but not for locally bound functions which will be eliminated by the transformations described in the next section). As another example, we give the standard Haskell definition of the function **filter**, the results produced by the pattern matching compiler, and the actual representation with implicit naming of bound variables. This example includes examples of the three different ways that local bindings can occur, as function arguments, in patterns, or in local definitions:

```
filter p []  = []
filter p (x:xs)
  = let rest = filter p xs
    in  if p x then x:rest
               else rest

filter o2 o1
 = case o1 of
     []      -> []
     (o4:o3) -> let o5 = filter o2 o3
                in  if o2 o4 then o4:o5
                             else o5

filter {- arity 2 -}
 = case o1 of
     []  -> []
     (:) -> let filter o2 o3
            in  if o2 o4 then o4:o5
                         else o5
```

One of the problems of this representation is that the meaning of an expression depends on the context in which it appears. For example, by itself, the local definition in the final version of **filter** above does not tell us the name of the variable that is bound to **filter o2 o3**. The fact that it appears in the scope of two function bound variables and two pattern bound variables is necessary to determine the correct variable name, in this case, **o5**. Fortunately, it is fairly easy to keep track of the number of bound variables as we traverse the representation of programs; this is what the 'current offset' parameter, **co**, is used for in many of the functions in **compiler.c**.

On the other hand, this choice of representation has some important benefits. First of all, omitting binding occurrences of variables reduces the space needed to store compiled programs. Although the savings are quite small, this was important for the early PC implementation of Gofer where space was limited. A more significant benefit is that the naming scheme for bound variables is carefully optimized for compatibility with the Gofer abstract machine described in Section 9. In particular, the variables **o1**, **o2**, ...are actually represented by **Offset** values corresponding to the positions of the corresponding values in the stack when the program is executed. This doesn't mean that the results of **compiler.c** could not be used for a different abstract machine. However, in the special case of the Gofer abstract machine, it avoids the need for any special environment mapping variables to stack locations[10].

[10] The Gofer compiler, **gofc**, added some time after the initial design of **compiler.c**, uses a compile-time optimization which changes the direct mapping from offsets to stack locations. A simulation of the run-time stack is used to determine the new location of bound variables. (See Section 11.)

The output of the pattern matching compiler is described by the grammar in Figure 12. As always,

```
LocalDef  ::=  ([Rhs], [FunDef])

FunDef    ::=  (Var, Int, (Fvs, FFs, Rhs))
Rhs       ::=  GUARDED [(Expr, Expr)]
           |   LETREC (LocalDef, Rhs)
           |   CASE (OFFSET Offset, Match)
           |   FATBAR (Rhs, Rhs)
           |   Expr

Match     ::=  [(Discr, Rhs)]

Expr      ::=  LETREC (LocalDef, Expr)
           |   OFFSET Offset
           |   ...
```

Figure 12: Bindings after pattern matching compiler

some additional comments are necessary to point out the most important features.

First, in preparation for lambda lifting, described in the next section, lists of local definitions are separated into a list of expressions, each of which is implicitly bound to an `Offset` value, and a list of function definitions. Each function definition, represented by *Fundef* in the grammar, includes the name of the function, its arity and a triple of the form (*Fvs*, *FFs*, *Rhs*) where:

- *Rhs* is the right hand side, or body, of the function.
- *Fvs* is a list of `Offset`s corresponding to the free variables appearing in *Rhs*.
- *Ffs* is a list of `Fundef`s that are referenced in *Rhs*. This links `Fundef` values together, reflecting the dependencies between them. Note that the *Fundef* values included in *Ffs* may either be defined in the same list as the current `Fundef` value or, otherwise, in some enclosing scope.

We will illustrate how this information is used in the following section. The task of collecting these details has little to do with the pattern matching compiler; if Gofer had been implemented in a Haskell-like language, this process would probably have been described using a separate pass over the representation of compiled terms. However, working in C, we chose to merge these two steps, avoiding a repeated traversal of the program graph.

Some final comments may be necessary to explain the `FATBAR` construct. A right hand side of the form `FATBAR` (*l*,*r*) is used to describe a combination of *l* and *r* which behaves like *l* except when that fails, either because of a failed pattern match or guard, in which case it behaves like *r*. This is exactly the same as the fatbar operator, [], used in [48]. A simple example is in the compilation of the function:

```
null []  =  True
null xs  =  False
```

Writing the definition in this way requires that the second equation is only used if the first fails. Applying the pattern matching compiler to this definition produces the following definition:

```
null {- arity 1 -}
  =  FATBAR (case o1 of
                [] -> True,
             False)
```

The reason for introducing `FATBAR` as a new construct, rather than a primitive function, is to make it easier for the code generator to recognize right hand sides of this form; in practice, many uses of the `FATBAR` construct can be eliminated during code generation. The only exception is when a `FATBAR` construct appears on the right hand size of a local variable definition, i.e. in a non-strict context. The code generator does use a primitive function to deal with this case.

8.3 Lambda lifting

Lambda-lifting is a program transformation that eliminates local function definitions. The result is a program containing only closed, global functions known as *supercombinators*, a term coined by Hughes in his presentation of an algorithm for lambda-lifting [18]. The Gofer compiler includes an implementation of an alternative algorithm proposed by Johnsson, who also introduced the term *lambda-lifting* [20]. Other descriptions of Johnsson's algorithm may be found in [48, Section 14.6] and in [49, Chapter 6].

Up to this point, all of the program analyses and transformations that we have described have been fairly general, and largely independent of any particular implementation technique; we would expect to find similar components in any implementation of a non-strict functional language. On the other hand, lambda-lifting is not essential for the compilation of such languages; for example, lambda-lifting is not required for either of the implementations by Turner

[52] or Peyton Jones [34]. The only reason for including a lambda-lifter in Gofer is to transform Gofer programs to a form that can be compiled for execution on the Gofer abstract machine, a variation on the Chalmers G-machine, that is described in Section 9.

8.3.1 A simple example

Lambda-lifting algorithms are based on a very simple idea; to turn a local function definition into a global function definition, add an extra parameter for each free variable in the body of the function. We will illustrate the effect of the lambda-lifting algorithm used in Gofer with the following definition of the `foldr` function. For convenience, we will write the definition using standard Gofer syntax. Of course, in practice, by this point in the compiler, the same definition will actually be represented using a `case` expression:

```
foldr a f = let g []     = a
                g (x:xs) = f x (g xs)
            in  g
```

This style of definition, using local definitions to describe higher-order functions, is popular with some Gofer programmers. On the other hand, the definition of `foldr` in the Haskell report [16], although equivalent, does not use a local definition. We will see that main effect of lambda-lifting in this example is to convert the definition above into something more closely resembling the definition in the Haskell report.

