Control flow analysis: a compilation paradigm for functional language

Manuel Serrano

INRIA

B.P. 105, Rocquencourt, 78153 Le Chesnay Cedex, FRANCE

Tel: +33 (1) 39 63 57 32. Fax: +33 (1) 39 63 53 30. E-mail: Manuel.Serrano@inria.fr

Revised version of "SAC 95" [31/08/95]

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 scheme 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 80 % of closure on the average.

We also present another analysis based on **cfa** which is useful for dynamically typed languages: the type investigation algorithm. Modifying Shivers' **cfa**, we realized a type inference algorithm that eliminates more than 60 % of dynamic type tests.

These two optimizations are fully integrated into a real 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 and the type investigation algorithm, the resulting executable programs run more than two time faster.

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 two important optimizations for which **cfa** proves to be interesting:

1. **Reduction of dynamic type tests :** Starting from a well-known control flow analysis, we have derived a type inference algorithm by computing type approximations rather than functional approximations. The new algorithm eliminates 65 % of dynamic type tests.

2. **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 theoritical basis and subsumes previously known ones. This analysis optimizes more than 80 % of closures on the average.

The first optimization concerns compilers of dynamically typed languages such as Scheme [17]; the second one is of more general interest: it applies to every λ-language compiler, including those statically type checked (such as ML [4, 20]).

The optimization described in this paper are fully integrated into a real Scheme compiler (Bigloo). We can therefore report reliable measures of their effectiveness. Our figures show that reduction of dynamic type tests combined with closure optimization leads to executable programs running 65 % faster (see section 4).

The paper is organized as follows : section 1 presents the control flow analysis. Section 2 studies the type investigation algorithm. Section 3 details the closure analysis. Section 4 gives benchmark figures and demonstrates the benefit of these two analyses.

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 PhD thesis [14]. 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)
K	$\in$	Cnst	(Constant values)
Π	$\in$	Prgm	(Program)
Γ	$\in$	Def	(Definition)
Σ	$\in$	Seq	(Non-empty sequence of expressions)

Concrete syntax

Π ::= $\Gamma \ldots \Gamma$
Γ ::= (**define** (F $V \ldots V$) Σ)
| (**define** V)
E ::= V
| (**quote** K)
| (**set!** V E)
| (**labels** ((F ($V \ldots V$) Σ) $\ldots$ (F ($V \ldots V$) Σ)) Σ)
| (**if** E E E)
| (**begin** Σ)
| (**function** F)
| (**funcall** V $E \ldots E$)
| (**failure**)
| (F $E \ldots E$)
Σ ::= E $E \ldots E$

