Bee: an Integrated Development Environment for the Scheme Programming Language

Manuel Serrano
`Manuel.Serrano@unice.fr`
`http://kaolin.unice.fr/~serrano`

Université de Nice Sophia-Antipolis
930, route des Colles, B.P. 145 F-06903 Sophia-Antipolis, CEDEX

Abstract

The Bee is an integrated development environment for the Scheme programming language. It provides the user with a connection between Scheme and the C programming language, a symbolic debugger, a profiler, an interpreter, an optimizing compiler that delivers stand alone executables, a source file browser, a project manager, user libraries and online documentation. This article details the facilities of the Bee, its user interface and presents an overview of the implementation of its main components.

1 Introduction

In a recent provocative paper, P. Wadler (Wadler, 1998) states that functional languages (henceforth FLs) are more than endangered, they are like dead because nobody uses them anymore! In studying the reasons for this failure he reports that FLs implementations which support the needs of programmers with real problems barely exist. Real programming requires real libraries and real environments that are today lacking for FLs. We strongly agree with these arguments. In the past years implementors of FL have mostly concentrated on efficiency. These studies have been successful because nowadays, compilers for FLs deliver programs that are reasonably fast in comparison to C compiled programs (Hartel *et al.*, 1996; Serrano & Feeley, 1996; Leroy, 1998). Unfortunately the good performance of FLs have not been sufficient to enlarge their diffusion. To some extent, programmers don't care about efficiency. This is one of the main lessons we can draw from the success of Java, since until very recently, Java was still slow but yet much more used than FLs. We think that programmers are more concerned with programming facility than with performance. In addition to the programming language itself, we think there are three main needs when programming:

The need for libraries Computing is becoming more and more complex because computers are more and more powerful, allowing them to process more and more tasks. A programming language must enable user interface programming, network programming, operating system programming, etc. These activities require libraries. The key point is that most of these libraries exist but they are not available from FLs. For instance, Motif is a popular graphics library available for the Unix operating system but Motif requires C to be used. Most of the libraries have a C API or, more recently, a Java API. These APIs are not available from FLs. Thus, there are many programs that are easier to write in C or Java. For instance, writing a program that fetches and displays documents from the network is easier to write in Java than in most FLs.

The need for packaging Following the tradition of Lisp, many FL are implemented in closed environments relying exclusively on a *read-eval-print* loop. This kind of implementation has two drawbacks: *(i)* It leads people to think that FLs cannot be compiled, or at least only with difficulty and as thus it leads people to think that FLs are slow. *(ii)* A monolithic *read-eval-print* loop based environment does not comply with the main modern operating systems philosophy. We may classify operating systems into two categories: the GUI class that contains operating systems like MacOS or Microsoft Windows, and the command line interface class that contains systems like Unix. The *read-eval-print* loop approach fits neither of these categories because it acts as another operating system inside an operating system. Useful programs interact with the host operating system (at least for opening and closing files), but the *read-eval-print* loop makes it almost impossible to write programs that meet the spirit of the host operating system. For instance, Unix programs frequently communicate by the means of pipes. This requires small stand alone programs that may be easily and quickly spawned from command line interfaces.

The need for development environments Programming is a complex task made of many small subtasks each requiring a specific tool. Let us suppose a programmer starting a new project under the Unix environment. His first task will be to write a `Makefile` used to compile his program. In order to ease source files browsing it may be convenient to generate a `tag` file (for instance using `etags`) that can be used by a text editor. In order to get a record of the various versions of the program, a revision control system such as `cvs` must be used. When debugging the program he needs special compilation flags and a tool such as `gdb`. If he provides documentation, other tools such as `man`, `texinfo` or a `html` browsers are required. When the program is fixed and it is to be tuned for efficiency, a profiler such as `prof` must be used. This requires other special compilation flags. If the developed software makes use of libraries then library builders such as `ar+ranlib` or `ld` must be used. In short, the development of real software requires the use of at least 10 different tools. Each of these uses its own syntax and its own methodology. This heterogeneity makes these tools difficult for non-expert programmers to use. The goal of an integrated environment is to make all the development tools available and easy to use. For instance, writing a `Makefile` is a typical target for a development environment. From a list of source files a `Makefile` may be automatically generated. This `Makefile` will handle re-compilation, `tags` file generation, profiling compilation, etc. Another example is an online documentation system. If the documentation is available from the source editor and if it simply requires a mouse click to be popped up then it saves time for the programmer. To summarize, an integrated environment helps the programmer to manage most of the tedious parts of the programming activity.

For a period of fifteen years very little research has been concerned with development environments for FLs. Very few implementations for FLs have provided development environments. The logical consequence is that programming in Lisp at the beginning of the 80's was far easier than using FLs today! Nowadays, we have extraordinarily powerful workstations but we use development environments that are much less attractive and much less powerful than the Lisp machines (Moon & Weinreb, 1981; Symbolics Inc., 1981) used to be! Even worse, development environments for functional languages are weaker than development environments for C!

1.1 The Bee, an Integrated Development Environment for Scheme

Experienced programmers have a lot of habits that constitute the basis of their experience. They know the methodology to be used when facing a specific problem. If they don't like FLs, it could be because the tools they are used to are not available for FLs or because the methodology they are used to does not fit well with FLs. We think there is a need to design and implement a development environment that will feel familiar to programmers. This has been the goal of the design of the Bee, an integrated development environment for the Scheme programming language (Kelsey *et al.*, 1998).

Designing and implementing a development environment is a long process requiring a development capacity that is beyond most academic development teams. The development of the Bee has been fast because our implementation strategy was based on reusing existing tools. In order to minimize the implementation effort, we have decided to concentrate on one operating system. The Bee has been designed to fit harmoniously the Unix style of programming which fosters separate compilation, enables user libraries and delivers small stand-alone applications. The development model follows the traditional *edit/compile/debug* cycle. Eventually, when programs are tuned for efficiency, a profiler reports on execution runtime statistics.

The Bee is an open environment. It is open to the C programming language. That is, it is easy to connect Scheme and C code inside Bee which tries hard to minimize the differences between the execution model of Scheme and the execution model of C. For instance, there is no extra cost associated with calling a C function from Scheme and vice-versa. We may also mention that the Bee's memory manager is compatible with the lack of memory management for C. As a consequence, Scheme objects may be used within C functions and C objects may be used within Scheme functions. With Bee, C APIs are available from Scheme code.

1.2 Overview of the article

The first Section is an overview of Bee user interface which focuses its presentation on Bee's look and feel.

Prior to creating the development environment we implemented a compiler for Scheme. This compiler, named Bigloo, compiles Scheme code to C code. Because we wanted the connection between Scheme and C as efficient as possible, we have designed Bee to produce C code that conforms C's standard programming (Cannon *et al.*, 1990). The direct consequence is that compiled Scheme code is strictly compatible with C. Because of this strong compatibility, the regular tools for C may be applied to Scheme code once it has been compiled into C. This has been the major idea for our implementation of the Bee. Instead of developing a brand new environment from scratch, we have been working at adapting C tools to Scheme. Section 3 presents the resulting C code compiled from a Bigloo Scheme module. It shows how Scheme constructions are mapped into C. This section is a requisite to the Sections 4, 5 and 6. Section 4 presents how Scheme code and C code can be mixed. Sections 5 and 6 present how C profilers and C debuggers may be adapted to Scheme. They present how the Scheme compiler provides applications with services that are to be used by the Bee. Finally, Section 7 presents some related work.

2 The Bee user interface

The central tool of an integrated development environment (henceforth IDE) is the text editor. Rather than embedding the editor into the IDE we have chosen to embed the IDE inside the text editor. This avoids wasting time in implementing a text editor that mimics the features of an already existing editor. Because Unix is the initial target system of our IDE, the choice of Emacs as editor was obvious. We have chosen the Xemacs variant (`http://www.xemacs.org`) because it provides more graphics facilities than other Emacs implementations. Figure 1 is a snapshot of a plain Bee editing session.

2.1 Bigloo Modules and the Bee Project Manager

Bigloo extends the Scheme programming language with many constructions. In particular Bigloo extends Scheme with modules which have an important impact to the programming style. Bigloo modules have two basic roles: to allow separate compilation and to increase the number of errors that can be detected by the compiler. For instance, because of modules, the compiler may complain about undeclared but referenced variables. Bigloo Modules are simple and easy to implement. They are represented by one or more files and have the following syntax:

(module *module-name*
  (main *main-ident*)$^?$
  (import *import*$^+$)*
  (export *export*$^+$)*
  (static *static*$^+$)*)

optional-body

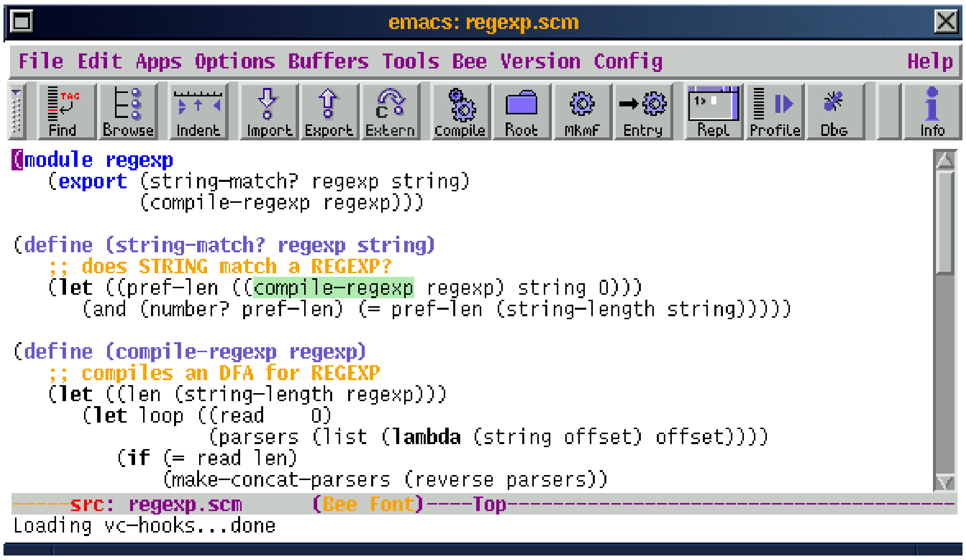


Fig. 1. A plain Ide window

The superscript $^?$ means zero or one occurrence, * means a possibly empty repetition and $^+$ means a non empty repetition.

Main clauses specify the name of the function that will be the starting point of the application.

Import clauses are used to import bindings into the module. In order to import, one just needs to state the identifier to be imported and its module name.

Export and *static* clauses play a dual role. They point out to the compiler that the module implements some bindings and distinguish those that can be used within other modules (they are exported) and those that cannot (they are static). These clauses do not contain simple identifiers, as in import clauses, but information about the exported bindings. It is then possible to export variables (mutable bindings) or functions (read-only bindings). Static clauses are optional (the bindings of a module which are not referenced in a clause are, by default, static).

Let us look at the example of the `fib` module that imports the functions `fib-fx` of module `fib-fixnum` and `fib-fl` of module `fib-flonum` and that exports the function `fib`. The `fixnum?` function is implicitly imported, as are all primitives.

```
(module fib
   (import (fib-fx fib-fixnum)
           (fib-fl fib-flonum))
   (export (fib x)))

(define (fib x)
   (if (fixnum? x) (fib-fx x) (fib-fl x)))
```

During compilation, the compiler performs a number of verifications:

- All referenced variables and functions must have been defined in the module, mentioned in an import clause or be primitive, i.e., there are no free variables.
- All the identifiers mentioned in static or export clauses must have been defined in the module with a value conforming to their prototype.
- Identifiers mentioned in static or export clauses whose prototypes are functions must not be `set!`'d.
- All function calls where the functional operator is known to be a function (i.e., the identifier in a functional position has a function prototype) must have an argument number that conforms to the prototype of the invoked function.