As it stands, we cannot treat `g` as a global function because it includes two free variables, `a` and `f`, in its definition. Adding these variables as extra parameters to `g`, we obtain the following definition:

```
foldr a f
  = let g a f []     = a
        g a f (x:xs) = f x (g a f xs)
    in  g a f
```

Since this new definition for `g` does not refer to any free variables, we can 'lift' the definition out as a new global function:

```
foldr a f    = g a f

g a f []     = a
g a f (x:xs) = f x (g a f xs)
```

We have now reduced the original program to a pair of supercombinator definitions without any local function bindings. In fact, in this example, there is an opportunity to improve the supercombinator definitions using η-reduction to define `foldr = g`. Even better, substituting `foldr` for `g` in the lifted function definition gives:

```
foldr a f []     = a
foldr a f (x:xs) = f x (foldr a f xs)
```

This final optimization using η-reduction is not included in the current Gofer implementation although it would be useful in a production quality compiler.

8.3.2 Lambda-lifting recursive definitions

In the general case, lambda-lifting is complicated by the need to deal with mutually recursive function definitions. As an example, consider the following definition:

```
f x y = let g z = x + h z
            h z = y - g z
        in  ...
```

Taking the same approach as the previous example, we might observe that, since `x` appears free in `g`, and `y` appears free in `h`, these definitions should be rewritten as:

```
f x y = let g x z = x + h y z
            h y z = y - g x z
        in  ...
```

But we still cannot turn the definitions for `g` and `h` into global functions because the transformation has introduced new free variables in the body of each function. For example, `y` now appears free in the body of `g`! However, we can repeat the original process, adding further parameters to the function definitions to obtain:

```
f x y = let g y x z = x + h x y z
            h x y z = y - g y x z
        in  ...
```

Finally, the definitions of `g` and `h` can be lifted out as global functions.

To explain this process a little more formally, we will write $lv(\mathtt{g})$ for the set of variables that have to be added to `g` before it can be lifted out as a global definition, and $fv(\mathtt{g})$ for the set of variables that appear free in the body of `g`. Our task now is to find solutions for the simultaneous equations:

$$\begin{aligned} lv(\mathtt{g}) &= fv(\mathtt{g}) \cup lv(\mathtt{h}) \\ lv(\mathtt{h}) &= fv(\mathtt{h}) \cup lv(\mathtt{g}) \end{aligned}$$

More accurately, since we would prefer to add as few parameters as possible, our real task is to find the least solution of these equations. Fortunately, there is a simple way to find least solutions to such equations by using a sequence of approximations and iterating until we reach a least fixed point. In other words, we generate a sequence of values of the form $(lv(\mathtt{g})_n, lv(\mathtt{h})_n)$ using the equations:

$$\begin{array}{lcl} lv(\mathtt{g})_0 & = & fv(\mathtt{g}) \\ lv(\mathtt{h})_0 & = & fv(\mathtt{h}) \\ lv(\mathtt{g})_{n+1} & = & lv(\mathtt{g})_n \cup lv(\mathtt{h})_n \\ lv(\mathtt{h})_{n+1} & = & lv(\mathtt{h})_n \cup lv(\mathtt{g})_n \end{array}$$

until we reach a point in the sequence where the sets produced do not change from one step to the next. Since we have only a finite number of equations, this process is guaranteed to terminate after a finite number of steps, obtaining a minimal solution to the original equations as its result.

The same technique applies to arbitrary collections of function bindings. For each function `f`, we need to find the smallest possible solution to an equation of the form:

$$lv(\mathtt{f}) = fvs \cup \bigcup_{\mathtt{g} \in ffs} lv(\mathtt{g}),$$

where *fvs* and *ffs* are the free variables and free functions, respectively, in the body of `f`. Note that these are exactly the values that we arranged for the pattern matching compiler to store in the representation of *FunDef* values, as described in Section 8.2. This information is used by the `solve()` function in `compiler.c` to solve the simultaneous equations and calculate the sets of variables to be added to each definition.

Apart from adding extra parameters to each function, we also need to add the appropriate variables as extra parameters in calls to lifted functions. For example, replacing calls to `h` with calls to `h x y` in the above. Perhaps the most direct implementation would be to define a function to carry out substitutions of the form `[h x y/h]P`, replacing free occurrences of `h` in `P` with `h x y`. However, this would be very expensive since it would almost certainly require multiple traversals of program fragments, particularly when dealing with nested local definitions. Our implementation avoids this problem by using a standard technique in the implementation of programming language interpreters, using an environment, `tr`, to record a translation for each variable. Since all of the information needed to compute $lv(\mathtt{f})$ sets is calculated before lambda-lifting, the translation environment `tr` can be extended with the appropriate translations for locally defined functions before traversing the bodies and scope of the local definition. This avoids any need for multiple traversals.

8.3.3 Representation of supercombinators

The result of the lambda-lifter is a collection of global functions, or supercombinators, each of which is described by its name, its arity, and a right hand side of the form specified by the grammar in Figure 13. The same representation is used for the supercombinator definitions passed to the code generator in `machine.c`.

Rhs ::= `GUARDED` [(*Expr*, *Expr*)]
| `LETREC` ([*Rhs*], *Rhs*)
| `CASE` (`OFFSET Offset`, *Match*)
| `FATBAR` (*Rhs*, *Rhs*)
| *Expr*

Match ::= [(*Discr*, *Rhs*)]

Expr ::= `LETREC` ([*Rhs*], *Expr*)
| `COND` (*Expr*,*Expr*,*Expr*)
| `AP` (*Expr*,*Expr*)
| *Const*
| `OFFSET Offset`
| `NAME Name`
| `SELECT Int`
| `DICTCELL Dict`

Figure 13: Grammar for supercombinators

8.4 Pre-compiler

The final, rather poorly named, stage of `compiler.c` massages the results of the lambda-lifter into a form that can be used as input to the code generator. The structure of compiled programs is not modified by this process. The only changes that it makes are adjustments to the numeric values of `Offset` values corresponding to bound variables. This is only necessary because of our decision to make binding occurrences of such variables implicit in the representation of programs, as described in Section 8.2. To see why some adjustments are necessary, consider the following example:

```
f o2 o1 = let g o3 = o2 + o3
          in  g o1
```

Rewriting this with implicit naming of bound variables gives:

```
f {- arity 2 -}
   = let g {- arity 1 -} = o2 + o3
     in  g o1
```

Lambda-lifting adds an extra parameter to g and produces the supercombinator definitions:

```
f {- arity 2 -} = g o1
g {- arity 2 -} = o2 + o3
```

However, expanding out the implicit bindings, the definition of g becomes:

```
g o2 o1 = o2 + o3
```

Clearly this is wrong; the variable o3 on the right hand side does not even appear on the left!

Fortunately, it is fairly easy to calculate the correct offset values when we lift out the body of a locally defined function if we know[11]:

- the arity, a, of the function prior to lifting,
- the offset, r, of the right hand side, and
- the list, **extraVars**, of free variables to be added as extra parameters.

