Control Flow Analysis: a Functional Languages Compilation Paradigm

Manuel Serrano

INRIA-Rocquencourt

Abstract

Control flow analysis (**cfa**) is now well known but is not widely used in real compilers since optimizations that can be achieved via **cfa** are not so clear. This paper aims at showing that control flow analysis is very valuable in practice by presenting a fundamental optimization based on **cfa**: the closure representation algorithm, the essential optimizing phase of a λ-language compiler. Since naïve and regular schemes to represent functions as heap allocated structures is far too inefficient, the main effort of modern functional languages compilers is devoted to minimize the amount of memory allocated for functions. In particular, compilers try to discover when a procedure can safely be handled without any allocation at all. Previously described methods to do so are *ad hoc*, depending on the language compiled, and not very precise. Using **cfa**, we present a general approach which produces better results. This refined closure analysis subsumes previously known ones and optimizes more than 90 % of closure on the average.
This optimization is fully integrated into the Bigloo compiler, so that we can report reliable measures obtained for real world programs. Time figures show that analyses based on **cfa** can be very efficient: when the compiler uses the improved closure allocation scheme the resulting executable programs run more than two time faster.
KEYWORDS: Scheme, ML, compilation, closure analysis, control flow analysis.

Introduction

For several years, control flow analysis (the determination of the call graph in the presence of functions as first-class values) has been studied in the literature about functional language compilation. Several theoretical models have been set up, and several algorithms have been suggested. Since these algorithms are complex, the attention has been focused on their design. But some crucial, although pragmatic, questions remain. Are these algorithms really useful in pratice ? What can they really improve ? This paper gives an important optimization for which **cfa** proves to be interesting: *the reduction of closures allocation.* Control flow analysis computes approximations of functional operators. These approximations are the basis of our closure allocation scheme. This optimizing closure analysis has a sound theoretical basis and subsumes previously known ones. This analysis optimizes more than 90 % of closures on the average.
The optimization described in this paper is fully integrated into the Bigloo Scheme compiler (Available by anonymous ftp from ftp.inria.fr [192.93.2.54], in the directory `/INRIA/Projects/icsla/Implementations` [8]). We can therefore report reliable measures of its effectiveness. Our figures show that closure optimization leads to executable programs running 70 % faster (see section 3).
The paper is organized as follows : section 1 presents the control flow analysis. Section 2 details the closure analysis. Section 3 gives benchmark figures and demonstrates the benefit of the analysis.

1 The control flow analysis

Control flow of modern functional language such as Scheme and ML, where functions are first-class citizens, may, by nature, be strongly dynamic. Nevertheless, static parts of the control can be revealed by control flow analysis.
The analysis we have made in Bigloo is close to the *"0cfa"* (0th-order Control Flow Analysis) described by O. Shivers in his Ph. D. thesis [10]. First, for each functional call in a program it statically computes an approximation of the set of functions that can be invoked in any execution. The approximations are sometimes too rough (for example, they contain too many elements to be relevant); but they are safe. Since the language we compile is the full Scheme language, we have been obliged to adapt Shivers' algorithm to deal with addtional constructs he did not consider. In addition, other data types approximations are computed.
Shivers has given a rigorous formalism to express a class of control flow analysis. On this basis, our work has focused on the utilizations of this analysis in real situations. For this reason, we shall just present the algorithm without recalling its theoretical basis nor proving its soundness (see Shivers' thesis).

1.1 The language used

Bigloo does not use continuation passing style (**cps**) as intermediate language. Therefore, by contrast to previous works, the language our **cfa** algorithm works on *is not* **cps**; it is a simplified direct style Scheme which looks like Lisp. Its grammar can be found below :

Syntactic categories

V	$\in$	VarId	(Variables identifier)
F	$\in$	FunId	(Functions identifier)
E	$\in$	Exp	(Expressions)
Π	$\in$	Prgm	(Program)
Γ	$\in$	Def	(Definition)
Σ	$\in$	Seq	(Non-empty sequence of expressions)

Concrete syntax

```
Π ::=   Γ...Γ
Γ ::=   (define (F V ... V) Σ)
      | (define V)
E ::=   V
      | (set! V E)
      | (labels ((F (V ... V) Σ) ... (F (V ... V) Σ)) Σ)
      | (if E E E)
      | (function F)
      | (funcall $V$ $E$ ... $E$)
      | (failure)
      | ($F$ $E$ ... $E$)
$\Sigma$ ::=  $E$ $E$ ... $E$
```

The keywords **define**, **set!**, and **if** belong to Scheme and have their usual meaning. Moreover, we have borrowed from Common Lisp **function**, **funcall** and **labels**. The **failure** form allows us to stop the computation; its semantics specifies that its call continuation will not be invoked. Moreover our language offers modules in which variables can be imported, exported or static (local to a module).

