`GOld`: a link-time optimizer for Scheme

Dominique Boucher
Locus Dialogue Inc.
460 Ste-Catherine Street west, suite 730
Montréal, Canada, H3B 1A7
`Dominique.Boucher@locusdialogue.com`

Abstract

This paper presents `GOld`, a link-time optimizer for Gambit-C that performs global optimization of multi-module Scheme programs. A technique called *abstract compilation* is used for the efficient implementation of its static analysis phase. The system removes unnecessary type checks, inlines system primitives, and does simple function inlining across module boundaries. We show that using this system, important runtime speedups can be achieved on programs free of non-standard declarations.

1 Introduction

Support for modular programming, in the broadest sense, is an essential property of any programming system intended for the development of large software systems. But modular programming usually comes with a cost: compilers are rarely capable of performing cross-module optimizations, thus producing sub-optimal executables.

In [5], we describe the architecture of a generic compilation system featuring global cross-module analysis and optimization of multimodule programs at link-time (we will give an overview of this architecture in a later section). This architecture leverages the use of static analyses developed in the abstract interpretation framework to the efficient analysis of multimodule programs.

In order to show the effectiveness of this architecture, we developed `GOld` (for **G**ambit-C **O**ptimizing **l**ink e**d**itor), a first (partial) implementation of this architecture based on the Gambit-C Scheme compiler that optimizes annotation-free Scheme programs a link time. The goal of the system was to obtain executables running at a speed comparable to the ones obtained when the source code is manually annotated for optimal performances, with a reasonable link time.

The paper is organized as follows. The next section presents the Gambit-C compiler and why we decided to use it as the basis for our system. Then we present the system architecture, describing the technique of abstract compilation. We then give an overview of the analysis and optimizations performed by the system, followed by results obtained on a variety of benchmark program. We finish with an overview of related works.

2 The Gambit-C system

Gambit-C is a high-performance Scheme compiler that produces C code. Even if it generates good code, it lacks a good static analysis, thus relying on programmer-supplied annotations to generate high performance code. For example, the annotation

```
(declare (standard-bindings))
```

tells the compiler that all calls to primitives, in the scope of the declaration, do refer to the standard Scheme primitives, thus enabling the inlining of some primitives like `not`, `char?`, etc., and the elimination of some tests for the number of arguments in calls to other primitives. The semantics of these annotations is not always easy to understand and often multiple annotations are required to obtain good performances. Also, these annotations make the code unreadable and are highly compiler-dependent. Morever, the compiler blindly follows those annotations, sometimes resulting in incorrect, hard to debug generated code.

But one of the main source of inefficiency with Gambit-C is the cost of procedure calls across the boundary of modules. Gambit-C is conformant with the Scheme standard, meaning that it must optimize all tail calls. Since each Scheme source file is compiled to a distinct C source file, a cross-module call results in a C call. But C compilers do not optimize tail calls in general. To solve this problem, Gambit-C uses a *trampoline* technique. Each Scheme file (or module) is compiled in a unique C function which serves as the entry point for the module. Each function in this module is accessed using a series of `goto`s. So a cross-module calls consists in five distinct operations:

1. a `goto` statement is executed to go to the end of the calling C function;
2. the C function returns control to the Gambit-C kernel;
3. the kernel determines to which module control should be passed;
4. the C function of the target module is entered;
5. a `switch` statement is executed to enter the called function.

When the called function returns, control is given back to the calling function using the same sequence of operations. We thus see that cross-module calls are very expensive. Morever, in the absence of adequate annotations, all calls to primitive functions will use this trampoline mechanism. (Note that intra-module calls require fewer operations, at most two operations in the worst case.)

These are the two problems our system tries to address: it tries to automatically insert the annotations and to reduce the cost of cross-module calls. We will now see how the `GOld` system is structured.

3 `GOld`'s architecture

The architecture of our system is based on the concept of abstract compilation, which we will now describe.

3.1 Abstract compilation

Abstract compilation [6, 17, 20] is an efficient implementation technique for static analysis which is based on abstract interpretation [9] and partial evaluation techniques [8].

Static analyses can be computed by means of an abstract interpreter, i.e. instead of executing the program with real values, the program is executed with *abstract* values. These abstract values represent some properties of the real values (such as the type, the range or other approximation of the real values). Primitive operations, like function application, are also interpreted differently, according to the analysis that must be computed. In short, the program is executed using a different (non-standard) semantics.

This approach is often implemented by an actual interpreter that repeatedly traverses the parse tree. But interpretation is costly because it adds a layer of abstraction to the analysis process. Performance can be improved by compiling the program with respect to this non-standard semantics. The resulting *analysis program* computes, when run, the analysis of the source program, thus eliminating the overhead of interpretation.

It could be possible to derive an abstract compiler using partial evaluation techniques, but writing the compiler by hand can enable a number of important optimizations.Examples of such optimizations are described in [5], where the author shows that when machine code is generated for the analysis programs, important speedups can be achieved (more than 10 in all cases, and between 15 and 30 in most cases).

3.2 Ideal architecture

As shown in [5], the cost of abstract compilation can sometimes outweigh the cost of the static analysis itself, resulting in a performance degradation. This is mainly true when some time is taken to optimize the analysis program.

In order to minimize the impact of abstract compilation, we devised a separate compilation system that reuses the analysis program of each module during the link phase. Of course, we make the assumption that, most of the time, only few modules are modified between each recompilation of the whole program. So the cost of abstract compilation of a given module can be amortized over its multiple executions.

Our system, `GOld`, is based on the architecture shown in figure 1. In this architecture, all modules are first statically analyzed (`LA`), locally optimized (`LO`), and an intermediate representation is generated (`IRG`). This compilation step results in two files, the first containing the intermediate representation (`.ir` file) and the second containing the module's analysis program as well as the result of the local analysis.

The second phase combines all the analysis programs, *links* them together, and the resulting program is executed, thus computing the global static analysis (`GA`). The result of this analysis is then utilized by the global optimizer (`GO`), which reads all the `.ir` files and generates the code for the final executable.

This architecture is interesting for a number of reasons. First, it allows static analyses developed for the local phase to be reused in the context of global analysis. So it is not necessary to develop new analyses, but instead focus on how to factor the analysis of a module into its local part and its global part. We define the local part as being the part of module for which the analysis is not affected by the interaction with the other modules).