The values of these parameters for the function g in the example above are 1, 4, and [o2], respectively. Writing n for the length of **extraVars**, the lifted function will have arity a+n. The adjusted value of an offset o in the body of a lifted function can be calculated as follows:

- If $1 \leq o < r-a$, then o corresponds to a free variable, included in the list **extraVars**. The adjusted offset is a value in the range $a<o' \leq a+n$, determined by the position of o in the list.
- If $r-a \leq o < r$, then o corresponds to a parameter of the function before lifting, and maps to an offset in the range $1 \leq o' \leq a$.
- If $r \leq o$, then o corresponds to an offset introduced in the body of the function, and maps to the offset $(o-r)+(a+n)>a+n$.

For the example above, these calculations map o2 to o2 and o3 to o1, so that the correct definition of g generated by the pre-compiler is:

```
g {- arity 2 -} = o2 + o1
```

[11] The parameters a, r, and n used in the description here correspond to **localArity**, **localOffset** and **numExtraVars**, respectively in the current implementation.

Despite appearances, the calculation for adjusting offsets is straightforward. The only reason for including the details here is to illustrate the consequences of our non-standard representation for programs with implicit introduction of bound variables.

8.5 Experiments with compiler.c

One of the best ways to explore and understand the workings of **compiler.c** is to study the output that it produces for particular Gofer programs. The **gofc** program provides an easy way to inspect the output of the compiler in a fairly readable form. On most systems, a command of the form:

```
gofc +D prog.gs
```

will run **gofc**, dumping pretty-printed versions of the supercombinators for the definitions in **prog.gs**, including those from the prelude, in the file **prog.gsc**. In theory, this output file could also be used as source code for an alternative back-end for the Gofer system. This might be useful, for example, in experiments with new code generators or program analyzers, using the Gofer front-end instead of writing a new parser, type checker, etc. The only problem with this is that the current pretty-printer does not include the structure of dictionary values in the output file. This shortcoming could be fixed by modifying the printer, or by avoiding programs and prelude files that involve type classes.

9 Program execution

The code in **machine.c** is used to compile supercombinator definitions produced by **compiler.c** to instructions for an abstract machine, and to simulate the execution of this machine to provide a lazy evaluator for Gofer programs. While there are some differences, this part of the Gofer system borrows heavily on the ideas used in the Chalmers G-machine, described in [4, 48]. On the other hand, by isolating the details of the abstract machine in a single file, it should, in principle, be possible to replace this part of the Gofer system with alternative back-ends, for example, based on the Three Instruction Machine [60]. In practice, this may also require changes elsewhere in the system, for example, in the storage management routines of **storage.c** to support heap allocation of closures, or in **compiler.c** to add extra program transformation steps.

We will not include formal rules for translating Gofer programs to the instruction set of the Gofer abstract

```
Expr  ::=  AP (Expr,Expr)
       |   Const
       |   NAME Name
       |   DICTCELL Dict
       |   FILECELL Int
```

Figure 14: Representation of program graphs

machine in this report; this would serve only to duplicate details that are already clearly documented in the source code for `machine.c`. Instead, we will give a detailed description of the way that the Gofer abstract machine works, including a description of its instruction set, and give some examples to illustrate how they are used to implement supercombinator reduction.

9.1 The evaluator

Gofer programs are executed by evaluating expressions. Often, the result is a list of characters or a list of I/O requests; this determines what the user sees as the results of running a program. Starting with the initial expression, represented by a graph structure in the heap, Gofer uses an evaluator to transform the graph by a sequence of reductions, each corresponding to an equation in the source program or to some built-in system primitive. As a simple example, the evaluation of the expression `show (3*4+5)` requires three reduction steps[12]:

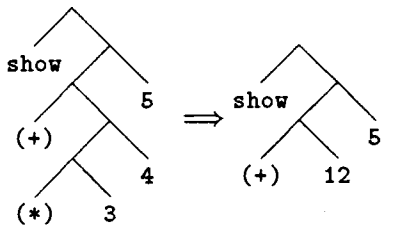
```
show (3*4+5)  ⟹  show (12+5)
              ⟹  show 17
              ⟹  "17"
```

Internally, these expressions are represented by a very simple form of program graph stored in the Gofer heap and described by the grammar in Figure 14. Note that program graphs are closed expressions (i.e. there are no free variables) built up from atomic values using function application. Graph structures are necessary because, in the general case, programs may include shared subexpressions or cycles, introduced by recursive definitions. The `FILECELL` values introduced here are used to implement input from a file as a lazy stream. Notice also that, while the grammar for program graphs includes dictionary values, there is no need to include selector functions; applications of selectors are implemented more efficiently using the `DICT` instruction described in Section 9.2.

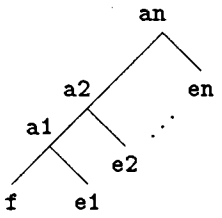
[12]For simplicity, we do not consider overloading of `show` or the arithmetic operations in this example.

Strictly speaking, we should also have included an extra production for indirection nodes in the grammar for *Exprs* in Figure 14; indirections are simply pointers to other expressions in the heap and are used in the implementation of lazy evaluation, particularly for the `UPDATE` instruction described in Section 9.2.

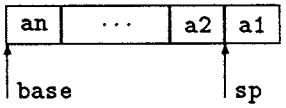
Returning to the example above, each step in the reduction sequence is actually implemented as a graph rewrite. For example, the reduction in the sequence corresponds to:



The evaluator is implemented as a function `eval()` that reduces the `Cell` value passed as its argument to *weak head normal form*. Suppose that the original expression is represented by a program graph of the form:



Starting at the application labeled `an`, the evaluator works its way down the spine of the graph, recording the values of each application node on the stack until it reaches the head, `f`. Using `base` to record the original position of the stack pointer when the evaluator begins, the layout of the stack by the time the head is reached can be illustrated by the following diagram:

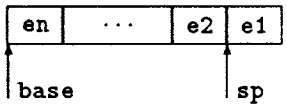


The most important cases we need to consider are as follows:

- If `f` is a constant value, for example, an integer or a floating point number, then the expression is already in weak head normal form. In fact, the type system guarantees that there will not be any arguments in this case. To return its result, the evaluator stores the value of the head, `f`, in the global variable `whnfHead`, serving as a register of the abstract machine. If the value is an

integer, then the evaluator also records the corresponding integer value in the variable **whnfInt**. In a similar way, floating point numbers are returned by placing the corresponding value in the **whnfFloat** variable.