Note: The **lambda** form does not exist, but functional values can be obtained by composing **function** and **labels**. Therefore, what is usually written in Scheme : `(lambda (...) ...)`, must be written in our language : `(labels ((`*id*` (...) ...)) (function `*id*`))`.

Note: The **call/cc** function does not appear in our language since it is not a special form but just a library function. This function does not need special processing.

1.2 The algorithm

We now present the approximation algorithm. In section ?? we use it to approximate types, the reader may refer to this section to get an intuitive idea of the general approximation algorithm. The algorithm performs a simple case analysis of its program argument. It needs informations about the identifiers appearing in the program to compute approximations. These informations about variables are described by five logical properties : a variable class predicate, function or not function ($\mathcal{FUN}$), and we define four locality properties: $\mathcal{FOR}$ and $\mathcal{ESC}$ for variables bound to functions, and $\mathcal{LOC}$, $\mathcal{GLO}$ for other variables.

Formally, for all the variables appearing in a program :

$\mathcal{LOC}(v)$	$\Leftrightarrow v$ is a <u>loc</u>al variable.
$\mathcal{GLO}(v)$	$\Leftrightarrow v$ is a <u>glo</u>bal variable.
$\mathcal{FOR}(v)$	$\Leftrightarrow v$ is a <u>for</u>eign function defined in another language (the implementation language, e.g. C or assembler).
$\mathcal{ESC}(v)$	$\Leftrightarrow v$ is an <u>esc</u>aping function (defined in another module or exported).

And for all the approximations computed by the algorithm :
$\mathcal{FUN}(x) \Leftrightarrow x \in$ FunId.

Note: Only global functions can be exported or imported, so they are the only ones that can satisfy the $\mathcal{ESC}$ predicate.

The abstract syntax tree is annotated with subsets of the following set : $R = \{\top, \bot\} \bigcup$ FunId

Approximations are sets whose elements are types, functions or two specific values, $\top$ denoting the *undefined* object and $\bot$ an approximation not yet computed. Every approximation containing $\top$ is undefined. These approximations are obtained using the $\mathcal{A}$ function. Initially, all the approximations have $\{\bot\}$ as value. The only operation defined on approximations, named `add-app!` is the extension of an approximation by a value. It is defined as follows :

$$\forall x \in R, v \in \text{VarId} \cup \text{FunId}, \; if \; y = \mathcal{A}(v) \; then$$
$$\texttt{add-app!}(v, x) \Rightarrow \mathcal{A}(v) = \texttt{if } x = \bot \texttt{ then } y \texttt{ else } \{x\} \bigcup y.$$