Expressions within the body of a module can be variable or function definitions (with the Scheme form `define`) or any Scheme expression. *Initializing* a module is an operation that binds variables to values and evaluates expressions that are not definitions. Definitions and expressions are evaluated in the order of their appearance in the module. Importation graphs of modules are not restricted to being trees or even acyclic since any module can import bindings from any other. The module initialization order is unspecified. For acyclic graphs, modules are initialized during a depth first traversal starting from the module containing the entry point of the application.

This design for the module system leads to a straightforward implementation. In particular, imported modules do not need to be compiled in order to compile the importing module. Bigloo only uses module clauses to know the exported bindings. As a consequence, Bigloo only needs that the source file of the module exists and contains at least its module declaration.

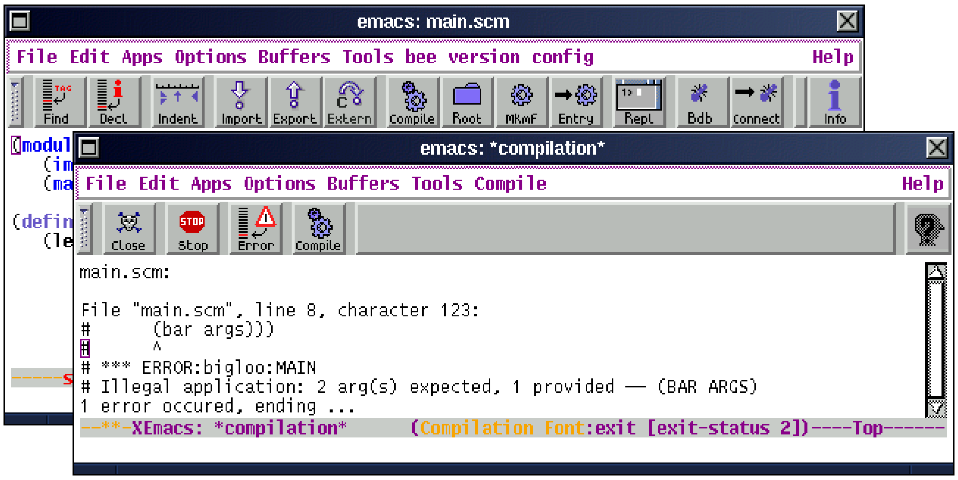
The goal of the Project Manager is to help with both compilation and browsing of modules. Prior to any other operation a project must be registered. Provided with Bigloo modules, clicking the Mkmf icon (see Figure 1) registers a project by creating three extra files. It creates a file that associates Bigloo modules to Unix files. It creates a tag file that is used to retrieve Scheme definitions. Finally, it creates a `Makefile` file that suits the Unix `make` tool. The construction of these three files explores the file hierarchy in order to find out in which files Bigloo modules are implemented. Once a project is registered, all of Bee's features are enabled. Let us consider a very simple Bee session consisting of a Bigloo application made of two modules, `main-module`:

```
(module main-module
   (import bar-module)
   (main   main))

(define (main argv)
   (let ((name (car argv))
         (args (cdr argv)))
      (bar args)))
```

and `bar-module`:

```
(module bar-module
   (export (bar x y)))

(define (bar x y)
   '...)
```

The project is compiled by clicking the Compile icon. Because our example source code is erroneous, this will raise a new window displaying the compilation error status:



Scheme errors are reported in the source code. The compilation status window may be used to pop up a window displaying the culprit line of source code. This feature is common for C environments but unusual for Scheme environments. It requires a compiler that is able to find out source locations from its compilation abstract syntax tree. Section 6.3.1 presents an overview of the implementation of that feature.

Because the project manager automatically handles the association between module name and source file name, there is no need to explicitly specify in which file a module is implemented. For instance, `main-module` imports `bar-module` but the name of the file implementing `bar-module` is not specified within the `main-module` import clause.

2.1.1 Libraries

Bigloo compiles modules into object files and links these files together to build stand-alone executables. The Bigloo object file format conforms with the object file format advocated by the platform hosting Bigloo. The benefit of using regular object files is that it is easy to use the tools of the host operating system to build both static and shared libraries. With Bigloo, users may create their own libraries built of Scheme object files.

2.1.2 Separate compilation

Bee does not recompile an entire project when only a small part of the project changes. To see how this works, let us re-examine the example of Section 2.1 and suppose that `bar-module` has been fixed. Re-compiling the project will spawn the compilation of `bar-module` and, possibly, the compilation of `main-module`. If we suppose that only the body of the `bar-module` module has been modified, then re-building will just require the compilation of `bar-module`. On the other hand, if we suppose that the prototype of one of the functions exported by `bar-module` has been modified, the new compilation will process `bar-module` *and* `main-module`. That is: module coherence is maintained by the project manager.

The technical solution for that coherence is mostly implemented in the generated `Makefile`. A Bigloo module *depends* on the *module checksum object* files, one for each module it imports. The *module checksum object* is a file associated to a Bigloo module that contains a unique number computed from the module clauses. When compiling a module `mod`, Bigloo computes its *module number* $\mathcal{M}$. If $\mathcal{M}$ is different than the number contained in the `mod`'s *module checksum object* `mod.mco`, then a new `mod.mco` file is generated. All modules that import `mod` depend on `mod.mco`. They are recompiled when `mod.mco` changes and `mod.mco` changes only when the clauses of its associated module `mod` change. Even if unlikely two different sets of module clauses may have the same checksum number. This may cause troubles for compiled applications. If modifying the export clauses left the module checksum number invariant, the project manager will fail at ensuring the coherency between modules. An application made of this module could then fail at run time. Using the whole set of import and export clauses instead of a number to represent modules resolves this problem. However, we have decided to use checksum numbers for the sake of efficiency (it is faster to compare two numbers than two complex data structures) and because the probability that a modification to the module clauses left the checksum invariant is extremely low.

2.1.3 Source file browsing

Once we have a registered project, source file browsing is enabled. There are two ways of browsing: using modules or definitions. The modules browser lists all of the modules and provides access to the modules for editing. To use definitions browsing, the user simply puts the editor cursor on an identifier and clicks on the Find icon. The definition browsing is context sensitive. That is, if the cursor is located inside a module clause, then the definition source browser will look for a module definition. If the cursor is located on a type identifier (a Bigloo type identifier is a specific token that looks like `::`*`ident`*), the source definition browser will look for a type or class definition. In all other situations, the browser will look for a variable or function definition. The definition browsing facility is well known by developers using Emacs; the key point here is that the generation of the tags file that enables that browsing has been automated. It does not require any user operation.

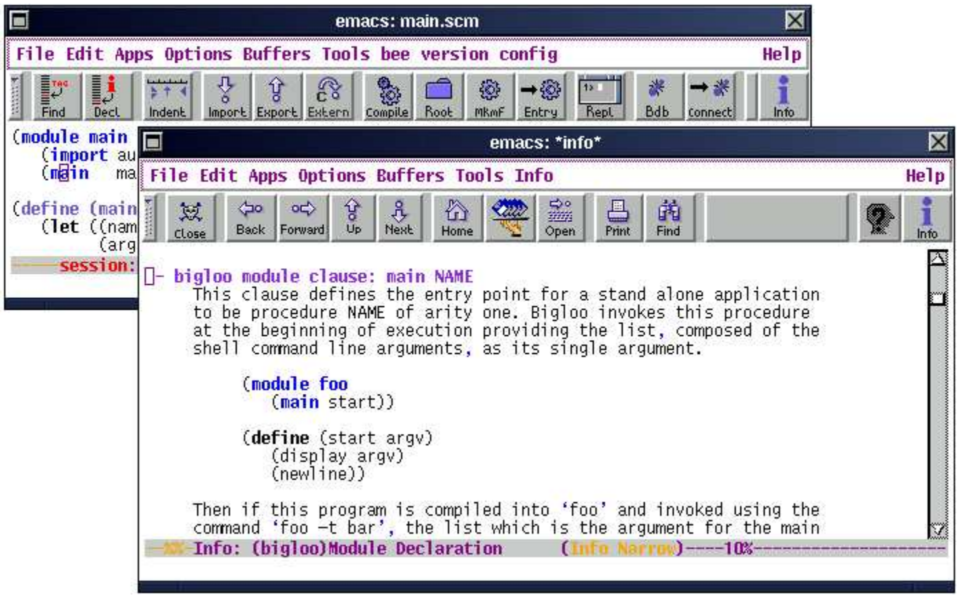


Fig. 2. The online documentation

2.2 Online documentation

Because online documentation is much more effectively indexed, it is far superior to printed documentation. Just like source browsing, Bee's online documentation is context sensitive. When the cursor is located on a source identifier, clicking the Info icon pops up a new window displaying the documentation for the current identifier. Online documentation may be requested while the cursor is above other kinds of tokens. For instance, when the cursor is on a number and the user clicks on Info the general documentation for numbers will be displayed. Figure 2 contains an example of window displaying the *main* module clause documentation.

2.3 Foreign interface

Scheme is a small language. The most up to date version of its documentation is merely 50 pages long (Kelsey *et al.*, 1998). Being so concise is advantageous in some situations such as teaching and formal reasoning but it has the obvious drawback that it cannot describe much of libraries. Actually, Scheme has very few libraries. Bigloo extends Scheme in many respects. For instance, Bigloo embeds a lexical grammar compiler, an algebraic grammar compiler, a pattern matching compiler, an object layer, a library for Unix processes and Unix sockets, and so on. Advanced system programming within Bigloo is implemented by means of the foreign interface. Section 4 presents in great detail the Bigloo foreign interface. The current section focuses on how the user connect's this program across the foreign interface.

Bee's Cigloo tool extracts function prototypes, variable prototypes, compound type declarations, and, to some extent, prototypes of Cpp macros from C files. Cigloo has some limitations due to C source file coding. Some Cpp macros are difficult because they may break the C syntax (Cpp macros not necessarily complies the C syntax), making a parser based tool such as Cigloo inefficient. Also, Cpp macros are not typed. All C definitions are monomorphically typed but Cpp macros may be polymorphic! Bigloo needs types for any foreign declaration and thus, it needs types for Cpp macros. Cigloo attempts to find out types for Cpp macros. When it fails, it assumes the macro arguments type to be C integers. Of course in many situations this assumption is erroneous and a Cigloo user will be forced to fix the Cigloo produced prototypes for Cpp macros. Cigloo can be invoked clicking the Extern icon. This will pop up a file browser that will let the user select the C file to be imported. For instance, let's examine the module `hash`:

```
(module hash
   (export (hash x)))

(define (hash x)
   (cond
      ((symbol? x)
       (hashstring (symbol->string x)))
      ((string? x)
       (hashstring x))
      (else
       ...)))
```

Let's suppose the `hashstring` function to be a C function implemented inside the file `hash.c`:

```
long hashstring( char *string ) {
   char c;
   unsigned long result = 0;

   while( c = *string++ )
      result += (result << 3) + (long)c;

   return result;
}
```

As it will presented Section 4 our Scheme compiler is able to handle C values (such as C `char *`) by automatically converting them into corresponding Scheme objects. Then, clicking the Extern icon and selecting the file `hash.c` will update `hash.scm` as follows:

```
(module hash
   (extern
    ;; beginning of hash.c
    (hashstring::long (::char*) "hashstring")
    (type char*->long "long $(char *)")
    ;; end of hash.c
    )
   (export (hash x)))

(define (hash x)
   (cond
      ((symbol? x)
       (hashstring (symbol->string x)))
      ((string? x)
       (hashstring x))
      (else
       ...)))
```