Secondly, libraries can be developed such that they will be optimized based on the client application. Only the `.ir` and `.ap` files could be published. This means that these libraries could be very high-level without incuring performance penalties to the programs using tem. Also, a good encoding scheme could be used to protect intellectual property.

The effectiveness of such an architecture depends heavily on the choice of the intermediate representation. This representation will determine which global optimizations are possible and which are not, and how fast the final generated code will be. However, we will not discuss in detail the issues raised by the design of a good intermediate representation. These issues are beyond the scope of this paper.

3.3 `GOld` architecture

`GOld` diverges from this architecture in a number of respects. The main reason is that we wanted to be less intrusive as possible in the Gambit-C system. Gambit-C was not designed in the first place to be modified as we wished.

The first (local) phase analyzes each module separately and stores the corresponding analysis program and the result of local analysis on disk. No intermediate representation is stored on disk. This means that the link phase has to parse each module again, do macro-expansion, and apply some other local optimizations. This is quite inefficient.

Second, the link phase generates C code instead of the final executable. There is one C file for each Scheme module, plus one link file, containing the C `main` function and other definitions. An extra compilation step is required here to generate the final executable. Nevertheless, our goal was to demonstrate that link-time analysis and optimisation of Scheme programs could be done efficiently, so we concentrated mainly on this aspect of the system.

4 Analysis and optimizations

This section outlines the analysis and optimizations performed by `GOld`. For a more in-depth, formal presentation see [5].

4.1 Static analysis

The static analysis implemented in `GOld` is mainly based on Olin Shivers' 0CFA [26, 27], which is a monovariant control flow analysis for high-order languages. Like the 0CFA, `GOld` computes the analysis by means of a fixpoint iteration algorithm. Both the local and global analyses are performed by running compiled analysis programs. These programs are represented at runtime by trees of closures, as in [6, 13].

However, our analysis differs from Shivers' one in a number of respects. First, source programs are not CPS-converted. That's because Gambit-C optimizes and compiles source programs in direct form and we did not want to add the cost of an extra program transformation (the CPS-conversion) to the link phase.

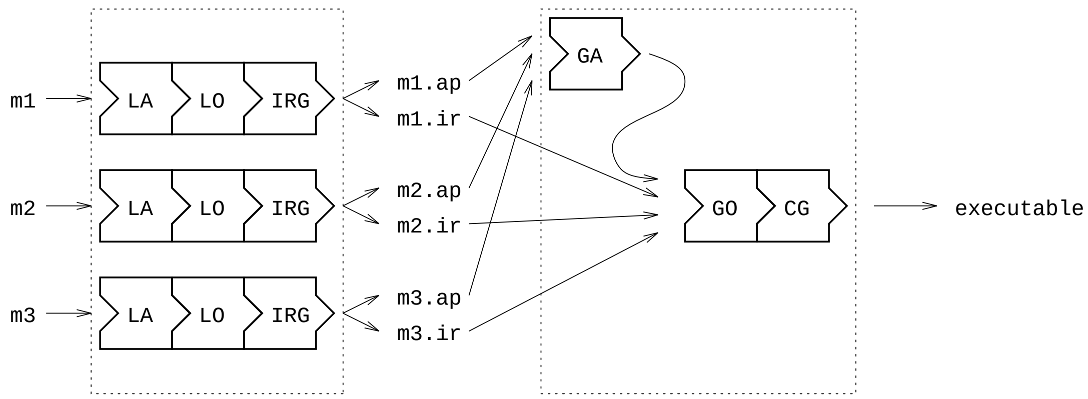

Figure 1: Compiler architecture.

Flat structure. An abstract function call, in the 0CFA, forces the abstract evaluation of the body of all abstract closures that result from the evaluation of the expression in operator position. Combined with a timestamp technique, this approach ensures that the interpretation algorithm terminates. It is also able to detect unused procedures in the program.

In order to avoid the management of timestamps and simplify the code generation for the analysis program, `GOld`'s analysis uses a different approach. The idea is to traverse the code of all expressions in the program once in each iteration of the fixpoint computation. In this context, an abstract function call only propagates the information associated with the actual parameters to the formal parameters of all abstract closures that can be bound to the operator.

This technique can lead to a less precise analysis in certain programs. Morever, it becomes much more difficult to detect unused procedures. Consider the following code fragment:

```
(define (foo x) (+ x 1))
(define (bar) (foo 'a))
(display (foo 2))
```

This program never calls `bar` at runtime. Unfortunately, our analysis will add the symbol `'a` to the approximation for `x` since `foo` is called from within `bar`.

On the other hand, this technique greatly simplifies the compilation of the analysis. Since the code of each procedure is only traversed once in each iteration, it is not necessary to include a code pointer in the representation of the abstract closure. An abstract function call is compiled to a single instruction that does the propagation of information from the actual parameters to the formal parameters. This implies that every program can be abstractly compiled to a sequence of abstract instructions that can be executed sequentially, except for conditional expressions (`if`, `and`, and `or`) which can introduce forward conditional branches.

Abstract values. The abstract values handled by `GOld` are comparable to those used in the Bigloo compiler [23, 24]. These values have been chosen to match the kind of values leading to the best optimizations by Gambit-C. These values can be classified into three broad categories:

Atomic values These values abstract either distinct constants (like `#t` and `#f`) or (finite or denumerable infinite) sets of values of the same type (like characters, symbols, strings, ports).

Numerical values These abstract values represent intervals of numerical values. Among them, we have one for real numbers, one for complex values, and one for *flonums* (real numbers directly representable by a machine floating-point value). For integer values, we have a special value abstracting the whole set of integers. We also have a finite set of intervals of integer values representable by a 32 bit machine word.

Structured values These values abstract heap-allocated data structures (pairs, vectors, and closures).

Mutation of primitive procedures. In Scheme, variables to which primitive procedures are bound can be redefined. In a multi-module setting, the compiler cannot blindly inline and optimize all calls to primitive procedures. To solve this problem, `GOld` proceeds iteratively. One only has to consider the analysis program as a function taking as input the set of mutated primitive procedures. Let $\mathcal{A}_i : \mathcal{P}(Prim) \rightarrow \mathcal{P}(Prim)$ denote such a function.

The first step consists of compiling all the analysis programs by assuming that no primitive is mutated. Then the global analysis in launched, while computing $P_1 = \mathcal{A}_1(\emptyset)$. If $P_1 \neq \emptyset = P_0$, then all modules containing calls to primitives in P_1 are *invalidated*. They must be reanalyzed locally and their analysis program must be regenerated. Then the global analysis is run again, resulting in a set $P_2 = P_1 \cup \mathcal{A}_2(P_1)$. Since the set of primitive procedures is finite, there will be an $n \geq 1$ such that $P_n = P_{n-1}$.