- If **f** is a constructor function, then the expression is also in weak head normal form. Once again, the evaluator returns the head of the expression in the variable **whnfHead**. In addition, the elements on the stack are rearranged so that the calling program can access the values of the arguments to the constructor function; for the example above, the layout of the stack when the evaluator returns will be:

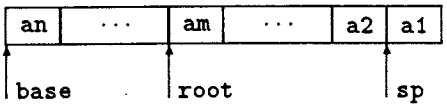


To illustrate how this can be used, the following fragment of C code shows how the Gofer evaluator could be used to calculate the sum of a list of integers, specified by an expression **e**:

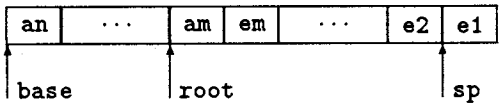
```
total = 0;
eval(e);
while (whnfHead==nameCons) {
  eval(pop());              /* head */
  total = total + whnfInt;
  eval(pop());              /* tail */
}
```

Notice that it is necessary to evaluate, not just every node in the list, but also every integer value that it contains. On the other hand, the Gofer type system allows us to omit run-time type checks from this code. For example, there is no need to check that the value produced by the first `eval(pop());` statement is an integer because the type system guarantees that it will be.

- If **f** is a function expecting **n** or fewer arguments, then the evaluator calls the code for **f** to carry out the appropriate modifications to the graph. This code may be implemented by a built-in primitive function, coded in C, or by a sequence of abstract machine instructions, as described below. Before calling the function, the evaluator rearranges the values on the stack. Suppose that **f** has arity **m**. First, we set a point, **root**, to identify the top **m** elements on the stack, to be used as arguments to **f**:



Then, to make it easier for the code for **f** to access the argument values rather than the application nodes from which they came, we modify the stack to give:



Note that the value **am** is retained in the **root** position on the stack; this will be used by the code for **f** to overwrite the original call to **f** with the resulting expression.

The function **f** may require further evaluation of its argument values; this can be achieved by further calls to `eval()` in the body of the code for **f**.

Finally, when the reduction is complete, the evaluator resets the stack pointer to the **root** position and continues, unwinding the result of the reduction onto the stack and looping until a weak head normal form is obtained.

9.2 Abstract machine instructions

A small number of Gofer functions are implemented as primitives, hand-coded in C. However, most of the functions in a typical program are represented by sequences of instructions to be executed by the Gofer abstract machine. These instructions have two purposes, first to examine the arguments of the function to determine which equation in the definition of a function should be applied, then to make the appropriate rewrite.

We will list the instruction set of the Gofer abstract machine in two groups. The first group, described in Section 9.2.1, are used primarily to construct the result of a reduction. The second, in Section 9.2.2, are used to deal with control flow.

9.2.1 Instructions for constructing values

The following instructions are used to construct fragments of program graph, using the stack to store temporary values as well as the function parameters and the results of calls to the evaluator.

- `LOAD` *n*: Push the value from position *n* in the current stack frame onto the top of the stack.

Used to access the values of function arguments or locally bound variables.

- `CELL` *c*: Push the constant cell value specified by *c* onto the top of the stack. Used to access constants such as constructor functions and supercombinators.

- `CHAR` *n*: Push a character value onto the top of the stack, corresponding to the integer value *n*.

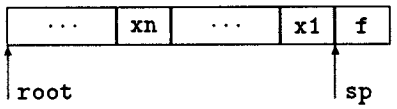
- `INT` *n*: Push the integer *n* onto the top of the stack.

- `FLOAT` *f*: Push the floating point number *f* onto the top of the stack.

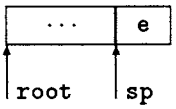
- `STRING` *str*: Push the string value *str* onto the top of the stack. Note that the string is represented using a `Text` value as described in Section 4.1.

- `MKAP` *n*: Apply the function on the top of the stack to the *n* argument values immediately below it. The top *n*+1 elements on the stack are replaced by the resulting expression. Note that this instruction does not involve any evaluation of the function, its arguments, or the resulting expression.

  For example, a `MKAP n` instruction takes a stack of the form:

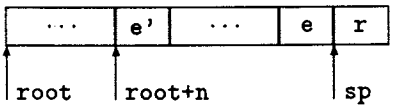


  and reduces it to the stack:

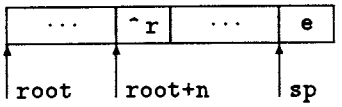


  where `e = f x1 ... xn`.

- `UPDATE` *n*: Removes the value, `r` say, from the top of the stack and updates the *n*th element in the current stack frame with a pointer to an indirection node to `r`. For example, the instruction `UPDATE n` reduces a stack of the form:



  to the stack:

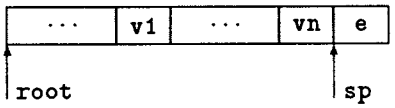


  This instruction is used to overwrite the root of an expression as part of the implementation of lazy evaluation, i.e. to ensure proper sharing of the result of an expression. It is also used to save the values of variables bound in a local definition. The indirection cell is used to avoid a loss of laziness [48, Section 12.4]. The use of an indirection is reflected by the caret symbol in the diagram above.

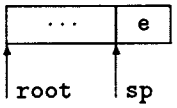
- `UPDAP` *n*: This instruction has almost the same effect as a `MKAP 1` instruction followed by an `UPDATE n` instruction; in other words, the *n*th element in the current stack frame is replaced with the application formed from the top two values on the stack (which are subsequently discarded). The difference is that the `UPDAP` instruction should only be used when it is known that the *n*th element on the stack already points to an application node; in this case, the old application node can be overwritten with the new values, avoiding the need to allocate a new application node.

- `ALLOC` *n*: Allocates *n* pairs, each initialized so that both first and second components are `NIL`, and pushes a pointer to each pair allocated onto the stack. This instruction is as part of the process of initializing (possibly recursive) local variable bindings.

- `SLIDE` *n*: Slides the value on the top of the stack down *n* places by removing the *n* values immediately below it. This instruction is used to deal with expressions involving local variable definitions. For example, suppose that the expression `e` is constructed using values `v1`, ..., `vn` bound to local variables. This produces a stack of the form:



  Executing a `SLIDE n` instruction produces a stack of the form:



  Of course, the expression `e` may itself contain pointers to the values in `v1`, ..., `vn`.

- `DICT` *n*: Replaces the value on the top of the stack—which must be a dictionary value, `d` say—with the value held in the `n`th slot of `d`. This instruction is used to implement dictionary lookup by compiling expressions of the form `(#n d)` to an instruction sequence:

```
... code to build d ...
DICT n
```

  The type system ensures that the `DICT` instruction will only ever be used when the value on the top of the stack is a dictionary. Furthermore, since all of the dictionary values required by a program are constructed before it is executed, there is no need to evaluate the dictionary `d` before the `DICT` instruction. Thus `DICT` can be implemented very efficiently with just a couple of machine instructions without needing a run-time representation for dictionary selectors.

- `ROOT` *n*: Used in the implementation of the root optimization; see Section 9.3 for further details.

9.2.2 Control flow

The following instructions are used to call the evaluator, to test the values that it returns, to indicate the end of a particular reduction, or to signal an irreducible expression.