Automatically, the extern import clause adds the file into the entry of the `Makefile` so that an object file will be produced for `hash.c`. To illustrate the limitation of Cigloo let's study its processing of the C file `point.h`:

```
1: typedef struct pt {
2:    double x, y;
3: } *point;
4:
5: #define PT_EGAL( pt1, pt2 ) \
6:    (((pt1)->x == (pt2)->x) && (((pt1)->y == (pt2)->y)))
```

`point.h` contains a type definition and a macro definition. Cigloo produces:

```
 (extern
   ;; beginning of src/session/point.h
File "point.h", line 5, character 48:
##define PT_EGAL( pt1, pt2 ) \
#^
# *** WARNING:bigloo:define:
Unknown type expression -- Using 'int' type
   (macro PT_EGAL::int (::int ::int) "PT_EGAL")
   (type s-pt (struct (x::double "x") (y::double "y")) "struct pt")
   (type point s-pt* "point")
   ;; end of src/session/point.h
   )
```

As one may see, the macro `PT_EGAL` has not been correctly handled by Cigloo because it got confused with the argument types. As a consequence, the return type for the `PT_EGAL` macro will have to be fixed manually. By contrast, Cigloo correctly

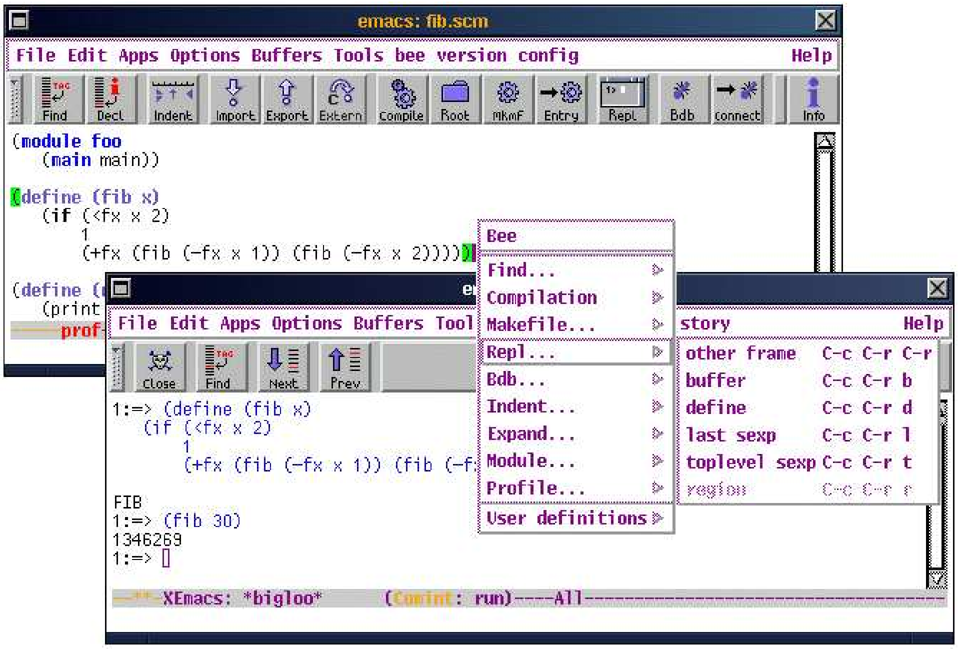


Fig. 3. An interpreter session

handles the type declaration for `point`. Declaring a C structure will cause Bigloo to create some accessors, some mutators, a predicate and a creator for that type. Bigloo programs will thus get the possibility to even allocate `point` C objects. When manipulated from Scheme, a C structure is referenced to via a handle containing a runtime type information and a pointer to the plain C structure.

2.4 Interpreting

A Scheme implementation must provide an evaluation mechanism which can be implemented by means of an interpreter or by online compilation coupled with objects dynamic loading. This feature is mandatory because the language in which macros are written is Scheme itself! This creates the need for an evaluation stage at compile time (or more precisely at macro expansion time). For Bigloo, we have chosen the first solution: the Bee contains an integrated interpreter. That interpreter is spawned by clicking the Repl icon. Expressions should be sent from the editor to the interpreter in various ways. One should keep in mind that the Bee interpreter is just a tool aiming at testing functions or writing macros. Large development projects should not rely on the interpreter which is slow because it is not tuned for performance.

We have tried to make the semantic of interpreted programs as close as possible to the semantic of compiled programs. The interpreter is able to call compiled code and vice versa. This enables the standard compiled Scheme library to be shared between compiled and interpreted code. However we have failed to get rid of all the differences between the interpreter and the compiler. For instance, the evaluation order of the arguments of a function call is left unspecified in Bigloo. Thus, it is unlikely that the arguments are evaluated in the same order if the call is interpreted or compiled. Apart from these small differences the interpreter resembles the compiler.

2.5 Profiling

During the tuning for performance, Bigloo programs may be profiled. This requires very little input from the user. The profile popup menu (within the Bee menu) has two entries, *compile for profile* and *run for profile*. Selecting the *run for profile* entry first asks the user for command line parameters, then spawns an execution, analyses the run sample, and pops up a new window displaying the profiling statistics. For instance, let's show the profile for:

```
(module foo
   (main main))

(define (fib x)
   (if (<fx x 2)
       1
       (+fx (fib (-fx x 1)) (fib (-fx x 2)))))

(define (main argv)
   (print (fib 30)))
```

Profiling this program pops up a window as shown in Figure 4. This tells us that `100.0%` of the execution time (`0.72` seconds) is spent in the `FIB` function and the functions its calls (actually `FIB` only calls itself). `FIB` has been called `1292536` times and amongst these calls only one has been operated from the `MAIN` function. Section 5 details how profiles are computed. Let's just mention that because Bigloo allows mixing of Scheme and C functions, the profiler displays both kinds of functions. They are presented in the profile results using different colors. One user option enables the Bee to hide every non Scheme function. That way only Scheme identifiers are presented to the user.

2.6 Debugging

The last tool we are to present is the Bigloo debugger, BDB which is invoked by clicking the Bdb icon. This pops up a debugging window. Various icons allow stepping, continuing, displaying local variables, function arguments, execution stack. In order to access BDB facilities a buffer must be connected to BDB. A buffer can be connected by clicking on its Connect icon. When the execution stops, the (possibly new) buffer displaying the source line is automatically connected. When a buffer is connected it displays a left margin. That margin is used to set, remove or change

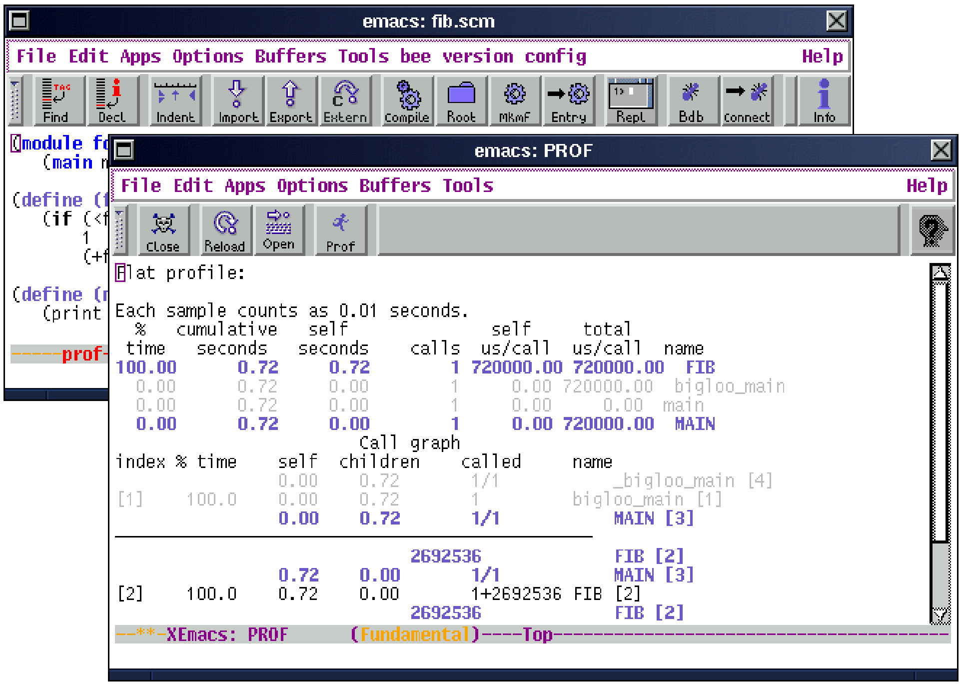


Fig. 4. An excerpt of profiling

the state of breakpoints. Icons in the left margin show the locations of breakpoints and the execution line is highlighted in the source text by the means of an arrow in the left margin. Figure 5 contains a snapshot of a debugging session where two breakpoints have been set: one disabled in the `main` function, one enabled in `fib`. On the snapshot, the debugger window has been split into three. The first one is the command window. Users may submit command line operations from that window. The second one displays the arguments of the current function and the third window displays the execution stack. Clicking on the various frame lines opens new windows displaying the source line corresponding to the clicked activation record. More BDB commands will be presented in Section 6.

3 The C code production

The purpose of this paper is not to present the compilation techniques used to get efficient C code production. They have already been presented in previous work (Serrano & Weis, 1995). But, we must present the framework we use to compile Scheme programs into C programs because this has a tremendous impact on the design and implementation of all the tools that are included in the programming environment.

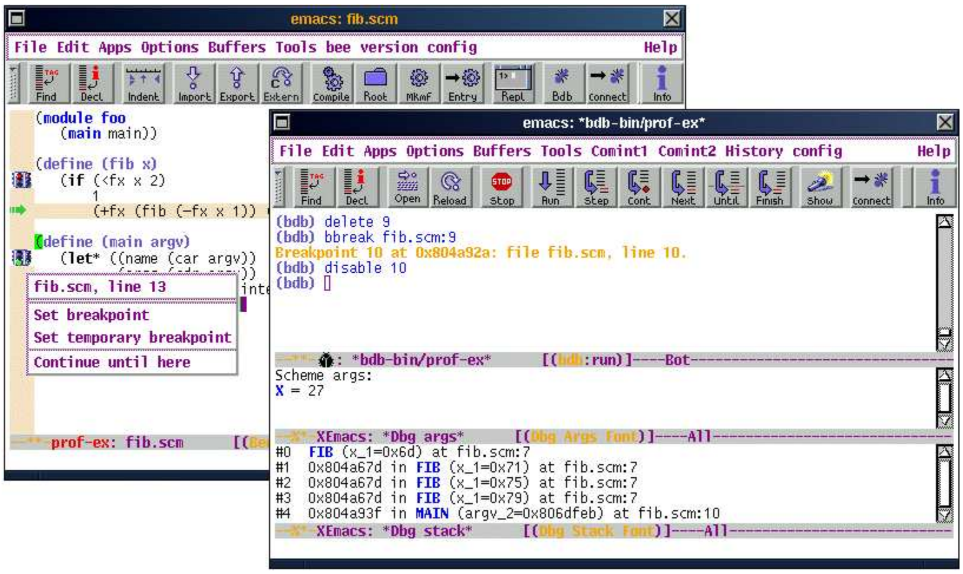


Fig. 5. A debugging session

3.1 What kind of C code to generate?

Because C is mostly portable, choosing to generate C code gives us a mostly portable Scheme compiler for free. There are many variations on the sort of C code which can be produced but two main directions have emerged:

1. the generation of C code that mimics a virtual machine; in this case, the C compiler is considered as a virtual assembly language (Tarditi *et al.*, 1992).
2. the generation of C code that resembles handwritten C code.