For a typical program that does not mutate any primitive, this method has the advantage of requiring only one iteration, hence being transparent to the programmer. If the initial assumption was that all the primitives are mutated, at least two iterations would have been necessary.

Polyvariant treatment of Scheme primitives. Although our analysis is monovariant, all Scheme primitives (when they are not redefined in the program) are treated polyvariantly to get more precise approximations.

All calls to basic allocating primitives (`cons`, `vector`, `make-vector`) generate distinct abstract structured values, as well as all `lambda`-expressions. But a few more primitives add new structured values. These are `append`, `reverse`, `map`, `vector->list`, and `list->vector`.

Note that all other Scheme primitives (`+`, `car`, `integer->char`, etc.) are also treated polyvariantly. The (abstract) code for these primitives (that is executed when the analysis program is run) is taken from a library of abstract primitives linked with the analysis program. This contrasts with other systems (like Stalin [28]) that append the standard Scheme library to the analyzed program.

Type filtering and interval narrowing. In order to get even more precise approximations, we implemented two more techniques in our analysis: type filtering and interval narrowing.

Type filtering uses the type tests in conditional expressions (`if` expressions) to constrain the approximation of the tested variable in both the consequent and the alternative arms of the `if` expressions. For example, in the following code

```
(if (pair? x)
    (foo x)
    (bar x))
```

the approximation for `x` will be constrained to only abstract values denoting pairs in the call `(foo x)`, and to abstract values **not** denoting pairs in the call `(bar x)`.

This idea is not new. It is described in [27] and is now widely used [3, 18]. Note that due to its flat structure, our analysis does not suffer from the *reflow semantics* problem described in Chapter 9 of Shivers' work.

Interval narrowing is a similar idea. We use inequality tests in `if` expressions to constrain the approximations. For example, in

```
(if (< i n)
    (foo i)
    ...)
```

if `i` and `n` are approximated by integer intervals ($[i, j]$ and $[k, l]$ resp.), then in the call `(foo i)`, the actual value for `i` is constrained to the interval $[i, \min(j, l-1)]$ while `n` is contrained to the interval $[\max(i+1, k), l]$. Using this technique, `GOld` is able to determine that `<` and `+` are both called on *fixnums* only in the following code:

```
(let ((len (vector-length v)))
  (let loop ((i 0))
    (if (< i len)
        (begin
          (foo (vector-ref v i))
          (loop (+ i 1)))
        #f)))
```

4.2 Intermediate representation

When stored on disk, the analysis programs are represented using bytecodes. These bytecodes encode the various low-level operations the analysis perform. There is 7 such bytecodes. These include the join ($\sqcup$) of two approximations, the abstract application of a variable on a list of variables, the application of primitive procedures (in fact their abstract version), etc. The analysis program can also include some labels, similar to that of assembly languages. These labels are required for conditional expressions. So we can think of these bytecodes as instructions to a very special virtual machine[1].

The file also contains a symbol table for the module. This table lists all the defined top-level variable in the module, the undefined symbols, and the redefined primitive variables.

[1] This representation will help the generation of machine code for the execution of the global analysis in a future version of the system.

At link time, the analysis programs are read and converted from their bytecode representation to trees of closures for execution. Since only one analysis program is read at a time, some sort of relocation must take place, much like any traditional linker. For example, if a module A refers to a variable `V` defined in some other module, the bytecodes for A will contain references to this external variable. The linker will then have to find out in which module the approximation to `V` lies, and then replace each reference to `V` in A by its corresponding approximation.

To do this linking, `GOld` uses a technique similar to the one described in [22]. It uses currification to represent the various stages needed to perform the relocation and generation of final code. In our setting, each closure generation procedure generates a function that accepts a global symbol table as its unique parameter. This function, when applied, resolves the external references by looking in the global symbol table and then returns a *thunk* (a function with no parameter) that represents the relocated code. The closure generation procedure for an `if` expression looks roughly like the following:

```
(define (generate-if tst consq alt)
  (lambda (symtable)     ; relocation
    (let ((reloc-test  (tst symtable))
          (reloc-consq (consq symtable))
          (reloc-alt   (alt symtable)))
      (lambda ()         ; final relocated code
        (if (reloc-tst) (reloc-consq) (reloc-alt))))))
```

4.3 Optimizations

`GOld` performs three different optimizations: implicit annotation of source code, inlining of primitive procedures, and cross-module inlining of user code. After these optimizations are performed, the usual Gambit-C optimizations are applied (β-reduction, λ-lifting [19])).

Implicit annotation. When appropriately annotated, Scheme programs can be compiled to very efficient code by Gambit-C. Our first optimization consists in annotating the abstract syntax tree (AST) of the source program, based on the global analysis results.

`GOld` tries to add the following annotations to the AST:

`(block)` This annotation tells the compiler that the variables in its scope are not mutated outside of the module. Gambit-C is then able to generate more efficient code for calls to such annotated procedures. A toplevel procedure definition `(define (F ...) ...)` is annotated if the analysis proves that `F` can only be bound to this procedure at runtime.

`(standard-bindings)` This annotation instructs the compiler that the predefined variables are bound to the corresponding primitive procedures. Gambit-C can then generate more efficient calls and, when combined with the `(not safe)` annotation, inline those primitives.

`(not safe)` This annotation instructs the compiler not to generate implicit type tests and bounds checks. In calls to primitive procedures, `GOld` adds this annotation if the number of arguments is correct and they have the right type. For small primitive procedures, this can completely eliminate the need for a cross-module call. This annotation is also added to all other calls such that the expression in operator position has been proven to be a closure.

(fixnum) and (flonum) When the arguments to some primitive arithmetic procedures can be proven to be only *fixnums* or *flonums*, these annotations enable the compiler to generate specialized code for these procedures.

It is the combination of many of these annotations that allows Gambit-C to generate the most efficient code possible.