- `EVAL`: Pops the top expression off the stack and calls the evaluator to calculate its value.

- `INTEQ` *n addr*: If the value in the `whnfInt` register is equal to the integer constant *n*, then continue with the current sequence of instructions. If the two values are not equal, then control transfers to *addr*. This instruction is used to support pattern matching of integer constants.

- `INTGE` *n addr*: If the value in the `whnfInt` register is greater than or equal to `n`, then the integer `whnfInt - n` is pushed onto the stack. If the test fails, then control passes to the address *addr*.

  This instruction is used to support `(p+k)` patterns. For example, an `INTGE 2 addr` instruction might be used to match the value on the top of the stack against the pattern `(v+2)`. If the value on the top of the stack is less than 2, then the match fails and execution transfers to the instruction at `addr`. Otherwise, the match is successful and the value pushed onto the stack gives the value bound to the variable `v`.

  Extending Gofer to allow `(p+k)` patterns to be matched against values of any type in the `Integral` class as permitted in Haskell would require some extensions to the `INTGE` instruction, or, more likely, more significant changes to the program transformations used in `compiler.c`. However, it seems more likely that future versions of Gofer will eliminate support for `(p+k)` patterns altogether.

- `INTDV` *n addr*: Similar to `INTGE`, this instruction is used to determine whether the value in the `whnfInt` register is a positive multiple of the integer `n`. If so, the value of `whnfInt/n` is pushed onto the stack. Otherwise, control transfers to the instruction at address `addr`.

  This instruction is used to support pattern matching of `(n*v)` patterns, added as an experimental feature to the Gofer interpreter. This form of pattern is now considered obsolete.

- `TEST` *c addr*: This instruction compares the value in `whnfHead` register with the value of the `Cell` constant *c*, branching to the instruction at address *addr* if the values are not the same.

  This instruction is used in the implementation of pattern matching. To illustrate this, consider a conditional expression:

```
if e then t else f
```

  and note that this is equivalent to the `case` expression:

```
case e of True  -> t
          False -> f
```

  The following code can be used to calculate the value of this conditional in situations where it is clear that results of the conditional will always be required (i.e. when the conditional appears in a strict context):

```
        ... code to build e ...
        EVAL
        TEST True label
        ... code to build t ...
        RETURN
label:  ... code to build f ...
        RETURN
```

  (Each of the sections of code that 'build' an expression here are expected to leave the required expression, unevaluated, on the top of the stack.)

As an aside, in situations where we cannot be sure that the value of the conditional will be required, we use code of the form:

```
... code to build f ...
... code to build t ...
... code to build e ...
CELL nameIf
MKAP 3
```

to build a delayed version of the conditional expression. The `nameIf` cell used here refers to a primitive function defined so that:

```
nameIf True  t f  =  t
nameIf False t f  =  f
```

Clearly, we would prefer to use the code containing the `TEST` instruction whenever possible because it avoids the need to build code for both `t` and `f` when the program executes.

- `GOTO` *addr*: Causes an unconditional jump to the instruction at address *addr*.
- `RETURN`: Signals the end of (one possible path through) the code for a supercombinator. The `RETURN` function usually follows the `UPDATE 0` or `UPDAP 0` instructions used to update the root of the current redex.
- `FAIL`: Used to signal a failed pattern match, usually resulting in a run-time error.

  For convenience, the current version of the code generator places a single `FAIL` instruction at a fixed address, recorded in the variable `noMatch` during the initialization process. Pattern matching failures in other parts of the program are triggered by branching to this address.
- `SETSTK` *n*: Resets the stack pointer to `root+`*n*. This instruction is used in to set the stack pointer to a known position in situations where its value cannot be determined at compile time.

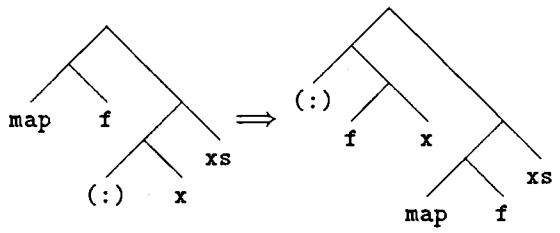
9.3 The root optimization

Quite a few of the functions in the Gofer standard prelude use the same initial parameters in each recursive call. For example, the same function `f` is used for each call to `map` in the following definition:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Apart from standard user-defined functions, several of the program transformations used in the Gofer compiler generate definitions of this form, particularly the addition of extra parameters in the implementation of overloading and the use of lambda-lifting. Since the same parameters are used for each call, it seems a shame that we have to duplicate them each time.

The Gofer code generator avoids this problem by detecting places where part of the original expression to be reduced can be reused to construct the result. For example, the second equation in the definition of `map` corresponds to the following reduction:



In this case, there is no need to reconstruct the application of `map` to `f` because we can recycle the application on the left hand side. As a result, this reduction requires the allocation of only 3 new application nodes, rather than the 4 nodes that would normally be required.

We refer to this as the 'root optimization' because all of the expressions that are recycled are derived from initial portions of the root of the original expression. The abstract machine instruction `ROOT n` is used to implement the root optimization; its purpose is to push the expression obtained by deleting the last `n` parameters from the root onto the top of the stack. For example, as we will see in the following section, a `ROOT 1` instruction is used in the compilation of `map`.

The root optimization was originally described in [23], including figures suggesting a 25–30% reduction in space requirements for some small examples using the optimization. We are not aware of any other systems that use a similar optimization.

9.4 Examples

This section gives some examples of the machine code produced by the system that is used to implement combinator reduction.

The simplest example is the identity function defined by `id x = x`. The corresponding sequence of machine instructions is:

```
0x0009      LOAD    1
```

```
0x000B      UPDATE  0
0x000D      RETURN
```

The values on the left are the addresses of the instructions on the right. The code is easy enough to follow; we push the first (and only) argument on to the stack and use it to overwrite the original call to the `id` function.

Only slightly more complicated, the implementation of function composition, defined by `(f . g) x = f (g x)`, is described by the instructions:

```
0x0051 LOAD 1
0x0053 LOAD 2
0x0055 MKAP 1
0x0057 LOAD 3
0x0059 UPDAP 0
0x005B RETURN
```

The first three instructions load the values of `x` and `g` onto the stack and combine them to form the subexpression `g x` of the result. The next two instructions overwrite the root of the original call to `(.)` with the application of `f` to `g x`, completing the reduction.