Each method has its own advantages and its own drawbacks but we claim that an optimizing compiler must generate “natural” C code (direction 2).

3.1.1 C as a virtual assembly language

If we use C as an assembly language, we abandon the usage of C control structures. In this case, the C code generated by the compiler has no C functions, the C stack is hardly ever used, and C variables just serve to maintain the registers of some abstract machine. That way, source language features that need some knowledge about the runtime behavior of programs are easier to compile (e.g., garbage collection or the Scheme `call/cc` function). Unfortunately: since the C code generated by the compiler bears no resemblance to the one written by C programmers, it is likely that C compilers will fail to optimize this code properly. The numerous optimizations performed by C compilers will probably not apply and the functional language compiler will not benefit from a high quality C compiler.

3.1.2 "Handwritten-like" C code

The alternate C code generation method mimics handwritten C code. By contrast with the preceding approach, the source language features that need to know the runtime behavior of programs are difficult to implement (and even more difficult to implement efficiently). The `call/cc` function is now hard to implement, and memory management is constrained by the presence of C values in the runtime space of the program (for instance C values stored into the stack). Furthermore, garbage collection must deal with ambiguous roots. On the other hand, the compilation of our generated C code is better, resulting in good, homogeneous performance that is not tied to some particular machine architecture. A previous study (Serrano & Weis, 1995) has reported that this technique, in general, delivers better performance than the virtual assembly one. As an added benefit of this "standard C code" generation, we can run the C programming environment tools on the generated C code: symbolic debuggers, code analyzers (such as `purify`) and profilers.

With this method of C code generation, every source language construct is compiled into an equivalent C construct if one exists. For instance, source language functions are compiled into C functions (or even into C loops when the source language functions are tail-call loops), source language variables are compiled into C variables, and so on. This mapping is an example of what we call the "natural mapping" from one language to the other. Bigloo uses this technology and projects Scheme terms to handwritten-like C code. To manage C values in the stack, Bigloo uses the Boehm's garbage collector (Boehm & Weiser, 1988; Boehm, 1996).

3.2 Compiling Scheme functions

The main design effort for Bigloo has been to compile functions as well as possible: in our mind this implies mapping Scheme functions to C functions (or even to C loops) and working hard to avoid heap-allocation of closures. The result is that global Scheme functions are compiled into global C functions and local functions that play the role of loops are compiled into C loops such as in:

```
(define (find-char str char)
   (let ((len (string-length str)))
      (let loop ((i 0))
         (cond
           ((= i len)
            -1)
           ((char=? (string-ref str i) char)
            i)
           (else
            (loop (+ i 1)))))))
```

The resulting C code is:

```
find_char_85_foo( obj str_1, obj char_2 ) {
  long len_10 = STRING_LENGTH( str_1 );
  long i_12 = 0;
  long aux_35;

loop_11:
  if ( i_12 == len_10 )
    aux_35 = ((long) -1);
  else {
    if ( STRING_REF( str_1, i_12 )==CCHAR( char_2 ) )
      aux_35 = i_12;
    else {
      i_12 = (i_12 + 1);
      goto loop_11;
    }
  }
  return BINT( aux_35 );
}
```

Some points have to be outlined:

- The scheme global function `find-char` is compiled into a C global function, `find_char_85_foo`.
- The scheme local function `loop` is compiled into a C loop by the means of a `goto`. Our Scheme compiler does not make any effort to implement the loop by the means of a C `while` construction. There is no need for this because, from optimizing C compilers point of view, `while` constructions or backward `goto` constructions are equivalent.
- Scheme arguments and Scheme local variables (e.g., `str` and `len`) are compiled into C arguments and C local variables.
- arithmetic is locally unboxed and untagged (the C variables `len_10` and `i_12` are of type `long`). That is, locally, numbers are implemented as immediate values that do not contain type information. In particular, no memory allocation is required to hold such numbers. The analysis that enables such code generation has been published in a previous paper (Serrano & Feeley, 1996).

Obviously, such compilation is enabled only when all recursive calls to the local function are tail recursive. If this condition is not met, a C global function is generated. That is, the Scheme code:

```
(define (copy-list-until l val)
   (let loop ((l l))
      (if (or (null? l) (eq? (car l) val))
          '()
          (cons (car l) (loop (cdr l))))))
```

Is compiled into:

```
obj copy_list_until( obj l_1 , obj val_2 ) {
   return copy_list_until_loop( val_2, l_1 );
}

obj copy_list_until_loop( obj val_22, obj l_3 ) {
   if( NULLP( l_3 ) || (CAR( l_3 ) == val_22) )
      return BNIL;
   else {
      obj arg1003_6, arg1004_7;
      arg1003_6 = CAR( l_3 );
      arg1004_7 = copy_list_until_loop( val_22, CDR( l_3 ) );
      return MAKE_PAIR( arg1003_6, arg1004_7 );
   }
}
```

When a function is used as a value, a closure must be allocated. For instance, the composition function `o` builds new closures:

```
(define (o f g)
   (let ((new-f (lambda (x) (f (g x)))))
      new-f))
```

The difficulties come from the fact that `new-f` is returned as a value (hence it cannot be allocated on the C stack) and that `new-f` closes over two free variables (`f` and `g`). Two functions are generated. One for the Scheme `o` global function:

```
obj o( obj f_1, obj g_2 ) {
  obj new_f_12;

  new_f_12 = make_closure( lambda_1, f_1, g_2 );
  return new_f_12;
}
```

And a second implementing the anonymous function:

```
obj lambda_1( obj clo_1, obj x_2 ) {
   obj f_12, g_23, aux_45;

   f_12 = PROCEDURE_REF( clo_1, 0 );
   g_23 = PROCEDURE_REF( clo_2, 1 );

   aux_45 = PROCEDURE_ENTRY( g_23 )( g_23, x_2 );

   return PROCEDURE_ENTRY( f_12 )( f_12, aux_45 );
}
```

Bigloo's closures are arrays. Each closure contains the free variables of the function, pointer to C functions, and an integer to record the function arity. This arity slot is mandatory for dynamically typed languages to ensure the soundness of function application.

3.3 Compiling Scheme variables

Scheme has one unique value space. That is, functions are values in the exact same way as integers are values. Scheme has one unique construction to bind global values. Binding a function to a variable or binding an integer to a variable is done using the same construction. However, in order to get efficient code, functions have to be distinguished from other values. Bigloo makes a distinction between global definitions which bind constants holding functions and those which bind values that may not be functions. Variables holding constant functions that are never changed (i.e., variables declared constant in the module clauses or non exported and never used as first argument of a `set!` form) are said to be functions and are compiled into C global functions. These functions require no runtime initialization and thus, forward references to these functions are safe. Other Scheme variables are compiled into C global variables. As previously seen, Bigloo succeeds in unboxing and untagging values for local variables. Unfortunately this is not possible for global variables. Because module import clauses may contain cycles (see Section 2.1) the value of an imported variable might be used before the module defining that variable is initialized. In such a situation the imported variable contains a special value denoting the *uninitialized* state. Any expression referring to that variable must then dynamically check for the type of that variable's value. Dynamic type checks only apply to boxed or tagged values. Thus global variables must contain boxed or tagged values. The consequence in the produced code is that Scheme variables are C global variables of a special type, called `obj`, that stands for all Scheme values.

3.4 Limits to the portability of the natural mapping

Scheme has its own customs and traditions (one may even think of them as local folklore) that make the natural mapping to C difficult to implement or even inappropriate for some Scheme constructions. Let us detail some of those.

3.4.1 Scheme tail recursion

As stated by the report defining the Scheme programming language (Kelsey *et al.*, 1998) a compliant Scheme implementation is required to be *properly tail-recursive.* A formal definition of proper tail recursion can be found in a paper by Clinger (Clinger, 1998). Obviously this requirement cannot be fulfilled with the natural mapping because C does not provide any mechanism to implement tail recursion. One might consider that limitation too high a price to pay. Solutions to implement tail recursion in C without stack consumption exist but they cannot be used with the natural mapping (Tarditi *et al.*, 1992; Baker, 1994; Feeley *et al.*, 1997). We consider the gains of the natural mapping more important. In addition, since local functions are compiled into C loops, which do not allocate stack activation frames, many of tail recursive functions are correctly handled by our compiler. Of course, it is possible to write a Scheme program that cannot be executed when compiled using Bigloo but in practice, we have never encountered program limited by the Bigloo compilation model.

3.4.2 Scheme exceptions

Scheme surpasses most of the existing exception mechanism with its powerful primitives: `call-with-current-continuation` (or `call/cc` in short) and `dynamic-wind`. In short, `call/cc` enables exceptions to be raised backwards *and* forwards. That is, one may invoke an exception even when being out of the dynamic scope of the exception handler. Implementing `call/cc` within the natural mapping framework is painful. We have been a whisker away from creating a portable implementation for `call/cc`. Unfortunately `call/cc` requires the ability to resume a computation which, for C, means saving and restoring the execution stack. Obviously there is no way to restore the stack in a portable way because C is not even specified as using a stack (ISO/IEC, 1990). A portable implementation for `call/cc` using the natural mapping is a lost cause. However we have been able to give an implementation of `call/cc` that compiles on every platform we have tested (Serrano, 1994). The tricks that are used in our implementation are totally inefficient. The Bigloo implementation for `call/cc` is likely to be the slowest implementation of `call/cc` available. It is possible to speed up many uses of `call/cc` however. Frequently, continuations are only used to escape from computations. That is, the continuations are invoked inside the dynamic extent of the `call/cc` expressions. To implement such continuations, C `setjmps` (or even C `gotos`) are preferred. Bigloo does not implement that optimization. Instead, it proposes alternative escape constructions (`bind-exit` and `unwind-protect`) that are borrowed from the Dylan programming language (Apple Computer & Technology, 1992; Shalit, 1996).

3.4.3 Garbage Collection

The very same problem exists for the Garbage Collector. There is just no portable way to find the collector roots for C. The collector roots are either on the C stack or in the C global variables. Finding the memory addresses for these two spaces is highly machine and operating system dependent. This require perennial efforts to make the garbage collector available for new architectures.

3.4.4 Other features

X. Leroy states several other features of C that make it an inadequate target language (Leroy, 1998): low level arithmetic with only primitive overflow detection, no multi-word arithmetic, no run time error catching, no traps for array bound checking. Some of these limitations can be worked around with specific implementations (e.g., when compiling for debug the Bigloo compiler inserts array bound checks in the C produced code). Some are unsolved (e.g., when the system raises a signal for runtime execution errors such as stack overflow or illegal arithmetic, Bigloo only traps the signal and suspends the execution).

3.5 Name mangling

Bigloo compilation entities are modules. A module encapsulates bindings and top level expressions. Amongst these bindings some are *exported*: that is, accessible from importing modules. The primary role of a module system is to prevent name collision. Two modules may declare two bindings that share a common name as long as these two modules do not import each other and are not both imported in a third module. This is in direct opposition to the C model. C does not have modules. A C binding of a C variable or a C function is either local to a file or global to the entire application. For C, it is illegal to use the same name to define several global variables. Every language owning more complex scoping rules than C must use some name encoding when compiled to C. This name encoding is called *name mangling*. The inverse operation, that is decoding, is called *name demangling*. The most common example of name mangling comes from C++ (Lippman, 1996) which is used in this language to give unique names to class function members. The C++ name mangling cannot be directly used for our compilation model. As we have presented in Section 3.2, it may happen that a local nested function is compiled into a C global function. This C function must have a name that is guaranteed not to be in collision with any other global C name. C++ name mangling is not powerful enough thus we have decided to use our own name mangling. There is no need to present in detail our *ad hoc* technique because it is absolutely banal. The important point here is that common tools embedding C++ name demangling are of no help to us.