Inlining of Scheme primitives. Sometimes, the result of the analysis is not precise enough to optimize all primitive calls using only annotations. In principle, this would result in more cross-module calls to the Scheme library. To reduce this cost, `GOld` inlines the primitive procedure, with all its type tests. For example, if `GOld` is unable to infer that the `arg1` is always bound to a pair in a call `(car arg1)`, it will replace this call with the following equivalent code:

```
(let ((y arg1))
  (if (let ()
        (declare (standard-bindings) (not safe))
        (pair? y))
      (let ()
        (declare (standard-bindings) (not safe))
        (car y))
      (car y)))
```

This code checks that `arg1` is a pair. If so, it calls the unsafe version of `car` (which will be inlined by the Gambit-C backend due to the annotations). Otherwise, it will call the safe version that will throw an error. In this case, the cross-module call does not have a great impact on performances since the program will fail.

For some primitives, more than one test are required before calling the unsafe version. `GOld` tries to remove most of these tests when possible, based on the result of the analysis. For `vector-ref` and `vector-set!`, `GOld` is sometimes able to keep only the bounds checking tests. In total, 50 primitives can be inlined this way.

Cross-module inlining. Scheme does not provide any mechanism for defining new datatypes (like C `structs` or Pascal `records`). Vectors and lists are used for this purpose and the programmer usually defines procedures for creating such a datatype, accessing its fields, modifying them, etc. This can result in many small procedures and many cross-module calls (typically, these procedures are defined in a single module and called in several modules).

In its attempt to reduce further the cost of cross-module calls, `GOld` tries to inline those small procedures across the boundary of modules using a very simple, but effective approach. During the separate compilation phase, `GOld` puts in a global database all the toplevel procedures that are candidates to cross-module inlining. During the global optimization phase, all calls to a candidate procedure are inlined.

To be a *candidate* for cross-module inlining, a toplevel procedure must satisfy the following two conditions: *(1)* all internal calls are calls to primitive procedures and *(2)* the size of the AST of the function body is less than or equal to $k \times n$ where n is the number of formal parameters and k is a fixed threshold.

This scheme is somewhat naïve, but it gives interesting results in practice. The search for a better inlining scheme was beyond the scope of our work and left for further improvements of the system. We will show the effect of different values of k in the next section.

5 Experimental results

We now present experimental results obtained using `GOld`. First, the system has been evaluated using only unimodule programs. Then we present results obtained on two larger programs.

In all cases, results have been obtained on a 400MHz Pentium II processor with 512Mb of RAM running Linux 2.2.9. Unless explicitly stated, all times are in seconds.

5.1 Unimodule programs

In order to show the effectiveness of `GOld`'s static analysis and optimizations, a number of unimodule programs have been compiled. The programs are described in figure 2. These are usual benchmarking programs for Scheme. Their size vary from a few tens of lines to a few thousands. The first column gives the program's name. The second column shows the number of lines of code and, in two cases, the number of lines of code required to represent the data for a particular instance of the problem to solve. The last column gives a brief description of the program.

Compilation times.

Figure 3 presents four measurements regarding the analysis and optimisation of these programs. The **iter** column gives the number of iterations needed to reach the fixed point during the local analysis. This number is usually small (between 4 and 15), except for `nucleic`, where it is 46. This is due to the program structure and not to the presence of constant structures representing the database of nucleotids.

The **analysis** column gives the execution time for the local analysis. This number depends on the number of iterations, the size and structure of the source program. Except for `nucleic` and `slatex`, this number is less than 1 second and often negligible. Note that this time does not include the initialization of the analysis, only its execution. The total time is given in parentheses and it includes the time to save the analysis program to disk[2].

The `link` column gives the total time required for the global analysis phase, including the analysis programs loading, generation and execution, the global optimizations, and the generation of C code. Since all the programs contain only one module, just one iteration is needed to reach the fixed point. This means that the C code generation outweighs the other steps.

Finally, the `gcc` column gives the compilation time of the resulting C programs by `gcc` v2.7.2.3, using the command-line options `-O3 -fomit-frame-pointer`. We see that in most cases, the global analysis time is less than the C compilation time, except for `boyer` and `nucleic`. Both programs contain a large amount of constant data, larger than the program itself and Gambit-C is not efficient at compiling these constants.

Execution times.

Figure 4 gives the execution time of all benchmarks. Three different times have been gathered. Column **annot** gives the execution time when the program is manually annotated in

[2]No special care has been taken to reduce the saving cost, Gambit-C 2.8a being a system with poor I/O performances. The current version is much more efficient in this respect.

program	lines	description
tak	12	The Takeuchi function.
fib	12	The Fibonacci function.
fibfp	13	A floating-point version of the Fibonacci function.
takl	27	The Takeuchi function using lists as counters.
cpstak	27	A CPS version of the Takeuchi function.
primes	28	Computation of prime numbers less than 1000.
nqueens	32	The 8-queens problem.
pnpoly	44	Program that tests if a point is contained in a 2D polygon.
mbrot	48	The Mandelbrot set computation.
dderiv	82	A symbolic derivation program.
puzzle	141	A program solving the Forest Baskett puzzle.
simplex	184	The simplex algorithm for linear programming.
browse	187	Program to create and browse through an AI-like database of units.
boyer	203 (+350)	A theorem-proving program.
earley	648	A parser generator based on Earley's algorithm and the execution of a parser.
nucleic	1000 (+2450)	A program for the 3D structure determination of a nucleic acid.
slatex	2337	A Scheme to LaTeX converter.

Figure 2: Unimodule program description.

program	iter	analysis	link	gcc
fibfp	4	< 0.01 (0.07)	0.17	0.35 (0.51)
fib	4	< 0.01 (0.07)	0.20	0.44 (0.55)
tak	5	< 0.01 (0.07)	0.31	0.55 (0.54)
takl	5	< 0.01 (0.09)	0.39	0.78 (0.54)
primes	5	< 0.01 (0.10)	0.25	0.57 (0.53)
cpstak	5	< 0.01 (0.10)	0.52	0.71 (0.55)
nqueens	6	< 0.01 (0.12)	0.45	0.65 (0.52)
mbrot	6	< 0.01 (0.14)	0.50	0.92 (0.52)
dderiv	8	< 0.01 (0.17)	0.62	1.15 (0.55)
pnpoly	4	< 0.01 (0.21)	0.82	1.11 (0.56)
puzzle	8	0.04 (0.57)	1.75	2.22 (0.58)
simplex	8	0.10 (1.30)	4.44	9.42 (0.57)
browse	11	0.12 (0.69)	2.09	3.14 (0.54)
boyer	13	0.16 (0.65)	3.67	2.86 (0.71)
earley	13	0.40 (2.39)	6.71	14.30 (0.54)
slatex	13	1.40 (5.75)	20.96	26.98 (1.18)
nucleic	46	3.89 (6.87)	33.19	29.62 (0.91)