As a simple example of pattern matching, consider the `(&&)` function defined by:

```
False && y  =  False
True  && y  =  y
```

The corresponding instruction sequence is as follows:

```
0x006E LOAD 2
0x0070 EVAL
0x0071 TEST False 0x0079
0x0074 CELL False
0x0076 UPDATE 0
0x0078 RETURN
0x0079 TEST True 0x0000
0x007C LOAD 1
0x007E UPDATE 0
0x0080 RETURN
```

These instructions evaluate the first argument of `(&&)`. If the result is `False`, then the definition returns the value `False` in the line labeled `0x0074`. If the result is not `False` then we branch to the `TEST` instruction labeled `0x0079`. The `0x0000` label used here is just the value of `noMatch`, mentioned in the description of the `FAIL` instruction above. Strictly speaking, this instruction is redundant and the program will never actually branch to `0x0000` because the type system guarantees that the only possible results that could occur here are `False`, which has already been eliminated, and `True`. The remaining instructions simply overwrite the original expression with the value of the second parameter, much like the implementation of `id`.

As a final example, the following sequence of instructions that is generated from the definition of the `map` function given above:

```
0x00D6 LOAD 1
0x00D8 EVAL
0x00D9 TEST [] 0x00E1
0x00DC CELL []
0x00DE UPDATE 0
0x00E0 RETURN
0x00E1 TEST : 0x0000
0x00E4 LOAD 3
0x00E6 ROOT 1
0x00E8 MKAP 1
0x00EA LOAD 4
0x00EC LOAD 2
0x00EE MKAP 1
0x00F0 CELL :
0x00F2 MKAP 1
0x00F4 UPDAP 0
0x00F6 RETURN
```

We will not comment further on this example, leaving the task of understanding how these instructions implement the `map` function as a strongly recommended exercise for the reader.

9.5 Experiments with `machine.c`

As with the type checker and compiler, one of the best ways to understand the workings of `machine.c` is to study the programs that it produces as its output. This can be achieved by modifying the source code to include the line:

```
#define DEBUG_CODE
```

at the beginning of `machine.c`, and recompiling. This will produce a custom version that shows the code for each compiled function as it is produced by the code generator. In practice, this tends to produce rather a lot of output so it is probably best to redirect the output of the modified compiler to a file that can be examined more carefully using a text editor. All of the examples in the previous section were obtained using a version of the interpreter that had been modified in this way.

10 Functional programming in an imperative world

We have already described, in Section 3.1, the reasons why the Gofer system was not written in a functional language. However, looking at the source code, it is clear that functional programming has been a significant influence on its development. In some places, there are even small fragments of Gofer code, or type annotations, that were used to develop some sections of the program before coding them in C.

It is not too difficult to find other aspects of the implementation of Gofer that have been influenced by functional programming systems, including:

- The garbage collected heap, with constructor functions such as `pair` and selectors `hd`, `tl`, `fst` and `snd`. Simple list processing functions such as `length` and (destructive) `append` and reverse are implemented using these functions.
- The C preprocessor is used quite heavily, for example, to support weak forms of higher-order functions and polymorphism. For example, the implementation uses a range of `map`-like operations for processing lists of values, implemented by macros such as:

```
#define mapBasic(_init,_step)  \
  {List Zs=(_init);            \
   for(;nonNull(Zs);Zs=tl(Zs)) \
   _step;}
```

Some aspects of the current implementation do rely on side-effects, for example, the error trapping mechanisms and the implementation of type checking, garbage collection and graph reduction. Some of these can be implemented cleanly and efficiently in a purely functional language, but others remain as open (and perhaps uninteresting) problems.

Based on our experience with Gofer, we believe that the ideas and idioms of functional programming can play an important and useful role in the development of programs, even if they are actually written in traditional imperative languages.

11 The Gofer compiler, `gofc`

A short time after the release of Gofer, I was asked whether it would be easy to produce a simple-minded 'compiler' that would translate programs written in a functional language into executable C code (without too many concerns about performance). Given the machinery that had already been developed for the Gofer interpreter, this turned out to be a relatively simple task.

The Gofer compiler, `gofc`, was developed by modifying the backend of the interpreter to output suitable C code for each of the abstract machine instructions in a given program. Combined with a run-time system derived from a simplified version of the interpreter storage management system and the implementation of primitive functions, `gofc` allows the development of small, stand-alone Gofer applications. In performance terms, simple tests suggest that compiled programs typically offer only a twofold increase in run-time speed over interpreted code. Serious Haskell compilers can do much better than this using more sophisticated implementation, analysis and optimization techniques. In addition, `gofc` does not support any form of separate compilation, making it unsuitable for large projects requiring repeated compilation. In its favour, `gofc` often produces significantly smaller executable binary files than other systems. The translation of Gofer programs to C is usually quite fast, although compilation of the resulting C programs can take rather longer. To some extent, the interpreter reduces the problems caused by the lack of separate compilation, offering a more responsive interactive development environment and delaying the need for compilation until the final stages of a project.

One important advantage of the simplified run-time system used by `gofc` is that it does not require a conservative garbage collector. In particular, it is not necessary to scan the C stack to look for pointers into the heap, avoiding many of the problems and restrictions of the garbage collection system in the interpreter described in Section 4.2.3. In fact, the source code for `gofc` currently includes two different garbage collectors, a simple mark-scan collector in `markscan.c` and a two-space, copying garbage collector in `twospace.c`. The latter supports the allocation of variable length blocks of heap space and has recently been used to add an implementation of Haskell-style arrays with $O(1)$ access time. By contrast, the array implementation for the mark-scan collectors in both the interpreter and the compiler requires a less efficient representation of arrays using linked pair cells. It would certainly be possible to extend the run-time system to support different garbage collectors. For example, one obvious possibility would be to experiment with the implementation of a simple generational collector.

We will not go into great details about the translation of abstract machine instructions to the corresponding

C code. However, as a simple example, the following implementation of the **map** function is taken directly from the output of **gofc** and should be compared with the sequence of abstract machine instructions for **map** given in Section 9.4.

```
comb2(sc_map)  /* map */
  needStack(4);
  eval(offset(1));
  test(mkCfun(0)) goto a;
  update(0,mkCfun(0));
  ret();
a:test(mkCfun(1)) fail();
  heap(3);
  pushpair(rootFst(offset(0)),offset(3));
  pushpair(offset(2),offset(4));
  topfun(mkCfun(1));
  updap2(0);
  ret();
End
```

This code makes heavy use of C preprocessor macros, defined in the header file **gofc.h**. The lines of the form **needStack(n)** and **heap(n)** are used to test for the availability of blocks of stack and heap storage, avoiding the need to check for overflow before each **push()** or **pair()** call, respectively[13]. Notice also that constructor functions are represented using values of the form **mkCfun(n)**. In the example above, the nil list, **[]**, and the cons function, **(:)**, are represented by the values **mkCfun(0)** and **mkCfun(1)**, respectively. The same two **mkCfun** values are used to represent the boolean values **False** and **True**, but the type system ensures that these two uses are never confused. However, this choice of representation does mean that it is impossible to write a fully polymorphic primitive function like **show'** in the interpreter that is able to interpret the run-time structure of a value and produce a corresponding printable representation as a string. This is not a major shortcoming; indeed this is exactly the kind of application that type classes are intended to be used for!