3.6 Efficiency of the natural mapping

We must conclude this section presenting the produced Bigloo C code with a short overview of the performance of the natural mapping. The full details are beyond the scope of this paper, so we just present an informal discussion. We think the natural mapping as a reasonable means to get close enough to the performance of pure C programs. Close enough means that Scheme programs written using an imperative style and their C counterparts perform similarly. This is made possible in 1993 by the garbage collector that delivers good performance. As described in an evaluation paper (Zorn, 1993) Boehm's collector (Boehm & Weiser, 1988) had a performance comparable to that of Unix `malloc` implementations. Ever since, the Boehm's collector has continued to improve and it is likely that its performance is now even closer to that of `malloc`.

Two papers describing the Bigloo compiler (Serrano & Feeley, 1996; Serrano, 1997) contain some time figures showing that Bigloo performance is in the best case equivalent to the C performance and in the worst case two times slower.

The second aspect of efficiency is the time it takes the compiler to compile source files. Slow compilation used to be a problem because the C produced files are large but with our modern fast computers this is not a problem anymore. For instance, the full bootstrap of the compiler and its library takes about 10 minutes even though it requires the compilation of about 70000 lines of Scheme code.

4 The connection between Scheme and C

C is the most popular language for modern operating system implementations where *operating system* is used here in a very general sense. For instance, we consider that the window management system to be part of the operating system. With our modern, powerful computers it is a huge handicap for a programming language not to be opened to current hardware possibilities. For instance programming languages must provide a means to implement graphical tasks. For languages other than C, there are two main ways to meet that requirement:

1. To implement, by hand, wrapper libraries for any C desired library.
2. To enable a connection to C.

The first solution has the obvious drawback that any new library will require a new implementation effort. Small development teams cannot afford that effort. The second solution requires an initial development effort but once completed, all C APIs are available.

One of the main interests of the natural mapping is that it enables a full connection between Scheme and C. In this section, we detail the different levels of connection starting with the most common one and ending with the rarest.

4.1 Foreign function interface

Frequently, the connection to C is thought of as a Foreign Function Interface (henceforth FFI). Actually, the FFI is the first step of the connection. It is mandatory but it is not enough. As explained in Section 3.2, Scheme functions are compiled into C functions. Bigloo compiled Scheme functions are thus compliant to the calling protocol fostered by host systems. There is no difference between the activation frame for a Scheme function and the activation frame for a C function. This helps when calling a C function and enables a C function to call Scheme functions. The only difficulty comes from Scheme's name mangling. This problem is solved by a special Scheme construction that is equivalent to the C++ `extern "C"` declaration that disables name mangling.

The remaining difficulty is related to the type system. Scheme types embed runtime type information and are thus distinct from primitive C types. This is an obstacle to the FFI. However, we wanted raw C types to flow around in Scheme programs. We have chosen to address this problem inside the compiler. The internal representation of a type inside the compiler is a data structure containing two fields. A type identifier and a list of converters for that type. A converter is made of three components, a *destination type*, a *type check* and a *type coercion*. The *type check* is

Table 1. *Type conversions of integer types*

		`obj`	`bint`	`long`
`obj`	(check)	`#t`	`(integer?` v`)`	`(integer?` v`)`
	(coercion)	v	v	`(bint->long` v`)`
`bint`	(check)	`#t`	`#t`	`#t`
	(coercion)	v	v	`(bint->long` v`)`
`long`	(check)	`#t`	`#t`	`#t`
	(coercion)	`(long->bint` v`)`	`(long->bint` v`)`	v

the code the compiler must emit for checking the validity of a type coercion. The *type coercion* is the expression that the compiler must emit when coercing values. C types and Scheme types are related by this mechanism of checks and coercions. For instance, let's consider the conversion from Scheme integers (denoted by the type `bint`) and C fix numbers (denoted by the type `long`). We recall here the existence of a general type for Scheme values, `obj`. Every Scheme type is a subtype of `obj`. That is, Scheme `bint` is a subtype of `obj` but C `long` is not. In order to convert the value of a variable v of type `bint` into a value of type `long`, a type conversion such as "`(bint->long` v`)`" is required. If the compiler does not know that v is of type `bint` but if v is known as being of type `obj` prior to the conversion, a type check must be inserted such as:

```
(if (integer? v)
    (bint->long v)
    (runtime-type-error v))
```

Table 1 summarizes the conversions of integer types. The left column contains the *from types*. For each type, the first line contains the type check that has to be applied, the second line contains the coercion function.

Converting a C integer into a Scheme integer is cheap because it is just a tag addition (such as "a C *shift* + a C *or*"). It may happen that the conversion from C to Scheme requires allocations. This arises for instance when converting a C `double` into a Scheme `real` or when converting a C `char *` into a Scheme `bstring`. Of course, for such objects, converting from C to Scheme is more expensive.

Thanks to our type conversion, a Scheme function may call any C function provided the compiler knows how to convert Scheme values into the C function formals type and how to convert the C value of the function return type into a Scheme value.

A Scheme function may also be called from C using the same technique. When declaring a Scheme function the program may contain annotations about the formal types and the function result. The types used in these annotations may be C types. Thus, from Scheme, it is possible to define a function that computes C values. In addition to the "no name mangling" construction, one may write a Scheme function that may be called from C, using ordinary C types.

Here is a small example of Scheme code using the FFI in both directions:

```
(module scheme<->c
   (extern (printf::int (::string ::long) "printf")
           (export (fib "scm_fib")))
   (export (fib::long ::long)))

(define (fib x)
   (if (< x 2)
       x
       (+ (fib (- x 1)) (fib (- x 2)))))

(printf "fib: %d" (fib 30))
```

That program implements a Scheme function `fib` that accepts a C `long` argument and return a C `long` value. That function is made available to C (i.e., it may be called from C code) under the name `scm_fib`. In addition, that program calls a C function `printf` from the Scheme code.

4.2 Extended foreign function interface

Because of the natural mapping, the structure of the compiled Scheme file and the structure of the source Scheme file are similar. The compiled file contains C statements and C expressions. One positive consequence is that it is easy to insert C macros into this compilation framework. Our foreign interface enables C macros. From the point of view of the Scheme compiler, there no difference between a C function and a C macro, except that a C macro may not be converted into a Scheme closure. Then, one may write code as follows:

```
(module scheme<->c.2
   (extern (macro c-islower?::bool (::char) "islower"))
   (export (islower? x)))

(define (islower? x)
   (if (char? x)
       (c-islower? x)
       #f))
```

even if `c-islower?` is implemented using a C macro.

The second step to the connection is to enable variables to be referenced from the foreign language. Once again, because of the natural mapping there is no difficulty for our Scheme compiler to introduce C global variable references. The general type coercion mechanism is used to ensure that C variable values are correctly boxed or wrapped before being used and that Scheme values are correctly unboxed or unwrapped before being set into C global variables. Scheme global variables may be exported to C. The name demangling construction presented for functions also applies to variables. To summarize, here is an example of Scheme code using the various forms of C import clause and C export clause.

```
1:  (module scheme<->c.3
2:     (extern (macro c-eof-object::int   "EOF")
3:             (macro getc::int (::file*) "getc")
4:             (nb-read-char::long        "nb_read_char")
5:             (export *stdout* "scheme_stdout"))
6:     (export *stdout*))
7:
8:  (define *stdout* (current-output-port))
9:  (define (getchar)
10:    (let ((c (getc *stdout*)))
11:       (set! nb-read-char (+ 1 nb-read-char))
12:       (if (= c c-eof-object)
13:           #f
14:           (integer->char c))))
```

The Scheme global variable `*stdout*` is exported to C where it is known as `scheme_stdout`. That variable has the `obj` type. A C global variable (`nb_read_char`), a C macro (`EOF` and a C function (`getc`) are made available to Scheme by means of the `extern` module clause line *2*. As the example illustrates line *11*, C global variables can be set from Scheme code.

4.3 Exhaustive connection

In the previous sections we have presented how C functions and C variables may be mixed up in Scheme code and *vice versa.* The limitation of the connection comes from the C types the Scheme compiler is aware of. Until now, we assumed that the Scheme compiler has some knowledge of every primitive C type (e.g., `char`, `int`, `long`, `char *`). Moreover, the compiler knows how to convert most Scheme values into values fitting the C primitive types. In addition, Bigloo knows how to build C compound types and how to convert values using these new constructed types. Let us suppose, for instance, a C program declaring and using a structure:

```
struct point {
   int x, y;
};

struct point *print_point( struct point *p ) {
   printf( "<point: %d, %d>", p->x, p->y );
   return p;
}
```

C `struct point` values may be allocated, filled and modified from Scheme. A Scheme module that wants to use the C `struct point` type has simply to declare it as follows:

```
(module scheme<->c.4
   (extern (type s-point
                 (struct (x::int "x")
                         (y::int "y"))
                         "struct point")
           (pp::s-point (::s-point) "print_point")))

(let ((p (make-s-point)))
   (p-x-set! p 3)
   (print (point? (pp p))))
```

The compiler automatically creates getters, setters and a type predicate for each new C type. An interesting feature comes from the ambiguous garbage collector we are using (Boehm & Weiser, 1988). Because the collector seeks objects even if they are only pointed to by the C stack or C variables, the `struct point` is allocated inside the Scheme heap.

5 The Profiler

Because each global Scheme function is translated into a global C function, assuming functions are not inlined, designing a profiler for Scheme is nearly as simple as designing a name demangler for our generated C code. One requirement of the implementation of our profiler, named BPROF, was not to modify the source code of any regular profiler, for the sake of simplicity and portability. As a consequence, BPROF works on any operating system which provides tools such as the Unix PROF, GPROF or similar commands.

Applications compiled for profiling, in addition to their regular computation, produce a file containing a name demangling table. With the Unix C model, a profiled execution computes a value and produces a call graph profile file (e.g., `gmon.out`). Our Scheme profiled executables produce a call graph profile file and a name mangling file. Then, BPROF simply invokes a regular C profiler, parses its output and demangles identifiers according to the demangling file. BPROF uses the demangling service provided by the application itself.

For instance, let us consider the following Scheme program:

```
(define (fib x)
   (if (<fx x 2)
       1
       (+fx (fib (-fx x 1)) (fib (-fx x 2)))))

(define (main argv)
   (print (fib 30)))
```

After compiling this program for profiling and running it, GPROF produces the following profile file:

```
% time     self  children    called     name
           0.00    0.58       1/1           main
 100.0     0.00    0.58       1         bigloo_main
           0.00    0.58       1/1           main_109_foo
-----------------------------------------
                       2692536              fib_1010_foo
           0.58    0.00       1/1           main_109_foo
 100.0     0.58    0.00       1+2692536 fib_1010_foo
                       2692536              fib_1010_foo
-----------------------------------------
           0.00    0.58       1/1           bigloo_main
 100.0     0.00    0.58       1         main_109_foo
           0.58    0.00       1/1           fib_1010_foo
```

Indentation is on purpose. It is intended to help the reading of large profile files. The name mangling table of `foo` is:

```
(MAIN                      "main_109_foo")
(FIB_1010                  "fib_1010_foo")
```

Thus, invoking BPROF produces:

```
% time     self  children    called     name
           0.00    0.58       1/1           _bigloo_main
 100.0     0.00    0.58       1         bigloo_main
           0.00    0.58       1/1           MAIN
-----------------------------------------
                       2692536              FIB
           0.58    0.00       1/1           MAIN
 100.0     0.58    0.00       1+2692536 FIB
                       2692536              FIB
-----------------------------------------
           0.00    0.58       1/1           bigloo_main
 100.0     0.00    0.58       1         MAIN
           0.58    0.00       1/1           FIB
```

We have simplified a little bit these excerpts to fit the article format. Nevertheless what one should notice is that non-Bigloo functions (such as `bigloo_main` which is a library function implemented in C) are left in the profile output. We have made this choice because we have designed Bigloo to foster applications using both Scheme and C. In that context accessing C functions profile information is relevant. For instance, this choice makes the garbage collector execution visible in the profiler result.