Functions being complex objects, we access their different slots using the following projections, $\downarrow_{body}$ for their body and $\downarrow_{formals_i}$ for their i^th formal parameter. To get the approximations, we process a fix point iteration (until no new approximations are added) with the following algorithm :

```
Ocfa-exp( exp ) =
      case exp of
         [ var ] :
            Ocfa-var( var )
         [ (set! var val) ] :
            Ocfa-set!( var, val )
         [ (labels (($f_1$ ($a_{1_1}$ ... $a_{1_{m_1}}$) $e_1$) ... ($f_n$ ...)) exp) ] :
            Ocfa-exp( exp )
         [ (if test then else) ] :
            Ocfa-exp( test ),
            Ocfa-exp( then ) $\bigcup$ Ocfa-exp( else )
         [ (function f) ] :
            { f }
         [ (funcall fun $a_1$ ... $a_n$) ] :
            Ocfa-unknown-app(Ocfa-exp(fun), ..., Ocfa-exp($a_n$))
         [ (failure) ] :
            { $\bot$ }
         [ (fun $a_1$ ... $a_n$) ] :
            Ocfa-known-app(fun, ..., Ocfa-exp($a_n$))
Ocfa-var( var ) =
      cond
         $\mathcal{LOC}$(var) : $\mathcal{A}$( var )
         $\mathcal{GLO}$(var) : { $\top$ }
Ocfa-set!( var, val ) =
      cond
         $\mathcal{LOC}$(var) :
            $\forall x \in$ Ocfa-exp( val ), add-app!( var, $x$ )
         $\mathcal{GLO}$(var) :
            set-top!( Ocfa-exp( val ) )
Ocfa-unknown-app( A, $A_1$, ..., $A_n$ ) =
      $\bigcup_{f \in A}$ Ocfa-try-app( f, $A_1$, ..., $A_n$ )

Ocfa-try-app( f, $A_1$, ..., $A_n$ ) =
      cond
         $\mathcal{FUN}$( f ) :
            Ocfa-known-app( f, $A_1$, ..., $A_n$ )
         $\top$ = f :
            $\forall i \in [1,n]$, set-top!( $A_i$ ), { $\top$ }
         else :
            Ocfa-error()
Ocfa-known-app( var, $A_1$, ..., $A_n$ ) =
      cond
         $\mathcal{ESC}$( var ) :
            set-top!( Ocfa-exp( var$\downarrow_{body}$ ) ),
            $\forall i \in [1,n]$ set-top!( $A_i$ ), { $\top$ }
         $\mathcal{FOR}$( var ) :
            Ocfa-foreign-app( var, $A_1$, ..., $A_n$ )
         else :
            Ocfa-function-body( var, $A_1$, ..., $A_n$ )
Ocfa-function-body( var, $A_1$, ..., $A_n$ ) =
      $\forall i \in [1,n]$, $\forall x \in A_i$ add-app!( var$\downarrow_{formals_i}$, $x$ ),
      Ocfa-exp( var$\downarrow_{body}$ )
set-top!( A )
      $\forall$ a $\in$ A, set-one-top!( a )
set-one-top!( a )
      cond
         $\mathcal{FUN}$( a ) $\wedge$ ($\mathcal{FOR}$( a ) $\vee$ $\mathcal{ESC}$( a )) :
            $\forall i \in [1,n]$, set-top!(a$\downarrow_{formals_i}$)
         else : add-app!( a, $\top$ )
```

Our algorithm is presented in a simplified way, as programs are supposed to be correct and fully alpha-converted. Furthermore, one can see that our algorithm could fall in an infinite loop. This could arise when approximating self-recursive functions (in the function *Ocfa-known-app*). This is straightforwardly avoided by using a stamp technique that prevents computing several approximations of the same function in the same iteration. These simplifications do not affect the rest of the paper.

2 Closures allocation

Scheme is dynamically typed, computed functional calls, where the function is not a constant syntactically known or recognized during compilation, must be identified. It is mandatory to check that the object to be applied is a function and that its arity is compatible with the number of arguments provided. Furthermore, Scheme has two different types of functions, fix arity and variable arity; procedures can have two entry points; one for each type of function. As a consequence, the minimum size of a procedure (without its environment) is four words. We show in this Section that using the *0cfa* analysis, we can reduce the size of procedures. Such an optimization depends on the way procedures are used. For that purpose, we introduce the concept of *procedure family*. Intuitively, the set FunId of program functions is partitioned in such a way that all elements belonging to the same partition are applied at the same places in the program.

2.1 Procedures family

Let us first give some definitions. Let *SITE* be the set of call sites of the program. For each element of *SITE*, the *0cfa* has computed the functions approximations. That is, for all $s = [(\texttt{funcall}\ f\ldots)]$ belonging to the set *SITE*, the *0cfa* has computed $\mathcal{A}(f)$.
We introduce the function $\mathcal{USE}$ which characterizes the use of program functions. We recall that $\mathcal{A}$ gives, for each call, the set of functions that can be applied. By contrast, $\mathcal{USE}$ gives, for each function, the set of sites where it can be invoked. Its definition can be found below :

$$\forall f \in \text{FunId}, \mathcal{USE}(f) = \\ \{s \in SITE \mid s = [(\texttt{funcall}\ g\ldots)] \wedge f \in \mathcal{A}(g)\}$$

Let us introduce three properties, $\mathcal{T}$, $\mathcal{X}$ and $\mathcal{S}$.