Figure 3: Local analysis of unimodule programs.

program	annot	gold		no annot	
dderiv	2.11	**(0.95)**	2.00	(4.38)	8.77
fibfp	15.46	**(1.02)**	15.70	(6.24)	98.03
slatex	6.23	**(1.04)**	6.46	(1.76)	11.38
cpstak	13.14	**(1.05)**	13.84	(8.85)	122.52
primes	7.35	**(1.09)**	8.02	(14.21)	113.98
mbrot	13.35	**(1.11)**	14.80	(8.61)	127.47
browse	5.86	**(1.12)**	6.55	(17.07)	111.86
boyer	4.21	**(1.13)**	4.77	(23.53)	112.21
nqueens	4.27	**(1.19)**	5.06	(21.58)	109.18
takl	3.62	**(1.31)**	4.75	(24.83)	117.83
fib	3.67	**(1.36)**	5.01	(23.28)	116.56
tak	3.63	**(1.40)**	5.09	(23.69)	120.51
earley	7.89	**(1.90)**	14.95	(3.27)	48.95
pnpoly	1.61	**(2.03)**	3.27	(22.12)	72.41
puzzle	3.54	**(2.82)**	9.98	(21.45)	214.09
simplex	4.29	**(3.26)**	13.99	(10.18)	142.37
nucleic	9.95	**(4.84)**	48.20	(2.79)	134.29

Figure 4: Execution time of unimodule programs.

order to get optimal results. Column **gold** gives the execution time of the same program when compiled with our system. The number in parentheses gives the ratio between column **gold** and column **annot**. Finally, the last column (**no annot**) gives the execution time for the program when compiled with Gambit-C without any declaration. The number in parentheses gives the speedup obtained using `GOld`[3].

We can first see that almost half of the programs compiled with `GOld` were less that 15% slower than the one manually annotated and that only 5 programs were more than 40% slower. Morever, 10 programs were between 10 and 25 times faster compared to the same program compiled without any annotation. We will now analyze in more details the differences between times shown in column **annot** and **gold** for some of the programs.

`boyer`, `browse`, and `dderiv` are three symbolic applications handling almost exclusively symbols and lists. A large portion of all type tests have been eliminated, accounting for the small differences between the optimized programs and the one manually annotated. In the case of `dderiv`, the negative difference could be explained by a different ordering of the basic blocs generated by Gambit-C, resulting in better cache performances.

`tak`, `takl`, and `cpstak` all compute the Takeuchi function. In the case of `tak`, the static analysis is only able to determine that values passed in parameters are integers, and not *fixnums*. Subtraction on integer values is more costly

[3]These speedups are shown only to give a feel of how close to the (presumably optimal) annotated programs the optimized ones can be. These numbers are by themselves not really significant.

than the subtraction on *fixnums*[4]. For `takl`, no call to `cdr` can be done without a type test, accounting for the 31% difference. The difference for `cpstack` is only 5% but we expected it to be the same as for `tak`. The cost of the subtraction is amortized by the large number of closures created at runtime and the number of garbage collects (280 for `cpstack`, 0 for `tak`).

`fib` and `fibfp` both compute the Fibonacci function. The only difference is the result type. `fib` computes an integer results, while `fibfp` returns an inexact real number. In the first case, `GOld` cannot infer that the result is a *fixnum*. In the second case it can prove that the result is a *flonum*. This explains why `fib` is 35% times slower than its annotated counterpart and `fibfp` is only 2% slower.

When optimized with `GOld`, `slatex` is only 4% slower than the annotated version, a good result for such a large program. But the 76% difference between the optimized version and the non-optimized, annotation-free version indicates that the cost of I/O outweighs all other costs.

The `mbrot` program shows that programming style can have a significant impact on performance. The inner loop of the Mandelbrot computation algorithm has the following stopping criteria:

```
(if (= c max-count)
    ...)
```

But `GOld` is unable to do the interval narrowing with equality tests (the analysis is mostly interested in ranges of values, not constant values). If the the following, equivalent code was used:

```
(if (>= c max-count)
    ...)
```

then many more type tests would have been eliminated and some arithmetic procedure calls would have been compiled much more efficiently.

`pnpoly` and `puzzle` are two programs performing lots of vector accesses and mutations. In most cases, `GOld` cannot eliminate the bounds checks and the type tests for the indices. That's why a factor of between 2 or 3 is obtained for them[5].

Finally, `simplex` and `nucleic` are the two programs for which the difference was the most important (3.26 for `simplex` and 4.84 for `nucleic`). In both cases, a lot of time is spent by vector accesses and mutations, as well as floating point operations. For `simplex`, most arithmetic operations have been optimized for *flonums*, which is not the case for `nucleic`. The latter contains a few calls to `sqrt`, for which the result can only be approximated by a complex number, not a *flonum* (`GOld` does not handle ranges of *flonum* values as for *fixnums*). Also, the database of nucleotids is represented using heterogeneous vectors (containing vectors, symbols and *flonums*) implementing a form of simple inheritance hierarchy. Accesses in these data structures result in too coarse approximations, so `GOld` can only optimize a few calls to arithmetic operations. This leads to a lot of boxing/unboxing of values, further degrading the performance due to many more calls to the garbage collector at runtime.

[4]The summation and subtraction of *fixnums* can be done in Gambit-C using only one machine instruction (apart from the overflow/underflow tests), thanks to the type encoding scheme that sets the two least significant digits to 0. The `(not safe)` and `(fixnum)` declarations must be present, of course. The type tests and tests for overflow add an extra cost to these operations. Short tests have shown that, on a Pentium processor, adding the type test for only one argument, the summation becomes 5 times slower. The summation and substraction of *flonums* are 2 times slower when type tests are added. Morever, since each intermediate floating-point value is allocated in the heap, boxing/unboxing costs are added to both operations and more garbage collects are required.

[5]Experimental results show that when all type tests and bounds checks are inlined, vector accesses are 3.3 times slower than their corresponding, fully annotated version. This factor rises to 10 for the same annotation-free program.