In its current version, BPROF is implemented with about 164 lines of Scheme code. The specific profiling code inside the compiler is less than 100 lines of Scheme code. Thus, given a compiler which produces C code using the C calling conventions, writing a profiler for that compiler is not even a half day's job!

6 The Debugger

First, we present a short overview of the debugger commands followed by its global architecture. This will be followed by a focus on the main implementation difficulties where we show how source line stepping is implemented, how breakpoints are set and so on.

6.1 Overview of the debugger, BDB

We start presenting an overview of the debugger, which gives the reader a precise idea of what can be achieved using it.

6.1.1 A summary of the debugger commands

In the same spirit as GDB (Stallman, 1988), BDB is a command line debugger. Its graphical interface is described in Section 2.6. In short, BDB proposes the exact same commands as GDB does, e.g., `break` to set a breakpoint, `step` to step on source line of code, `continue` to resume an execution, etc. The stack may be unwound by means of the `frame` command. For each stack frame, the value of the formal parameters and the local variables may be displayed. The dynamic type of a Scheme value may be displayed. When the program stops, Scheme expressions may be evaluated in the environment of the program in use at the breakpoint location. Instead of presenting a whole debugger manual, we present now a very short sample session. Consider this short Bigloo program:

```
(module dbg-example
   (main main))

(define (fib x)
   (if (< x 2)
       1
       (+ (fib (- x 1)) (fib (- x 2)))))

(define (main argv)
   (print (fib (string->integer (cadr argv)))))
```

Also consider this annotated excerpt from a debugging session (italicized sentences provide commenting, they are not input to or output from the debugger):

```
$ bdb dbg-example
Reading symbols from dbg-example...
```
At this point we may set a breakpoint into the `fib` *function*

to be continued...

```
(bdb) break FIB
Breakpoint 1 at 0x804aa6e: file dbg.scm, line 4
```
We start an execution.
```
(bdb) run 30
Starting program: dbg-example 30
Breakpoint 1, FIB(...) at dbg-example.scm:line 4
4  (define (fib x)
```
The debugger prints the source line where the execution stopped.
Then the execution stack is printed.
```
(bdb:FIB) info stack
#0  FIB(...) at dbg-example.scm: 4
#1  MAIN(...) at dbg-example.scm: 9
#2  bigloo_main(...) at dbg-example.scm: 1
#3  main(...) at dbg-example.scm: 1
```
The value of the parameters of the current stack frame are printed.
```
(bdb:FIB) info args
X = (BINT) 30
```
Scheme expressions are evaluated.
```
(bdb:FIB) print (APPLY + (CONS X '(2)))
32
```
Execution is resumed.
```
(bdb:FIB) step
...
```

6.1.2 A dual mode debugger

The same BDB commands are used to inspect Scheme and C activation frames and to display Scheme and C expressions evaluation. However evaluating and displaying Scheme and C expressions require different operations. For instance, name mangling and demangling must take place for Scheme expressions. To let BDB discover when mangling or demangling is necessary, we have chosen the convention that Scheme identifiers are composed of mixed case, as long as there is at least one lowercase character. This explains why setting a breakpoint in the Scheme `fib` function is typeset `break FIB`. Using `break fib` would set a breakpoint in a C function called `fib`. On the other hand, when a stack frame is displayed BDB checks if the function of that frame is implemented in Scheme or in C. A function is implemented in Scheme if the function's identifier is found in the Scheme symbol table. When displaying Scheme stack frame, the function identifier and the formal parameters are demangled.

It is important that both Scheme and C inspection operations are available from

BDB because the Bee advocates the integration of Scheme and C programs. Thus it is consistent to have a debugger that is either able to step into Scheme or C source text. This is reason why BDB does not hide activation frame that are associated to C functions. We'll make a small modification to the example of Section 6.1.1 by supposing that `fib-2` is a function defined in C:

```
 1: (module dbg-example
 2:    (extern (fib-2::int (::int) "fib_2")
 3:            (export fib "fib"))
 4:    (export (fib::int ::int))
 5:    (main main))
 6:
 7: (define (fib x)
 8:    (if (< x 2)
 9:        1
10:        (+ (fib-2 (- x 1)) (fib-2 (- x 2)))))
11:
12: (define (main argv)
13:    ...)
```

Let us spawn a new debugging session:

```
Let's set a breakpoint into the fib_2 C function
(bdb) b fib_2
Breakpoint 3 at 0x804a294: file cdbg.c, line 4
Let's start an execution.
(bdb) run 30
Breakpoint 3, fib_2 (x=29) at cdbg.c:4
Execution stack is:
(bdb:fib_2) info stack
#0  fib_2(...) at cdbg.c:4
#1  FIB(...) at dbg.scm: 7
#2  MAIN(...) at dbg.scm: 12
#3  bigloo_main(...) at dbg.scm: 1
#4  main(...) at dbg.scm: 1
Because the execution stopped inside a C function, inspecting the current stack frame
displays informations relative to the current C function.
(bdb:fib_2) info args
x = 29
```

6.2 The debugger architecture

In order to get portability and availability when designing the debugger, we chose to reuse existing C debuggers. That is, instead of writing a new debugger from scratch, we wrote a front-end for an existing debugger. We deployed the same strategy as DDD (Zeller & Lütkhaus, 1995). The reasons for this choice are twofold:

- Fully featured C debuggers exist. For instance, the GNU debugger GDB implements most of the common features of the C debuggers.
- Most of all we wanted to avoid wasting time in re-implementing C debugger machinery such as breaking, stepping and continuing.

When debugging a Bigloo program three processes are spawned. One process for BDB, another for GDB, and the last one corresponding to the debugged process (henceforth called the patient). An overview of the debugger architecture is presented in Figure 6.

6.2.1 BDB *connected to* GDB

BDB spawns a GDB process that will, in turn, spawn the patient process. BDB takes over GDB inputs and outputs. That is the command line is read by BDB and all outputs are parsed by BDB before being printed out. All information relative to the execution are held by GDB. Thus, printing any information relative to the execution state requires a communication between BDB and GDB. Printing the execution stack frame is a good example. When the user invokes the BDB "`info stack`" command, BDB sends the command to GDB, reads back the whole execution stack frames and filters that list according to the Scheme symbol table.

6.2.2 BDB *connected to the patient*

In various situations BDB must also request information from the patient process. A first example is the way BDB accesses the global Scheme symbol table. GDB is able to retrieve the symbol table of an executable. This symbol table does not correspond to the Scheme symbols that compose the program because of name mangling (see Section 3.5) and because many unique Scheme symbols are compiled into two C symbols (such as Scheme closures) leading to extra C identifiers. The actual Scheme symbol table is located inside the patient static area. Using a framework close to the one used for profiled compilations, the Scheme compiler inserts into the executable a piece of data that represents the Scheme symbol table. When BDB intercepts a symbol in a GDB output, it checks to see if that symbol has to be demangled. A symbol has to be demangled if it is associated to a Scheme binding. BDB maintains a local Scheme symbol table that acts as a cache. If a symbol is not found in its cache, BDB requests the global Scheme symbol table from the patient by the means of the GDB `call` command. The result of that operation is stored in the BDB Scheme symbol table cache.

Another situation where BDB must ask for services from the patient is when a Scheme expression must be evaluated. This evaluation is not performed by the debugger but instead by the patient. As with the symbol table, the request for evaluation is sent to the patient via the GDB `call` command. This technique prevents the debugger from embedding its own evaluator and, more importantly, it ensures that the evaluation environment will be that of the patient. As we have already seen for profiled executions, debugged applications embed services that are to be used by the Bee.

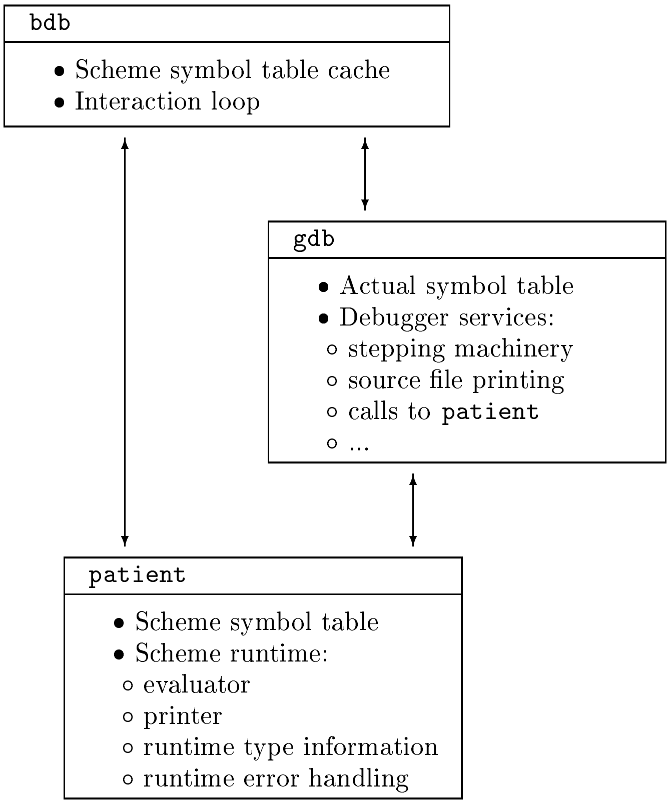


Fig. 6. The debugger architecture

6.3 Source line stepping

As with traditional C debuggers, the granularity of stepping is the source line. That is, the command "`step`" executes until the execution thread reaches a new source line's code. The goal of the BDB implementation is to mimic as much as possible the GDB behavior when stepping. The difficulty to be faced is specific to the Scheme programming language. It consists of keeping track of the original source file position in the compiler abstract syntax tree.

6.3.1 Source file positioning in the compiler abstract syntax tree

One elegant aspect of Scheme is that the abstract syntax tree for the first compiler stage is a subset of Scheme's values. During macro expansion, a program is represented as an s-expression, which is either an atom or a pair of s-expressions. A macro is thus a function that takes an s-expression and returns a new s-expression as a result. The direct consequence is that there is no place in this s-expression to store source file information. Several solutions exist to solve this problem. We have implemented one within our compiler that is rather tricky. We describe it very briefly.

We have added one new primitive data type: the `extended pair` which is a subtype of the regular `pair` type. An extended pair is a pair with an additional slot, the `cer`. The reader builds s-expressions using `extended pairs` instead of normal pairs and the `cer` slot is filled with the source file position. Because `extended pair` is a subtype of `pair`, the s-expressions built by the reader are compatible with normal s-expressions and thus, may be used within macro expansion.

There are some limitations to this solution. For instance, we are not able to find out the position of atoms in the programs. Also, if all the expressions of a program are rewritten by user macro expansions it is likely that the compiler won't be able to find many source line locations. In practice, however, our solution seems sufficient. We have just taken care when implementing the compiler macros to re-use as many lists read from the source code as possible. Avoiding the allocation of new pairs allows the macro expansion not to discard the source file position contained in the `cer` slots of lists build by the reader. After macro expansion, the program is a mixture of plain pairs and extended pairs. The first stage of the compiler spreads into plain pairs the source file location of their nesting extended pairs. The result is that unannotated expressions are assigned the location of the last tracked location. Thus, locations for macro generated expressions are the locations of the calls to the macro.