$\mathcal{T}$ property: A function f satisfies $\mathcal{T}$ if for all its call sites $[(\texttt{funcall}\ g\ldots)]$, all the elements belonging to the approximations set of g are functions also satisfying the $\mathcal{T}$ predicate. This predicate allows to group together functions according to their utilization family. Its definition is :

$$\forall f \in \text{FunId}, \mathcal{T}(f) \Leftrightarrow \\ \neg\mathcal{ESC}(f) \wedge \forall s = [(\texttt{funcall}\ g\ldots)] \in \mathcal{USE}(f) \\ \forall a \in \mathcal{A}(g), a \neq f, \mathcal{FUN}(a) \wedge \mathcal{T}(a)$$

$\mathcal{X}$ property: A function satisfies the $\mathcal{X}$ predicate if for every call site, it is the only function that can be invoked. The definition of $\mathcal{X}$ is :

$$\forall f \in \text{FunId}, \mathcal{X}(f) \Leftrightarrow \\ \neg\mathcal{ESC}(f) \wedge (\forall s = [(\texttt{funcall}\ g\ldots)] \in \mathcal{USE}(f), \mathcal{A}(g) = \{f\})$$

$\mathcal{S}$ property: A function f satisfies the $\mathcal{S}$ property if it does not exist any `funcall` site where f can be invoked. The definition of $\mathcal{S}$ is :

$$\forall f \in \text{FunId}, \mathcal{S}(f) \Leftrightarrow \neg\mathcal{ESC}(f) \wedge \mathcal{USE}(f) = \emptyset$$

Note: $\forall f \in FunId,\ \mathcal{S}(f) \Rightarrow \mathcal{X}(f),\ \mathcal{X}(f) \Rightarrow \mathcal{T}(f)$

Definition 1 *Let s be* $[(\texttt{funcall}\ f\ldots)]$. *If there is a function in* $\mathcal{A}(f)$ *that satisfies the* $\mathcal{T}$ *predicate, then* $\mathcal{A}(f)$ *is called a family of procedures.*

Proposition 1 *For every f in FunId, if f satisfies the* $\mathcal{S}$ *predicate, then f does not require any allocation, nor for a structure carrying the closure, neither for any environment.*

If a function f satisfies the $\mathcal{S}$ predicate, all invocations to f are direct (not computed with the `funcall` form) and can only occur in the definition scope of f. Then, each call to f can be compiled as a direct branch (if f has free variables, they are added to the list of the formal arguments) □

Proposition 2 *Functions which are never used as actual arguments nor returned as values satisfy the* $\mathcal{S}$ *predicate.*

This proposition is obvious because those functions are always directly invoked (by contrast to the computed calls); they never appear in any `funcall` □

Note: This proposition is very important because most of the functions are always directly used. Our control analysis can be here widely applied.

Proposition 3 *For every f in FunId, if f satisfies the* $\mathcal{X}$ *predicate, then f does not require closure structure allocation.*[1]

Indeed, if a function f satisfies the $\mathcal{X}$ predicate then

$$\forall s = [(\texttt{funcall}\ g\ldots)] \in \mathcal{USE}(f), \mathcal{A}(g) = \{f\}$$

This means that only f can be invoked. The computed call can be replaced (using a *"lambda lifting"* [3] pass if needed) by a direct call which does not require any allocation. □

Proposition 4 *For each f in FunId, if f satisfies the* $\mathcal{T}$ *predicate then f can be allocated in an more efficient way (i.e. with no tag, no arity, and with only one entry point slot).*

This is true because if a function f satisfies the $\mathcal{T}$ predicate, it means that :
$\forall s = [(\texttt{funcall}\ g\ldots)] \in \mathcal{USE}(f), \forall a \in \mathcal{A}(g), a \in \text{FunId}$.
This family is known at compilation time, hence, type correctness can be statically checked. Furthermore, if functions with variable arity are forbidden to satisfy the $\mathcal{T}$ predicate, then one entry point is enough. □

2.2 Compilation improvements

We show in this section how the Bigloo compiler uses these three predicates. Four constructs can be improved: (`function`, `funcall`, `labels` and the global definition). Here are the improvements for each of them.