5.2 Multimodule programs

We now analyze `GOld`'s performance on two multimodule programs:

etos: an Erlang [2] to Scheme compiler written by Marc Feeley and Martin Larose, version 1.4;

Gambit-C: the Gambit-C compiler itself, version 2.8a [12].

etos consists of 12 modules. These modules are described at figure 5. Column **lines** gives the number of lines of each module. As we can see, a few modules are as small as a few tens of lines, while the four larger ones have more than 1300 lines. Among them, `erlang.l.scm` and `erlang.y.scm` were automatically produced by a lexical analyzer generator and a parser generator. In the case of Gambit-C, there are 15 modules (Figure 6), ranging from 9 to more than 3000 lines of code. In contrast to **etos**, none of its modules have been automatically generated.

Compilation times

Figures 5 and 6 gives several metrics regarding the local analysis of each module in the two systems. Column **iter** gives the number of iterations needed to reach the fixed point. Column **local** gives the total local analysis time, including the initialization, the closure generation, the analysis program generation and the saving of the local analysis results to disk. The local analysis time itself is given in parentheses (a value of 0.00 indicates a negligible time).

For the larger modules, this time is more than two seconds. Except for `erlang.y.scm` in **etos** and `_gvm.scm` in Gambit-C, the local analysis time is much less than one second. This means that most of the time is spent doing other things like reading the source code, applying some semantics-preserving transformations (in Gambit-C), generating the analysis program, saving this program to disk, etc. Nevertheless, these times can be further reduced by optimizing those steps.

Figure 7 gives some measurements regarding the generation of the two executables using `GOld`. In the first half of the table, we give number of iterations needed to reach the fixed point, the loading time of the analysis programs and the local results, the execution time of the global analysis program, the total link time (including the optimization time of each module and the C code generation time), and the time needed by `gcc` to build the executable. The second part of the table gives similar results for the whole system being compiled as a single module (all the modules in one source file). The measures are: the analysis time and the number of iterations, the compilation time by Gambit-C and the compilation time by `gcc`.

etos. We see that the time spent in the optimization and code generation (C and object code) phases is much larger than the global analysis time. This means that, in this case, global analysis is cheap, considering that the global analysis program is not compiled to object code but to closures. Also, the Erlang scanner (the `erlang.y.scm` module) contains a vector in which 167 elements are closures. These are

module	lines	size	iter	local
err.scm	31	635	2	0.04 (0.00)
bv.scm	68	2278	5	0.16 (0.01)
util.scm	83	2424	3	0.18 (0.00)
lr-dvr.scm	85	3667	5	0.12 (0.00)
obj.scm	90	4396	4	0.08 (0.00)
globals.scm	93	1716	2	0.05 (0.00)
fv.scm	200	6263	6	0.32 (0.05)
match.scm	426	13609	5	0.55 (0.05)
comp.scm	1354	44156	10	2.32 (0.54)
erlang.y.scm	1392	61582	3	2.38 (0.07)
ast.scm	1418	46674	6	2.19 (0.25)
erlang.l.scm	1826	63609	17	5.25 (1.81)
Total	7066	251009	—	13.64 (2.78)

Figure 5: The **etos** modules.

module	lines	size	iter	local
gsc.scm	9	271	2	0.02 (0.00)
_parms.scm	199	9724	2	0.24 (0.00)
_back.scm	219	8605	2	0.16 (0.00)
_t-c-3.scm	274	7484	4	0.35 (0.01)
_env.scm	359	10663	7	0.48 (0.04)
_host.scm	522	13971	4	0.46 (0.02)
_prims.scm	528	35742	1	0.15 (0.00)
_utils.scm	545	15060	8	1.06 (0.12)
_source.scm	1234	43971	8	1.78 (0.19)
_ptree2.scm	1709	58867	7	2.57 (0.28)
_t-c-1.scm	1709	53758	11	3.31 (0.69)
_gvm.scm	1841	65290	14	6.30 (1.26)
_ptree1.scm	2301	81829	10	4.68 (0.93)
_front.scm	2668	94875	7	4.57 (0.46)
_t-c-2.scm	3059	108887	6	3.64 (0.36)
Total	17176	608997	—	29.77 (4.32)

Figure 6: The Gambit-C modules.

semantic actions associated with reductions in an LALR(1) analysis table (see [1]). If this scanner had been written manually, the analysis time for this module would have been reduced considerably.

Gambit-C. In the case of Gambit-C, global analysis is much more costly, while not being prohibitive. It takes about 12 minutes. This can be explained by the fact that Gambit-C is written in a very functional style, with lots of calls to allocation primitives like `map`, `reverse`, and `append`. This makes the approximations much larger (630 elements in the average for all sets containing more that 2 elements). More efficient set manipulation routines as well as compilation of the analysis program to native code would greatly improve these results.

We note that in both cases, the global analysis time is almost twice as fast as the local analysis when all the modules are put in one source file. Morever, even when the local analysis time of each separately compiled module is taken into account, global analysis is still faster (41.1 seconds against 50.2 for `etos` and 729.7 against 1350.1 for

Original program	**etos**	**Gambit-C**
Number of iterations	16	15
Intermediate file loading	5.5	14.9
Global analysis (sec.)	27.5	699.9
Link (sec.)	111.8	862.0
`gcc` (sec.)	68.7	141.7
In one module	**etos**	**Gambit-C**
Analysis	50.2	1350.1
Number of iterations	20	18
Gambit-C (sec.)	95.8	392.7
`gcc` (sec.)	147.2	315.23

Figure 7: Executable generation statistics.

program	annot	gold		no annot	
etos	7.43	(**1.28**)	9.57	(2.23)	21.33
Gambit-C	8.91	(**1.29**)	11.51	(3.61)	41.53

Figure 8: Execution times.

Gambit-C). There are several reasons explaining this fact. First, more iterations are required when all modules are concatenated together. Since the approximations are larger in the last iterations, they (the last iterations) cost more. Also, part of the code is not re-analyzed during the global analysis due to the analysis factorization. Finally, the analysis complexity (which is cubic) must be taken into account. We also note that the compilation by **Gambit-C** and by `gcc` are longer (more than a factor of 2 for `gcc`). This is due to the fact that most algorithms used in compilation have a complexity more than linear.

Execution times

Figure 8 shows the execution time of both programs compiled in three different ways. The **etos** compiler is used to compile an Erlang program 165 lines long 50 times, and **Gambit-C** to compile its larger module, `_t-c-2.scm`. In the first column, all the modules were manually prefixed with the following declarations:

```
(declare
  (block)
  (not safe)
  (standard-bindings)
  (fixnum))
```

except for a few modules requiring generic arithmetic. In these cases, the `(fixnum)` declaration was replaced with `(generic)`. The second column gives the execution time when the programs are compiled using `GOld` and the number in parentheses gives the ratio between this time and the one obtained when the program is manually annotated. Finally, the last column gives the execution time when the programs are compiled without any declaration. Numbers in parentheses show the ratio between this time and the one obtained with the programs compiled using `GOld`.