One feature of the translation from machine instructions to C is the use of technique suggested by [48, Section 19.3.2] to simulate the run-time stack at compile-time. This allows the code generator to avoid unnecessary use of the stack, but requires a more complicated form of instructions in generated C programs. For example, the single line:

```
pushpair(rootFst(offset(0)),offset(3));
```

in the implementation of **map** above corresponds to the sequence of three abstract machine instructions:

```
LOAD 3; ROOT 1; MKAP 1
```

This sequence of instructions uses two stack locations, duplicating a value that is already available from another position on the stack. The two values pushed onto the stack are treated, implicitly, as arguments of the **MKAP** instruction. By contrast, the C code version avoids two unnecessary pushes, but requires the arguments of **pushpair()** to be specified explicitly. It is not clear whether this has any noticeable effect on execution time.

The **gofc** compiler shares a substantial amount of code with the interpreter. Only the main program (in **gofc.c**), the code generator (in **cmachine.c**), and the definitions of built-in functions (in **cbuiltin.c**) are different from the corresponding components in the interpreter. In fact, an early decision was to ensure that the compiler and the interpreter could be built from *exactly* the same compiled object code for common components. Sharing code to this degree has obvious benefits, for example, there is only one version of the type checker to maintain and compile. However, it has also been necessary to make some compromises in the code to allow proper sharing, and the results are not entirely satisfactory.

The implementation of the run-time system (in **runtime.c**) suffers from even more severe problems because it does not share any code with other parts of the system. As a result, every primitive function in the interpreter has had to be rewritten for the compiler. This is tedious at best, with coding errors leading to discrepancies in the behaviour of compiled and interpreted code. In addition, the run-time system compiles to a single object code file that is included as a whole in every compiled program. A better approach would be to arrange it as a library of modules, allowing the linker free to omit definitions that are not needed in a given program. In fact, **gofc** already goes to some lengths to avoid including unused code from user programs[14], so it is a pity that the same principles are not used for primitive functions. At first sight, we might hope that this problem could be overcome simply by spliting the source code for the run-time system into separate parts. Unfortunately, it would also be necessary to make more fundamental changes to the representation of primitives to obtain any benefits from this.

[13] The **heap(n)** instruction is only useful for the two-space collector which allows allocation of cells in contiguous blocks. If the mark-scan collector is used, it is just as efficient to continue testing for heap exhaustion before each **pair()** call, and to treat **heap()** calls as no-ops.

[14] For example, most programs use only a fraction of the functions defined in the standard prelude.

None of this is really surprising given that the `gofc` implementation was hacked into a system that had never been intended or designed to support a compiler. Perhaps there will be an opportunity to develop a new version of the system at some point in the future, allowing for both an interpreter and a compiler in the initial design, and avoiding some of the structural problems in the current version. As every programmer knows, the development of a system always provides valuable lessons for the next version!

12 Future directions

Although the amount of time available for work on Gofer is currently quite limited, there are always plenty of ideas for future developments and extensions[15] to the system. In this final section, we describe some of these topics as an indication of possible directions for further development of the Gofer system.

Closer compatibility with Haskell To some, the goal of closer compatibility may seem strange; Gofer is, after all, very closely based on Haskell! However, as mentioned earlier, there are some small, but annoying incompatibilities between the two languages; neither is a subset of the other. Fortunately, these problems are becoming less severe as new releases of Gofer move (very slowly) towards closer compatibility with Haskell.

Module system There was never any thought when Gofer was first released that it would be used for anything but small programs. Perhaps 100 lines, maybe as much as 500 lines, but certainly nothing more. As such, and with concerns about being able to run the interpreter on small machines, it seemed appropriate to avoid the complexities of any form of module system. This has actually been quite a sensible approach, given the intended use of Gofer as an experimental system; it is easier to develop and experiment with language extensions if we do not have to worry about their interaction with modules. However, it seems that Gofer is often used for quite large programs, and that some form of module system would be useful.

One of the first concessions in this area was to extend the parser to read, but otherwise ignore, the module headers used in Haskell programs. This means that some Haskell programs making only trivial use of modules can be fed directly into Gofer without any changes. Indeed, it seems that Gofer is often used in this way to develop sections of programs that will ultimately be compiled using a full Haskell system.

One option, particularly with the goals of compatibility in mind, would be to add a proper implementation of the Haskell module system. This would require a fair amount of work, but otherwise seems quite feasible. The most difficult task would be to adapt the current interactive user interface to deal properly with collections of program modules. See [15] for work dealing with similar problems, targeted at Scheme programmers.

On the other hand, the Haskell module system has been criticized as one of the weakest parts of the language definition, and may perhaps change in future versions of the language. A more ambitious goal might be to use Gofer as a testbed for alternative module systems, providing, for example, the power of Standard ML style parametric modules, but also preserving the character of Haskell, including type classes, lack of side-effects and call-by-name/lazy semantics. We are currently exploring some proposals for module systems of this kind, although we are still a long way from any concrete implementation.

Records One of the weakest aspects in the collection of primitive datatypes available to the Gofer programmer is the lack of support for any form of records. From a semantic viewpoint, records are just a form of tuple, but to the programmer it is often more convenient to access components of a large tuple by name, rather than by position. Records of one form or another are a standard part of many languages from Pascal to Standard ML, but are not included in the current definition of Haskell. A flexible implementation of extensible records, building on the use of qualified types and hence suitable for inclusion in Gofer, has been proposed in [29]. We hope to experiment with a prototype implementation of this system in the near future.

Performance and optimization Given that execution speed was never a major design Goal, we have been pleasantly surprised by the performance of the Gofer interpreter. However, having developed a system that permits functional programming on small computers, it would be nice if we could also offer the level of performance necessary to make the language a realistic tool for serious application development on those machines. One possibility would be to build a

[15] There are also several cases where it would be desirable to *remove* features. Without making firm commitments to particular features, studies of the current version of the source code will reveal that it is already possible to compile the interpreter without support for `(n+k)` patterns or `Dialogue` style I/O...

more ambitious compiler, targeted at an abstract machine with less interpretive overhead and using more sophisticated optimization techniques.

Lessons learned The Gofer implementation has been pushed far beyond the expectations of the original design, and the source has been extended, patched and modified many times in its relatively short life. In the process, we have learnt many useful lessons about how the design might have been improved. I hope that it will not be too long before someone, perhaps even me, has the opportunity to develop a new, cleaner version of the system, building on this experience ...

Acknowledgements

The preparation of this report was supported in part by a grant from ARPA, contract number N00014-91-J-4043. My thanks to many Gofer users whose frequent questions about the implementation have finally prompted me to write this report. Thanks in particular to Göran Uddeborg and Carl Fosler, for their helpful comments on an earlier draft.