Let us illustrate locations propagation over macro generated expressions with the simple example:

```
 1: (define-macro (when test . body)
 2:    `(ifr ,test (begin ,@body)))
 3:
 4: (define (display-point pt)
 5:    (when (point? pt)
 6:       (display "<point ")
 7:       (display (point-x pt) pt)
 8:       (display ", " pt)
 9:       (display (point-y pt))
10:       (display ">")))
```

There are two errors in this program. The `if` construction is miss-spelled `ifr` in line *2*. The `display` optional output port argument has the wrong type line *7*. When compiling Bigloo reports:

```
File "foo.scm", line 10, character 170:
#   (when (point? pt)
#   ^
# *** ERROR:bigloo:IFR
# Unbound variable -- IFR
```

That is the error inside the generated text is located at the macro call site. When this first error is fixed and the program is re-executed, Bee reports:

```
File "foo.scm", line 13, character 248:
#      (display ", " pt)
#      ^
# *** ERROR:bigloo:#{POINT 2 3}
# Type ‘OUTPUT-PORT’ expected, ‘STRUCT’ provided -- #{POINT 2 3}
```

The error is correctly located because the expression `(display ", " pt)` is not *generated* by macro expansion. It is only *inserted* by the expansion of the call to `when`. The expression holds its own source location which was propagated by the macro expansion stage.

6.3.2 Source file positioning in the executable file

Once source file locations are held by the compiler, it is very easy to propagate them into the executable file. As our compilation framework consists of producing C files, we simply use the Iso C (ISO/IEC, 1990) "`# line`" facility which allows source position annotation inside source programs. Given a Scheme source file, the resulting C program, annotated of many "`# line`" directives, is hardly human readable but this technique succeeds at propagating Scheme source file information into executables. Because one Scheme function may be compiled into two C functions, it might happen that two C functions are defined at the same virtual source line of code. GDB is not always able to correctly handle such a situation and we have been obliged to implement a workaround. This is presented in Section 6.4.

6.4 Breakpointing

When provided with fully C "`# line`" annotated programs, C compilers are able to deliver executables which embed debugging information that allow source line debuggers such as GDB to step, break and resume execution correctly most of the time. In theory, the technique presented in Section 6.3.2 is sufficient to enable GDB to break, step and resume inside Scheme programs. In practice, because of some GDB implementation limitations, it is not that simple! When execution reaches a breakpoint that is set at the beginning of a function definition, it may happen that GDB stops the execution before the activation frame is completely initialized. This depends on the platform GDB is running on and how the breakpoint has been set up. Stopping execution before the activation frame construction is completed has one nasty effect: if inspected before stepping one assembly instruction from the breakpoint position, the actual arguments of the function contain invalid values. Although annoying, when debugging C programs this is not a real disaster because the actual arguments are fixed up with one single "`step`" issue. When debugging Scheme programs the problem is much more severe. In order to be displayed, Scheme objects have to be correctly tagged or boxed. When requesting the printing of an object located at a certain memory location that does not correspond to a legal object (as it might happen with a dangling pointer), the runtime system is likely to crash. Stopping execution at a position where arguments are not correctly set up is unacceptable for Scheme. Fortunately, two simple tricks are enough to fix the problem.

GDB is not able to correctly set breakpoints when they are inserted at the first line of a function definition or when they are inserted at a memory address that corresponds to a pointer to a function (such as the GDB command "`break *(&foo)`"). Fortunately, GDB is able to behave correctly when function identifiers are used instead of source lines. Let us assume a function `foo` defined in a file `foo.scm` at line *27*. Setting a breakpoint using the command "`b foo.scm:27`" may fail but "`b foo`" succeeds! As a consequence the solution is easy for us: BDB maintains a hash table of all Scheme definitions where the user may set a breakpoint. When a breakpoint using the line position is requested, BDB checks in its hash table if a function definition is associated to that line position. If it does, the breakpoint to the source line number is turned into a breakpoint to the function identifier. The same solution applies when breakpoints are set to memory addresses.

The second alternative trick is even simpler. GDB gets confused when breakpoints are set at the very first lines that start function definitions. If breakpoints are set at the first line of the function bodies then GDB breaks correctly. The solution is to tailor the C code generation to make the function definition start at a fake line. For instance, if the Scheme code is:

```
27:  (define (foo x)
28:     (bar x))
```

The compiler must not generated such a code:

```
#27 "foo.scm"
obj foo_foo(obj x_1) {
#28 "foo.scm"
...
```

Because using the GDB command `b foo.scm:27` makes GDB confused. However if we change the C code for:

```
#99999 "foo.scm"
obj foo_foo(obj x_1) {
#27 "foo.scm"
{/* a dummy statement /*}
#28 "foo.scm"
...
```

Then, the command `b foo.scm:27` won't stop the execution at the function definition location but at the first line of the function body. That way GDB is able to correctly build the activation frame for `FOO`. Bee implements the last solution.

6.4.1 Stopping inside local definitions

A similar problem with variable initialization arises when the execution thread reaches a local variable definition such as in the following when execution reaches line 7 (we remind that Scheme local variables are compiled into C local variables):

```
6:  (define (foo x)
7:     (let ((y (+ 1 x)))
8:        (print y)))
```

Variable `y` will be available for printing only when fully initialized. A new solution is required for such a situation and, this time, the help of the compiler is required. The solution consists of adding dummy code labeled with the same line number as the source line of `y`'s definition that will make the execution stop before `y` is bound. Then, when using command such as "`info locals`", `y` won't even be defined before line *8*. The C code emitted contains a simple dummy variable definition:

```
/* foo */
#6 "foo.scm"
obj foo_foo(obj x_1) {
#7 "foo.scm"
   int bigloo_dummy_bdb; bigloo_dummy_bdb = 0; {
#7 "foo.scm"
      obj y_3 = plus_r4_numbers_6_5( BINT(((long) 1)), x_1);
...
```

6.4.2 Breakpointing into closures

The last difficulty with breakpointing comes from closures. Because Scheme has closures, it is mandatory for the debugger to have the ability to set breakpoints in closures. Here is the method used within BDB for this.

1. When the user requests a breakpoint in a closure, BDB asks the patient for the address of the C function associated with that closure (see Section 3.2 for a short description of closure compilation) and the breakpoint is set in that function.
2. Setting a plain breakpoint in the C function associated to the closure is not enough because the execution will stop each time it reaches this C function even when entered from another Scheme closure associated with that same C function. Thus, BDB does not set a plain breakpoint but a conditional breakpoint. Because the first argument of a C function associated to a closure is the closure itself, the condition for the breakpoint is a simple test that checks that the first argument of the closure is the closure that has been used to set the breakpoint.

6.5 Inspecting Scheme variables and evaluating Scheme expressions

When execution stops in a function implemented in Scheme, commands to inspect local variables and function parameters may be issued. These commands require the collaboration of BDB, GDB and the patient. To support this, BDB asks GDB for the current frame and the current local variables and arguments. From the current stack frame, BDB deduces the function where execution stopped. From that function and with its symbol table, BDB is able to retrieve the Scheme name associated with the function and the list of local variables. This list is an association list that associates C names to Scheme names. According to that list BDB is then able to build the current local Scheme environment. This local environment is used to evaluate Scheme expressions. Prints and evaluations are performed by the patient not BDB itself. BDB calls the patient via GDB and the `call` primitive. The called function is the patient evaluation function. The arguments to the `call` command are made of the local environment that is built from an association list that associates Scheme identifiers to memory addresses.

6.6 Pros and cons of the Scheme symbol table implementation

BDB maintains a partial copy of the Scheme symbol table that acts as a cache. That table stores information relative to name mangling. It is partial because it may lack some symbols. When BDB checks for an identifier, if it does not find the identifier in its cache, it requests the patient for a consultation of the complete global Scheme table. The result of that consultation is stored in the local cache. The cache is flushed each time a BDB `file` command is issued (the `file` command re-loads a binary executable file).

This framework presents some qualities:

- Unless all symbols are needed by user commands, the complete Scheme symbol table is not duplicated. It only resides in the static memory area of the patient.
- BDB loading of Scheme symbols is lazy and it only takes place when GDB stops an execution. That is, requests to the PATIENT are raised during interactions with the BDB user.

On the other hand, it has an important drawback:

- Requests are sent to the patient by the means of the GDB `call` command. This command can *only* be issued when a patient execution is running. In other words, before the execution is initiated, BDB cannot access the patient global symbol table. The direct consequence is that it is not possible, for instance, to set a breakpoint using the identifier of a Scheme function before the execution is started because at that point BDB is unable to process name mangling for that identifier. In short, the traditional debugging sequence `break main; run` is not accessible to BDB!

Previous version of BDB loaded the entire table whenever the patient was restored, but this was too slow so we implemented lighter weight strategy in the latest version o BDB, Bee automatically generates one file describing the global symbols of the source code. That file is used to allow fast browsing over the source files (see Section 2.1.3). BDB builds an early version of its Symbol table cache by simply loading that file each time a `file` command is issued! That way, it has been possible to get rid of the complexity of spawning fake executions to download the global symbol table while enabling early breakpoints using Scheme identifiers.

6.7 Concluding remarks on the debugger

Although Bee's profiler is much smaller, the debugger is till relatively small, 5800 lines of Scheme code + 500 lines of C for the implementation of the debugger and 600 lines of Scheme code inside the compiler. Compared to the half a million lines of C code in GDB 7000 lines is quite small. Designing and implementing a debugger is a complex task. Thus, even the smartest design won't succeed in hiding all the complexity of the implementation. We have now acquired the conviction that our global debugger architecture was the only one allowing us to have a complete debugger for a couple of months' effort. We have shown in this section that our debugging technique has drawbacks that have required rather tricky solutions. On the other hand, it has two enormous advantages. First, all the machine specific part for the debugger has been re-used. Second, we benefit from the *savoir faire* of the whole GDB team for implementing breakpoints, stepping watchpoints, etc. Compared to the task of writing a debugger completely from scratch, the small shell we have written is worth the effort.

7 Related work

Wadler's statement that implementations for FL hardly ever provide full-featured development environments came as no surprise to us. We know of three exceptions: Ericsson's Erlang (Ericsson, 1998), Harlequin's ML Works (Harlequin, 1998) and Allegro Common Lisp Composer (Franz Inc., 1998). Implementing an IDE from scratch requires a huge and time-consuming effort. This probably explains why very few academic projects concentrate on programming environments. One of the most significant is PLT (Findler *et al.*, 1997). Before comparing development environments, we provide a comparison of connection techniques between Lisp-like languages and C.

7.1 Lisp and C

Three main papers describe a connection between Lisp and C:

- The Rose and Muller paper (Rose & Muller, 1992) describes the *Embeddable SHell* foreign interface. The ESH project aims at using Scheme for implementing a glue language. The foreign interface goal is to allow ESH to use all C types. The ESH runtime allows ESH objects to have a layout as close as possible to C objects. Unfortunately, the paper is missing many implementation details, so it is difficult to understand if their runtime decisions enable fast code.