We note that both programs compiled using `GOld` are less than 30% slower than the manually annotated ones and they are more than twice as fast as the one compiled without any declarations. There are several reasons for that:

k	runtime	executable (Ko)	gcc
0	10.31 (1.00)	772.5 (1.00)	55.8 (1.00)
2	10.00 (0.97)	789.5 (1.02)	57.5 (1.03)
4	10.03 (0.97)	845.9 (1.10)	65.0 (1.16)
6	9.57 (0.93)	879.6 (1.14)	68.7 (1.23)
8	9.79 (0.95)	1007.0 (1.30)	122.2 (2.18)
10	9.87 (0.96)	1005.5 (1.30)	119.0 (2.13)
20	9.82 (0.95)	1005.5 (1.30)	120.0 (2.15)
50	9.88 (0.95)	1005.5 (1.30)	120.6 (2.06)

Figure 9: Impact of different values of k for **etos**.

k	runtime	executable (Ko)	gcc
0	15.74 (1.00)	1785.7 (1.00)	104.5 (1.00)
2	14.98 (0.95)	1780.3 (1.00)	104.4 (1.00)
4	13.15 (0.84)	1935.4 (1.08)	126.1 (1.21)
6	12.74 (0.81)	1941.8 (1.09)	126.7 (1.21)
8	12.82 (0.81)	1948.4 (1.09)	128.2 (1.23)
10	11.66 (0.74)	1969.3 (1.10)	130.0 (1.24)
20	11.51 (0.73)	2077.6 (1.16)	141.7 (1.36)
50	11.63 (0.74)	2081.5 (1.17)	142.2 (1.36)

Figure 10: Impact of different values of k for **Gambit-C**.

1. The AST nodes processed by both programs are implemented using plain, heterogeneous vectors. `GOld` is unable to use the predicates used for testing the type of a node to narrow down its approximations and optimize further the accesses to these data structures. Vector accesses thus remain costly.

2. Both programs are written in a functional style, making great use of primitives functions like `map`, `reverse`, `assq`, etc. These primitives all require cross-module calls because they are implemented in the Scheme library. All that `GOld` is able to save is reference to the global variable and the type test that ensures that the variable is bound to a procedure.

3. **etos** and **Gambit-C** are symbolic applications, being both compilers. They do not make many calls to arithmetic procedures. When left unoptimized, arithmetic procedures are more costly compared to type tests in data structure accesses.

4. It seems that Gambit-C is allowed to make better optimizations when the `(block) (not safe)` optimisations are present. It is able to generate more efficient code for intramodule calls and it can also inline more functions.

Impact of cross-module inlining

The cross-module inlining optimization has a non-negligible impact on the performance of **etos** and **Gambit-C**. Figures 9 and 10 present three measures for different values of k, the cross-module inlining threshold. The three columns give, from left to right, the execution time of the program, the size of the executable (in kilobytes), and the time taken by `gcc` to generate the executable. The number in parentheses give the ratio between the number on its left and the same measure for $k = 0$ (when $k = 0$, inlining is disabled).

These numbers show that the execution times can be significantly improved by cross-module inlining. **etos** can be speeded up by 7% whe $k = 6$ and `Gambit-C` by 27% when $k = 20$. For these thresholds, the executables are larger by 14% and 16% resp. and the compilation times are increased by 23% and 36% resp.

Surprisingly, the compilation time for **etos** is more than a factor of 2 slower when $k >= 8$. The module `erlang.y.scm` is mainly responsible for that. Its corresponding C file is 2.5 times larger (673Kb when $k = 0$ and 1595Kb when $k = 8$) and takes 4 times longer to compile by `gcc` (13 seconds for $k = 0$ compared to 65 seconds when $k = 8$). This is because a somewhat large procedure is inlined 167 times.

Of course, the impact of inlining diminishes as k increases. When the inlined procedure is bigger, the cost of invoking it (even through a cross-module call) is amortized.

5.3 Discussion

These results clearly show that `GOld` can produce programs with performances very close to those obtained when the source programs are fully annotated.

But it is also clear that the global analysis phase could be made faster by generating machine code instead of closures for representanting the analysis programs. Some experiments [5] support this assertion. The local analysis phase could generate native code and the global analysis would only need a relocation of imported and exported variables phase before running the global analysis program.

Also, the results outlined above show that `GOld`'s static analysis is not precise enough in some cases. At least two reasons explain this fact. First, numerically-intensive programs suffer from the set of numerical types in Scheme and the semantics of many numerical operators which make our analysis difficult. Secondly, the presence of a special form for defining structured datatypes could be of great help. Referencing a given position in a vector is approximated in the analysis by combining the approximations of all the elements in the vector. This is a problem with heterogeneous vectors, i.e. vectors in which the elements are of different types. This leads to approximations that are far too conservative.

Finally, even with a very naïve inlining scheme, cross-module inlining is worth its cost (relatively small in `GOld`). This is mainly due to the cost of cross-module calls in the case of small procedures.

6 Related work

Although `GOld` seems to be the first system based on abstract compilation for the implementation of the global analysis phase, it relates to a number of other systems we will now describe.

Bigloo [23, 25] is a Scheme to C compiler. But in contrast to Gambit-C, Bigloo does not entirely comply to the Scheme standard in order to generate more efficient code. For example, it does not optimize tail calls in general, but only in limited cases. Also, it relies on a sophisticated, first order module system. This system allows the programmer to supply the signature of the exported functions, easing the static analysis' job. But even with this module system, compilation flags must be specified when invoking the compiler to ensure optimal performances. For example, the `simplex` program runs 3.16 times faster when `-unsafe` flag is specified, telling Bigloo that all dynamic type tests and bounds checks can be omitted.

Stalin [28] is yet another Scheme to C compiler, developed by Jeffrey Mark Siskind. Like `GOld`, source programs are free of annotations. That's because Stalin relies on a sophisticated set-based static analysis. But unfortunately, all of the program must be given in a single module and compile times are quite long, even for very small programs.