- `(labels ((`f`...)...)...)` or `(define (`f`...)...)` : If f satisfies the $\mathcal{S}$ predicate, then no closure is built for it. Function calls to f will be direct branches.
- `(function `f`)` :

  o Trivial case : if f satisfies the $\mathcal{X}$ predicate and if all elements of $\mathcal{USE}(f)$ are in the lexical scope of f, the form is removed.

  o If f satisfies the $\mathcal{X}$ predicate and if f has zero or one free variable, then no allocation is required, The form is replaced by the free variable. If f has several free variables, the form is replaced by the (allocated) list of the free variables.

  o If f satisfies the $\mathcal{T}$ predicate, then neither entry point for functions with variable arity, nor tag, nor arity slot have to be allocated ; hence a smaller structure is allocated.
- `(funcall `f` ...)` :

  o Let `[(funcall f ...)]` be a call site. If there exists an unique g in $\mathcal{A}(f)$ that satisfies the $\mathcal{X}$ predicate, then we compile a direct application.

  o Let `[(funcall f ...)]` be a call site. If all elements of $\mathcal{A}(f)$ satisfy the $\mathcal{T}$ predicate, we compile a computed application which is "easy" that is, with no type checking or arity checking.

[1]No allocation is required for the structure representing the closure ; by contrast, an environment can be allocated.

- (**funcall** $f \ldots$) or ($f \ldots$) : If the result of the application is known and if we know that the function invoked does not have any side effects (this information is obtained by a previous pass), we replace the call by its result.

2.3 Applications

We study some typical examples to illustrate the improvements described above. Our goal is to provide an intuitive idea of the optimizations carried out. Section 3 is devoted to an analysis of the improved performances obtained by the techniques previously described.

$\mathcal{S}$: **First order functions** The program:

```
(letrec ((fib (lambda (x)
        (if (< x 2) 1 (+ (fib (- x 1)) (fib (- x 2)))))))
  (fib 20))
```

includes no function used as actual argument or returned value. Then, the function `fib` satisfies the $\mathcal{S}$ predicate. No allocation is required to run this program. At runtime, the `fib` function will only be a code entry point.
This optimization is very important since like `fib`, many functions satisfy the $\mathcal{S}$ predicate. The measures of section 3 will prove this.

$\mathcal{X}$: **Currying** In ML, all functions are unary; a natural way to express functions with a higher arity is currying. The following program illustrates the improvement made when a function satisfies the $\mathcal{X}$ predicate.

```
(define plus (lambda (x) (lambda (y) (+ x y))))
(define foo (lambda (x y) ((plus x) y)))
```

The function `(lambda (y) (+ x y))` does, since it is only applied in the expression `((plus x) y)`. The nature of its allocation is changed by replacing it by the list of its free variables. Here, `x` is the only free variable. After optimization, the code becomes:

```
(define plus (lambda (x) x))
(define lambda-1 (lambda (env y) (+ env y)))
(define foo (lambda (x y) (lambda-1 (plus x) y)))
```

One can see that, in this case, no further allocation is needed.

$\mathcal{T}$: **Denotational semantics** Given a denotational semantics, our analysis is able to proves to all closures allocated to evaluated termes are applied at a same point in a program. All these closures satisfies the $\mathcal{T}$ predicates, hence all these functions form a family. It is possible to obtain a "byte-code" interpreter by carrying out an optimization of this family of procedures : rather than allocating closure, one can decide to give them a unique identifier and to represent them as a pair $< num \times env >$. Each calls to one of these procedures would become an indexed jump to the first projection of these pairs. These first projections could then be seen as "byte-codes" and the place in the program function are applied could be seen as "byte-code" interpreters.