- H. Davis, P. Parquier and N. Séniak have designed a rich interface between Ilog Talk and C++ (Davis *et al.*, 1994). The integration takes into account the object layer of the two languages. It seems that Ilog Talk offers the same facilities as the Bee, the difference coming from the compiler. The Bigloo compiler generates stub code for foreign objects. The Talk compiler does not embed a stub generator and connection between C++ and Talk requires extra C function calls. The goal of Bigloo was to provide a very cheap connection between Scheme and C. The solution of Ilog Talk does not meet this goal.
- G. Attardi has presented an implementation of Common Lisp enabling connection between Lisp and C (Attardi, 1994). This implementation uses a compiler that produces C code that mostly conforms to the standard C programming rules (Cannon *et al.*, 1990). The foreign connection relies on the sharing of a common runtime library. The Lisp compiler has no knowledge of any kind of C types, which is the main difference with our work.

7.2 DrScheme

The only integrated environment for Scheme we know of has been developed by the PLT team of Rice University (Findler *et al.*, 1997; Felleisen *et al.*, 1998). They have released an environment named DrScheme which is designed for teaching. It is not a true integrated environment yet because it does not contain project management, profilers, or a dynamic debugger.

DrScheme relies on a graphical user interface for editing and interactively evaluating Scheme programs. It helps programmers to understand their source code by providing syntactic help where, for instance, a program's syntax may be checked and, on success, DrScheme is able to draw transient arrows from an identifier introduction to its uses and vice versa. Programs need to be completely checked for DrScheme to be able to display the arrows. It should be possible to take advantage of Scheme syntax in order to make some kind of partial parsing and to start displaying information even when the whole program is not present. Nevertheless, we must acknowledge that DrScheme has nice features that are currently missing in the Bee. The most innovative feature of DrScheme relies on its embedded static debugging analyses (Flanagan *et al.*, 1996; Flanagan, 1997). The key idea is to use a *set based analysis* (i.e., an abstract interpretation) for static debugging. This usage of abstract interpretation is rather new. The very few systems we know that use abstract interpretation do so for optimization purposes (Rozas, 1992; Serrano & Feeley, 1996). These analyses have not been designed to help the understanding of the source code. This new usage of analysis for program understanding seems very promising and it will be welcome if new research work presents static analysis for program comprehension (Pessaux & Leroy, 1999).

7.3 Java Development Environment

To some extent, Bee is inspired by the Java Development Environment (henceforth JDE) (Kinnucan, 1998). The JDE is an integrated environment for the Java language entirely implemented in Emacs Lisp. JDE and the Bee have lot of similarities. The JDE relies on the Sun Java Development Kit. The JDE enables source file browsing by the means of regular Emacs packages. For portability reasons JDE has not chosen between Emacs and Xemacs, that is, it runs on both. We think this is an error because it prevents the JDE from using any graphical interface. For instance, when debugging, it is preferable to set a breakpoint using a mouse rather than using a command line interface.

7.4 Allegro Common Lisp

Allegro Common Lisp (henceforth ACL) has two different integrated environments. The first one, called "ACL Composer", is not available within free distributions. As a consequence we have not been able to test it. Its second IDE is based on Emacs. Even if the choice of Emacs seems familiar, the approach used in ACL's IDE is different from the one used in the Bee. Many facilities are shared by the two environments (for instance, tags files which allow quick source file browsing are handled by both environments) but ACL's IDE does not attempt to match the habits of C programmers under the Unix environment. Common Lisp does not make use of a Makefile. The ACL environment is really unfamiliar to someone used to the C development cycle. Thus, using the ACL environment requires changes in the programmer habits. As we have pointed out in the introduction, we think that enforcing an unusual programming methodology discourages programmers.

In addition, the Emacs environment for ACL does not make use of any graphical interface. Even if command line interfaces are powerful, in some situations they are inferior to graphical interfaces. For instance, when debugging a program, it is much more convenient to set a breakpoint using a mouse click inside the source code than typing a line number to describe the position of the breakpoint.

7.5 Lisp machines and interactive environments

The programming methodology advocated in this paper is the opposite of the one in used in old Lisp environments (Sandewall, 1978). We enforce static features such as modules, impossibility to refer to unbound variables, type and arity checks. Our environment is designed for the delivery of stand alone applications, which is the dominating trend for the last twenty years. Lisp environments were designed toward permanent interactions with the programmer. It is however possible that such environments get a revival interest because they seem to propose features now needed by network programming (such as the dynamic upgrades of some portion of a program).

Conclusion

More than eight years ago, R. Gabriel stated that Lisp was lacking some important features and that its development was endangered (Gabriel, 1991). According to Gabriel the two main deficiencies were: a connection to C and the ability to produce stand alone executables. The Bee is an integrated development environment for the Scheme programming language that implements these features. Within the Bee, Scheme code and C code may be integrated harmoniously to build small stand-alone executables.

In the first section of this paper the graphical user interface was presented. The remaining sections have been devoted to the presentation of the portable implementation of two of the main environment tools: the profiler and the debugger. These implementations are driven by the very same idea: minimizing the implementation effort by relying as much as possible on existing C tools. For instance, our debugger is 6000 lines of Scheme code which is far fewer than the half million lines of C code for GDB! Our debugger may be used on any computer and operating system provided the GNU debugger GDB is ported to that machine. Designing our debugger as a shell around GDB enabled us not to implement any architecture specific features. These are embedded in GDB.

In order for Scheme to be able to use regular C tools such as GDB, the runtime execution of Scheme must comply with the C execution model. For instance, Scheme function invocations must conform to the C call protocol of the host machine. The easiest way to reach this compliance is to compile Scheme to C using what we call the "natural mapping" framework. We can think of no reason why a natural mapping could not be used for other languages.

The class of functional languages will be more widely used if we bring the development facilities of classical languages to users. In the coming months we will have to maintain our implementation work in order to deliver more libraries. For instance, a connection to a graphical toolkit or a library for networking are likely to be the next targets of our implementation effort.

Acknowledgments

Many thanks to Barrie Stott, Bahman Rafatjoo, Céline and the anonymous referees of Journal of Functional Programming for their helpful feedbacks on this work.

References

Apple Computer, Eastern Research, & Technology. (1992). *Dylan, an object-oriented dynamic language*. Apple Computer, Inc.

Attardi, G. (1994). *The Embeddable Common Lisp*. Tech. rept. unpublished. Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy.

Baker, H. (1994). *CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A <1>*. Tech. rept. TR-Nbl-94-01. .

Boehm, H.J. (1996). Simple Garbage-Collector-Safety. *Conference on programming language design and implementation*. SIGPLAN Notices.

Boehm, H.J., & Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software — practice and experience*, **18**(9), 807–820.

Cannon, L., Elliot, R., Kirchoff, L., Miller, J., Milner, J., Mitze, R., Schan, E., Whittinton, N., Spencer, H. Keppel, D., & Brader, M. 1990 (June). *Recommended C Style and Coding Standards.*

Clinger, W. 1998 (June). Proper Tail Recursion and Space Efficiency. *Conference on programming language design and implementation.*

Davis, H., Parquier, P., & Séniak, N. (1994). Sweet Harmony: the Talk/C++ Connection. *Pages 121–127 of: Conference record of the 1994 acm conference on lisp and functional programming.*

Ericsson. (1998). *Erlang.* http://www.erlang.se.

Feeley, M., Miller, J., Rozas, G., & Wilson, J. 1997 (Aug.). *Compiling Higher-Order Languages into Fully Tail-Recursive Portable C.* Tech. rept. Rapport technique 1078. Université de Montréal, Département d'informatique et r.o.

Felleisen, M., Findler, R., Flatt, M., & Krishnamurthi, S. (1998). The DrScheme Project: An Overview. SIGPLAN *notices.*

Findler, R., Flanagan, C., Flatt, M., Krishnamuryhi, S., & Felleisen, M. (1997). DrScheme: A Pedagogic Programming Environment for Scheme. *Int. symp. on programming languages, implementations, logics, and programs.*

Flanagan, C. (1997). *Effective Static Debugging via Componential Set-Based Aanlysis.* Ph.D. thesis, Rice University.

Flanagan, C., Flatt, M., Krishnamuryhi, S., Weirich, S., & Felleisen, M. (1996). Static Debugging: Browsing the Web of Program Invariants. *Conference on programming language design and implementation.* SIGPLAN Notices.

Franz Inc. (1998). *Allegro Common 5.0.* http://www.franz.com/.

Gabriel, R. (1991). *Lisp: Good News, Bad News, How to Win Big.* http://www.ai.mit.edu-/articles/good-news/good-news.html.

Harlequin. (1998). *ML Works.* http://www.harlequin.com.

Hartel, P. H., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C. H., Grieskamp, W., van Groningen, J. H. G., Hammond, K., Hausman, B., Ivory, M. Y., Lee, P., Leroy, X., Loosemore, S., Röjemo, N., Serrano, M., Talpin, J.-P., Thackray, J., Weis, P., & Wentworth, P. (1996). Pseudoknot: a Float-Intensive Benchmark for Functional Compilers. *Journal of functional programming*, **6**(4), 621–655.

ISO/IEC. 1990 (July). *9899 programming language - C.* Tech. rept. DIS 9899. ISO.

Kelsey, R., Clinger, W., & Rees, J. (1998). The Revised^5 Report on the Algorithmic Language Scheme. *Higher-order and symbolic computation*, **11**(1).

Kinnucan, P. (1998). *JDE User's Guide.* http://sunsite.auc.dk/jde/.

Leroy, X. 1998 (June). Compilation techniques for function and object-oriented languages. *Page tutorial of: Conference on programming language design and implementation.*

Lippman, S. (1996). *Inside The C++ Object Model.* Addison-Wesley.

Moon, D., & Weinreb, D. (1981). *Lisp machine manual.* Technical Report. MIT Artificial Intelligence Laboratory.

Pessaux, F., & Leroy, X. (1999). Typed-based analysis of uncaught exceptions. *Symposium on principles of programming languages.*

Rose, J. R., & Muller, H. (1992). Integrating the Scheme and C languages. *Pages 247–259 of: Conference record of the 1992 acm conference on lisp and functional programming.*

Rozas, G.J. 1992 (June). Taming the Y operator. *Pages 226–234 of: Conference record of the 1982 acm symposium on lisp and functional programming.* ACM.

Sandewall, E. (1978). Programming in an Interactive Environment: the "LISP" Experience. *Computing Surveys*, **10**(1), 35–71.

Serrano, M. 1994 (Dec.). *Vers une compilation portable* et *performante des langages fonctionnels.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France.

Serrano, M. 1997 (Sept.). Inline expansion: *when* and *how*? *Pages 143–147 of: Int. symp. on programming languages, implementations, logics, and programs.*

Serrano, M., & Feeley, M. 1996 (May). Sorage Use Analysis and its Applications. *Pages 50–61 of: 1fst int'l conf. on functional programming.*

Serrano, M., & Weis, P. 1995 (Sept.). Bigloo: a portable and optimizing compiler for strict functional languages. *Pages 366–381 of: 2nd static analysis symposium.* Lecture Notes on Computer Science.

Shalit, A. (1996). *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley.

Stallman, R.M. 1988 (Feb.). *Gdb manual.* Second edition. Free Software Foundation, Inc.

Symbolics Inc. (1981). *Symbolics Software.*

Tarditi, D., Acharya, A., & Lee, P. (1992). No assembly required: Compiling Standard ML to C. *Acm letters on programming languages and systems*, **2**(1), 161–177.

Wadler, P. (1998). Why no one uses functional languages. SIGPLAN *notices*, **33**(8), 23–27.

Zeller, A., & Lütkhaus, D. 1995 (Aug.). *DDD – A Free Graphical Front-End for UNIX Debuggers.* Tech. rept. Abteilung Softwaretechnologie, Technische Universtität Braunschweig.

Zorn, B. (1993). The Measured Cost of Conservative Garbage Collection. *Software — practice and experience*, **23**(7), 733–756.