CM [4] is a compilation management system for SML/NJ. It analyzes automatically the intermodule dependencies. The modules can be organized to form groups, which can themselves form supergroups, etc. CM also does cross-module inlining of functions, based on a technique called λ-splitting.

There is also MrSpidey [16], Rice's Scheme program development environment. Its graphical explanation facility is based on a constraint-based static analysis [15], the *componential set-based analysis*. This analysis uses several algorithms to try to minimize the number of constraints, without loosing information. It would be interesting to see how the constraint minimization algorithms could be used to optimize our analysis programs (either at abstract compilation time or at global analysis time, using just-in-time compilation).

For procedural languages, there are a number of systems optimizing object code at link-time [11, 10, 29, 30]. The work of Wall [30] is mainly concerned with the global allocation of registers at link-time. The system first compiles each module individually to object code, with certain instructions carefully annotated. These annotations indicate how the operands and the results of the annotated instruction are related to the global register allocation candidates. Only these instructions are considered by the optimization phase of the linker, which can modify them or remove them.

Srivastava and Wall [29] went one step further and developed OM, a system that performs many cross-module optimizations. It reads all the modules, transform them to a register transfer language (RTL), runs an interprocedural analysis, and performs various interprocedural optimizations, like loop-invariant code motion and dead-code elimination. It then translates the optimized RTL back to object code.

Chow[7] and Odnert and Santhanam [21] both describe systems for the global allocation of registers. These systems do not compile the modules to object code, though. They implement more sophisticated architectures.

However, we argue that object code is a too low-level program representation for performing cross-module optimizations of higher-order, dynamic languages. For instance, the `alto` system[11, 10] that performs several cross-module optimizations is only able to speed up Scheme programs generated by Gambit-C by 10% on the average.

Finally, Fernández [14] developed `mld`, a link-time optimizer for Modula-3 and C++. This system performs simple optimizations resembling partial evaluation like the replacement of generic method calls by direct calls to the appropriate method when the context allows it. It also uses profiling data to guide the optimization of most promising code segments. But unlike `alto`, this system translates each module to an intermediate representation. Machine code is generated only at link-time, after the global optimization phase. But unlike our work, the author was not really concerned with the speed of the analysis (which is quite low, due to its simplicity).

7 Conclusion

We have presented `GOld`, a link-time cross-module optimizer for Scheme based on the Gambit-C compiler. Its architecture is based on *abstract compilation*, a technique for the efficient implementation of static analyses devised in the abstract interpretation framework. We have shown that, using this system, multi-module, annotation-free Scheme programs can be made to run almost as fast as their fully annotated counterpart but with reasonable compile times.

8 Acknowledgments

I would like to thank Marc Feeley from the University of Montréal for supervising this work. I'm also grateful to Peter Stubley from Locus Dialogue who revised draft versions of this paper. Finally, I would like to thank the Scheme 2000 program committee for all their valuable comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachussets, 1986.

[2] J. L. Armstrong, S. R. Virding, and C. Wikström. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[3] M. Ashley. A Practical and Flexible Flow Analysis for Higher-Order Languages. In *Proceedings of the 1996 ACM Conference on Principles of Programming Languages*, 1996.

[4] M. Blume and A. Appel. Lambda-splitting: a higher-order approach to cross-module optimizations. In *Proceedings of the ACM SIGLAN Conference on Functional Programming*, 1997.

[5] D. Boucher. *Analyse et optimisation globales de modules compilés séparément.* PhD thesis, Université de Montréal, August 1999.

[6] D. Boucher and M. Feeley. Abstract compilation: a new implementation paradigm for static analysis. In *International Conference on Compiler Construction.* Springer-Verlag, 1996.

[7] F. C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, 1988.

[8] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximations of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[10] S. Debray, R. Muth, and S. Watterson. Link-time Improvement of Scheme Programs. In *Proc. 8th International Conference on Compiler Construction (CC'99)*, pages 76–90, Mars 1999.

[11] S. Debray, R. Muth, S. Watterson, and K. De Bosschere. `alto`: A Link-Time Optimizer for the DEC Alpha. Technical Report 98-14, Université d'Arizona, Décembre 1998.

[12] M. Feeley. Gambit-C version 2.8. `http://www.iro.umontreal.ca/~gambit`.

[13] M. Feeley and G. Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.

[14] M. F. Fernández. Simple and Effective Link-Time Optimization of Modula-3 Programs. In *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, Juin 1995.

[15] C. Flanagan and M. Felleisen. Componential Set-Based Analysis. *ACM Transactions on Programming Languages and Systems*, Février 1999.

[16] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the 1996 Conference on Programming Languages Design and Implementation*, 1996.

[17] M. V. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. In *Proceedings of the 1988 International Conference on Logic Programming*, 1988.

[18] S. Jagannathan and A. Wright. Effective flow-analysis for avoiding runtime checks. In *Proceedings of the 1995 ACM Conference on Principles of Programming Languages*, September, 1995.

[19] T. Johnsson. Lambda-lifting: Transforming programs to recursive equations. In *Proc. Conference on Functional Programming and Computer Architecture*. Springer-Verlag, 1985.

[20] I-P. Lin and J. Tan. Compiling Dataflow Analysis of Logic Programs. In *Conference on Programming Language Design and Implementation*, pages 106–115, 1992.

[21] D. Odnert and V. Santhanam. Register allocation across procedure and module boundaries. In *Conference on Programming Language Design and Implementation*, pages 28–39, June 1990.

[22] N. Ramsey. Relocating machine instructions by currying. In *ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, pages 226–236, Mai 1996.

[23] M. Serrano. *Vers une compilation portable* et *performante des langages fonctionnels*. PhD thesis, Université Pierre et Marie Curie (Paris VI), Paris, France, December 1994.

[24] M. Serrano. Control flow analysis: a compilation paradigm for functional language. In *Proceedings of SAC 95*, 1995.

[25] M. Serrano. Bigloo User's Manual. Technical report, Inria, Rocquencourt, March 1994.

[26] O. Shivers. Control Flow Analysis in Scheme. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[27] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[28] J. M. Siskind. Stalin, an Optimizing Compiler for Scheme. `ftp://ftp.nj.nec.com/pub/qobi/stalin.tar.Z`.

[29] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. Technical Report 92/6, DEC Western Research Laboratory, Palo Alto, California, December 1992.

[30] D. W. Wall. Global register allocation at link time. Technical Report 86/3, DEC Western Research Laboratory, Palo Alto, California, October 1986.