3 Measures

This Section presents the results obtained by measuring code size, compilation times and execution times with and without *0cfa*-based optimisations. We investigate different programs, written in different styles by different people. They do not describe all realistic situations one can face but help to get an accurate idea of the optimization results obtained.
The performances we give have been obtained using our compiler, which generates C code. Three compilations have been launched for each program : the first one without any optimization, the second with some optimizations (common sub-expression elimination) (**O**), and finally, one compilation with all optimizations, including the one presented in this paper (**O2**). For all the compilation modes so far described, we have evaluated the number of closures allocations. We have also measured the compilations time. For each time figure, δ is the ratio $1 - O2/O$. Time figures (expressed in seconds) present the average of several consecutive executions run on a Sun 4/670 (SS-2 equivalent). For each programs, **compil.** is the time in seconds to produce C code; **compil.+cc** is the global compilation time (with linking); **.o size** is the size of the file obtained after the C compilation (expressed in Kilo bytes); **proc** is the number of procedure allocations without any optimizations. In the **O2** column, it represents the number of closures which are not optimized; $\mathcal{T}$ is the number of procedures that have been simplified since they satisfy the $\mathcal{T}$ predicate; $\mathcal{X}$ is the number of procedures eliminated since they satisfy the $\mathcal{X}$ predicate; $\mathcal{S}$ is the number of procedures eliminated since they satisfy the $\mathcal{S}$ predicate; **run** is the execution time of the program.
Conform (569 lines, 5 iterations) uses many functions, but few of them are required to be allocated. The closure optimization has a great impact on this program.

Conform		O	O2	δ
compil.	6.4 s	6.4 s	60.5 s	-845 %
compil.+cc	72.6 s	66.1 s	100.5 s	-52 %
.o size (Kbytes)	144	136	52	61 %
proc	94	94	5	95 %
$\mathcal{T}$	-	-	0	-
$\mathcal{X}$	-	-	16	-
$\mathcal{S}$	-	-	73	-
run	52.2 s	43.4 s	15.8 s	90 %

Earley (661 lines, 5 iterations) uses vectors and each access to one of them requires a bound check. These numerous tests make the syntax tree wery large with many function calls. This slows down the *0cfa*.

Earley		O	O2	δ
compil.	5.7 s	5.7 s	11.9 s	-108 %
compil.+cc	21.5 s	21.5 s	21.1 s	2 %
.o size (Kbytes)	120	90	33	63 %
proc	75	75	4	95 %
$\mathcal{T}$	-	-	0	-
$\mathcal{X}$	-	-	0	-
$\mathcal{S}$	-	-	71	-
run	33.7 s	33.2 s	7.0 s	79 %

Semantics (231 lines, 5 iterations) being a denotational semantics, the optimization described in Section 2.3 is applied. This explains why there is so many functions satisfying the $\mathcal{T}$ predicate in this example.

Semantics		O	O2	δ
compil.	4.2 s	4 s	13.1 s	-227 %
compil.+cc	29.8 s	29.4 s	23.4 s	20 %
.o size (Kbytes)	49	45	18	60 %
proc	54	54	8	85 %
$\mathcal{T}$	-	-	36	-
$\mathcal{X}$	-	-	4	-
$\mathcal{S}$	-	-	8	-
run	32.5 s	32.6 s	15.4 s	53 %

- Since our benchmarks are not usual (the traditional Gabriel benchmarks do not fit our needs since they are rather small and they are not higher order), we shortly give the time figures for another Scheme to C compiler. We chose Bartlett's one [1]. The measures have been obtained with the March 15th, 1993 release. All produced executables are unsafe and use fixnum arithmetic.

scc	Semantics	Conform	Earley
compil.+cc	29.9 s	55 s	111.5 s
run	30.6 s	35.4 s	9.7 s

• The *0cfa* analysis is rather time consuming and it has a high complexity in the worst case: ($\mathcal{O}(n^3)$ where n is the number of functions and calls). Furthermore it significantly increases the time spent in the first part of the compilation, although at an acceptable level. But, paradoxically, sometimes (**earley** and **semantics**) it allows to have a faster global compilation due to a better generated C code. Since the C compiler runs more slowly than Bigloo, the time lost in the first passes is compensated by the time gained in the last ones. This paradox can be observed for programs where the *0cfa* reaches efficient results. Of course, for the others, the time lost is not compensated. In practice, for real-sized programs, the *0cfa* analysis is perfectly usable. For example, Bigloo's bootstrap (more than 30.000 Scheme lines) takes 45 minutes without *0cfa* and 55 minutes using it.
Moreover, as shown in the benchmark results, the *0cfa* seems to require few iterations to reach the fix point. For our examples, the maximum number of iterations is only 5. We used the analysis in the "real world" and we never found programs that required an polynomial number of iterations.