Work on the current version of Gofer began in January, 1991, using an 8086-based home PC and the Borland Turbo C compiler. Development of the system has continued in small bursts ever since, as time permits. Fortunately, I have had the opportunity to upgrade my development system a little since the early days! I know that I have chalked up many hours working on the development of Gofer, its documentation, sample programs, and, most recently, this report. Most of this work has been carried out at home at times that I might otherwise have spent with my wife, Melanie. To her, a special heart-felt thank-you for constant support, encouragement and patience, always.

References

[1] J. Tiuryn A.J. Kfoury and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[2] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[3] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Lecture Notes in Computer Science.

[4] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The computer journal*, 32(2):127–141, April 1989.

[5] Lennart Augustsson. Implementing haskell overloading. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

[6] J.E. Hopcroft A.V. Aho and D. Ullman. *Design and analysis of algorithms.* Addison Wesley, 1983.

[7] R. Bird and P. Wadler. *Introduction to functional programming.* Prentice Hall, 1988.

[8] Stephen M. Blott. *An approach to overloading with polymorphism.* PhD thesis, Department of Computing Science, University of Glasgow, July 1991. (draft version).

[9] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software–Practice and Experience*, 18(9):807–820, September 1988.

[10] R.S. Boyer and J.S. Moore. The sharing of structure in theorem proving programs. *Machine Intelligence*, 7, 1972. Edinburgh University Press.

[11] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, N.M., January 1982.

[12] C. Hall, K. Hammond, W. Partain, S.L. Peyton Jones, and P. Wadler. The glasgow haskell compiler: a retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland*, July 1992. Springer Verlag Workshops in computing series.

[13] Cordelia V. Hall. Using overloading to express distinctions between evaluators. *Information Processing Letters*, December 1993.

[14] K. Hammond and S. Blott. Implementing Haskell type classes. In *Proceedings of the 1989*

Glasgow Workshop on Functional Programming, Fraserburgh, Scotland, 1989. Springer Verlag Workshops in computing series.

[15] Sho huan Simon Tung. Interactive modular programming in Scheme. In *ACM conference on LISP and Functional Programming*, San Francisco, CA, June 1992.

[16] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[17] P. Hudak and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0). Technical report, University of Glasgow, April 1990.

[18] R.J.M. Hughes. Graph reduction with supercombinators. Technical monograph PRG-28, Programming Research Group, Oxford University Computing Laboratory, 1982.

[19] S.C. Johnson. *Yacc: yet another compiler-compiler*. Unix programmer's manual 2B.

[20] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In Jouannaud, editor, *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture*, pages 190–205, New York, 1985. Springer-Verlag. Lecture Notes in Computer Science, 201.

[21] Mark P. Jones. A new approach to type class overloading. Distributed on Haskell mailing list, February 1991.

[22] Mark P. Jones. Computing with lattices: An application of type classes. Technical Report PRG-TR-11-90, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, June 1990. Superceded by [27].

[23] Mark P. Jones, October 1991. Article posted on the internet newsgroup `comp.lang.functional`.

[24] Mark P. Jones. *Introduction to Gofer 2.20*, September 1991. Available by anonymous ftp from `nebula.cs.yale.edu` in the directory `pub/haskell/gofer` as part of the standard Gofer distribution.

[25] Mark P. Jones. *Release notes for Gofer 2.21*, November 1991. Included as part of the standard Gofer distribution.

[26] Mark P. Jones. Type inference for qualified types. Technical Report PRG-TR-10-91, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, May 1991. Largely superceded by [29].

[27] Mark P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, 2(4), October 1992. Updated and expanded version of [22].

[28] Mark P. Jones. Efficient implementation of type class overloading (draft). Superseded by [29, Chapters 6 and 7], March 1992.

[29] Mark P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. To be published by Cambridge University Press. Currently available as Technical Monograph PRG-106, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, U.K. email: library@comlab.ox.ac.uk.

[30] Mark P. Jones. A theory of qualified types. In *ESOP '92: European Symposium on Programming, Rennes, France*, New York, February 1992. Springer-Verlag. Lecture Notes in Computer Science, 582.

[31] Mark P. Jones. Partial evaluation for dictionary-free overloading. Research Report YALEU/DCS/RR-959, Yale University, New Haven, Connecticut, USA, April 1993.

[32] Mark P. Jones. *Release notes for Gofer 2.28*, February 1993. Included as part of the standard Gofer distribution.

[33] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

[34] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[35] S.L. Peyton Jones and P. Wadler. A static semantics for haskell (draft). Technical report, Department of Computing Science, University of Glasgow, February 1992.

[36] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP '88: European Symposium on Programming, Nancy, France*, New York, 1988. Springer-Verlag. Lecture Notes in Computer Science, 300.

[37] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, first edition, 1978.

[38] Konstantin Läufer and Martin Odersky. Self-interpretation and relfection in a statically typed language. In *Proc. OOPSLA '93, Workshop on Reflection and Metalevel Architectures*, September 1993.

[39] J. Launchbury and S.L. Peyton Jones. Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994. To appear.

[40] Xavier Leroy and Michel Mauny. The Caml Light system, version 0.5 — documentation and user's guide. Technical report L-5, INRIA, 1992.

[41] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[42] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* The MIT Press, 1990.

[43] Torben Mogensen. *Ratatosk: A Parser Generator and Scanner Generator for Gofer*, 1993.

[44] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *5th ACM conference on Functional Programming Languages and Computer Architecture*, New York, 1991. Springer-Verlag. Lecture Notes in Computer Science, 523.

[45] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proceedings 20th Symposium on Principles of Programming Languages.* ACM, January 1993.

[46] J. Peterson and M.P. Jones. Implementing type classes. In *Proceedings of ACM SIGPLAN Symposium on Programming Language Design and Implementation.* ACM SIGPLAN, June 1993.

[47] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages.* ACM, January 1993.

[48] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, 1987.

[49] S.L. Peyton Jones and D. Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[50] Geoffrey Seward Smith. *Polymorphic type inference for languages with overloading and subtyping.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, August 1991.

[51] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[52] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[53] D. Volpano and G. Smith. On the complexity of ML typability with overloading. In *5th ACM conference on Functional Programming Languages and Computer Architecture*, New York, 1991. Springer-Verlag. Lecture Notes in Computer Science, 523.

[54] P. Wadler. Efficient compilation of pattern matching. Chapter 5 in [48].

[55] P. Wadler. List comprehensions. Chapter 7 in [48].

[56] P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.

[57] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.

[58] P. Wadler and Q. Miller. An introduction to Orwell 6.0. Technical report, Programming Research Group, Oxford University Computing Laboratory, 1990.

[59] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software—Practice and Experience*, 20(7):719–727, July 1990.

[60] S.C. Wray and J. Fairbairn. Non-strict languages—programming and implementation. *The computer journal*, 32(2):142–151, April 1989.