4 Related works

Three kinds of work are related to ours. First of them, the studies of control flow.

• Olin Shivers has published numerous papers on control flow analysis in modern functional languages such as Scheme or ML[9, 10]. His aim was to study the analysis as well as its applications : the first half of his PhD thesis is devoted to the analysis semantics and the second to application examples. He has defined a general analysis of which the *0cfa* is a special case. He has also used a more precise analysis called the *"1cfa"*, which was prototyped in T [4], a Scheme compiler. Shivers has briefly given time figures for the *1cfa* analysis but no precise measures of its costs and benefits. Therefore no conclusions can be drawn on its relevance in real situations. Shivers has shown several applications of the control flow analysis but the optimizations he has mentioned (classical optimizations of "data flow analysis") are already made in our compiler *without* the *0cfa*. The main reason is that, as T uses *cps* as intermediate language, the control is very dynamic. In this case, a control flow analysis is mandatory to isolate static parts of the control. We do not have these problems since we use a suitable intermediate language. As a consequence, we have advantageously used the *0cfa* analysis to perform others optimizations such as the closures allocation optimization.
1cfa gives more refined approximations: rather than only knowing which functions are invoked on each call site, the *1cfa* approximations indicate how functions have been passed around to reach the call site. But we don't think that this more refined approximations help the compilation: what can be done with this information in a compiler ? We have found no answers to this question. This is the reason why we chose the *0cfa* rather than the *1cfa* in our compiler.

• Guillermo Rozas shows, in his paper [6], how by using a technique close to the *0cfa*, he is able to compile in the same way a program written with the fix point operator Y, and a program using special forms to introduce local recursive calls. His paper focused on how the analysis works rather than on its possible uses. He stressed problems we have not mentioned here, such as the elimination of useless environments, or the errors that can be introduced by the results of a control analysis. Unfortunately, although the optimizations have been implemented in his compiler (liar), he gave no measures allowing to have an idea of their impact.

• Previous closure analyses have to be compared to the one presented in this article. D. Kranz's PhD thesis [5] and N. Séniak's one [7] are devoted to closure analysis. Both of them describe almost the same analysis. They divide functions in two sets: the ones requiring environment allocation and the ones which do not. The algorithms presented in these theses to decide to allocate or not closures correspond exactly to the computation of our $\mathcal{S}$ property. This means that, in those contexts, if a function does not satisfy $\mathcal{S}$, then its closure is allocated. In our case, it will be so only if it does not satisfy, in addition to $\mathcal{S}$, the $\mathcal{X}$ and $\mathcal{T}$ predicates. Since Kranz' and Séniak's analyses are strictly subsumed by ours, the results obtained with *0cfa* analysis are at least as good as theirs.

Conclusion

Using an already known analysis (the *0cfa*), this paper describes a new optimization. It concerns closure allocations. For several "real programs" we measure that the new closures optimization removes 87 % of allocations. This improvements represent a gain of 70 % on execution times and 60 % on the sizes of the object files produced (more than two times smaller). All modern functional languages such as ML and Scheme are in the scope of the presented optimization.

References

[1] J.F. Bartlett. Scheme->C a Portable Scheme-to-C Compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California, January 1989.

[2] P.H. Hartel, H. Glase, and J.M. Wild. Compilation of Functional Languages Using Flow Graph Analysis. *Software — Practice and Experience*, 24(2):127–173, February 1994.

[3] T. Johnson. *Lambda Lifting :* Transforming Programs to Recursive Equations. In *Proceedings of the ACM Conference on Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

[4] D. Kranz, R. Kesley, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

[5] D.A. Kranz. *ORBIT: An Optimizing Compiler For Scheme.* PhD thesis, Yale university, February 1988.

[6] G.J. Rozas. Taming the Y operator. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 226–234. ACM, June 1992.

[7] N. Séniak. *Théorie et pratique de Sqil: un langage intermédiaire pour la compilation des langages fonctionnels.* PhD thesis, Université Pierre et Marie Curie (Paris VI), November 1991.

[8] M. Serrano. Bigloo user's manual. Technical Report to appear, INRIA-Rocquencourt, France, 1994.

[9] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[10] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.