The keywords `define`, `set!`, `if`, `quote` and `begin` belong to Scheme and have their usual meaning. Moreover, we have borrowed from Common Lisp [18] `function`, `funcall` and `labels`. The `failure` form allows 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))
```

The influence of N. Séniak [12] compilation methodology (where dynamic parts of the control are isolated) has led to the removal of `lambda` form. Functions, when used as values, are always processed by the `function` special form; this simplifies the remainder of the compilation.

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.
As functions can be introduced solely by variables, when in the rest of this text, we speak of the lexical scope of a function, it should be understood, as the scope of the variable which has introduced the function.

1.2 The algorithm

We now present the approximation algorithm. In section 2 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 local variable.
$\mathcal{GLO}(v)$	$\Leftrightarrow$	v is a global variable.
$\mathcal{FOR}(v)$	$\Leftrightarrow$	v is a foreign function defined in another language (the implementation language, e.g. C or assembler).
$\mathcal{ESC}(v)$	$\Leftrightarrow$	v is an escaping 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 :
$R = \{ \top, \bot, \text{int, string, char, ... } \} \bigcup \text{FunId}$
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 :

0cfa-exp(exp) =
  case exp **of**
    ⟦ var ⟧ :
      0cfa-var(var)
    ⟦ (quote datum) ⟧ :
      0cfa-quote(datum)
    ⟦ (set! var val) ⟧ :
      0cfa-set!(var, val)
    ⟦ (labels ((f$_1$ (a$_{1_1}$ ... a$_{1_{m_1}}$) e$_1$) ... (f$_n$ ...)) exp) ⟧ :
      0cfa-exp(exp)
    ⟦ (if test then else) ⟧ :
      0cfa-exp(test),
      0cfa-exp(then) ⋃ *0cfa-exp*(else)
    ⟦ (begin exp$_1$ ... exp$_n$) ⟧ :
      0cfa-exp(exp$_1$), ..., *0cfa-exp*(exp$_n$)
    ⟦ (function f) ⟧ :
      { f }
    ⟦ (funcall fun a$_1$ ... a$_n$) ⟧ :
      0cfa-unknown-app(*0cfa-exp*(fun), ..., *0cfa-exp*(a$_n$))
    ⟦ (failure) ⟧ :
      { ⊥ }
    ⟦ (fun a$_1$ ... a$_n$) ⟧ :
      0cfa-known-app(fun, ..., *0cfa-exp*(a$_n$))

0cfa-var(var) =
  cond
    $\mathcal{LOC}$(var) : $\mathcal{A}$(var)
    $\mathcal{GLO}$(var) : { $\top$ }

0cfa-quote(exp) =
  { `type-of`(exp) }

0cfa-set!(var, val) =
  cond
    $\mathcal{LOC}$(var) :
      $\forall x \in$ *0cfa-exp*(val), `add-app!`(var, x)
    $\mathcal{GLO}$(var) :
      set-top!(*0cfa-exp*(val))

0cfa-unknown-app(A, A$_1$, ..., A$_n$) =
  $\bigcup_{f \in A}$ *0cfa-try-app*(f, A$_1$, ..., A_n)

0cfa-try-app(f, A$_1$, ..., A$_n$) =
  cond
    $\mathcal{FUN}$(f) :
      0cfa-known-app(f, A$_1$, ..., A$_n$)
    $\top$ = f :
      $\forall i \in [1,n]$, *set-top!*(A$_i$), { $\top$ }
    else :
      0cfa-error()

0cfa-known-app(var, A$_1$, ..., A$_n$) =
  cond
    $\mathcal{ESC}$(var) :
      set-top!(*0cfa-exp*(var$\downarrow_{body}$)),
      $\forall i \in [1,n]$ *set-top!*(A$_i$), { $\top$ }
    $\mathcal{FOR}$(var) :
      0cfa-foreign-app(var, A$_1$, ..., A$_n$)
    else :
      0cfa-function-body(var, A$_1$, ..., A$_n$)

0cfa-function-body(var, A$_1$, ..., A$_n$) =
  $\forall i \in [1,n], \forall x \in$ A$_i$ `add-app!`(var$\downarrow_{formals_i}$, x),
  0cfa-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 *0cfa-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 Types investigation

Scheme is dynamically typed. To avoid unpredictable behavior, types have to be checked at run time. For example, before accessing the `car` of an object, it is necessary to verify that this object is a `pair`. Usually numerous, these tests have two important consequences : they increase the size of the compiled code and they slow down the execution. We show in this section that, by improving the algorithm given in section 1.2, we are able to eliminate many type tests while ensuring correct executions.
Before going further we have to describe the method chosen to introduce these dynamic tests type for Scheme programs. The sole operations requiring type verifications are :

- Data structures access.
- Arithmetic operations.
- Computed functional call.[1]

These three kinds of operations are, either for efficiency reasons or by necessity, implemented in the target language. This language is inevitably known by the compiler but does not play any distinct role. During the compilation, it is considered as a foreign language with which the source language should be interfaced. From this point of view, all operations requiring type verifications are implemented in the target language. Consequently, only interface between the source and target languages may introduce type checking. We call *foreign interface* all constructions allowing the utilization of programs written in another language than Scheme. Foreign interface allows to invoke functions that will return Scheme refined type objects, i.e. objects whose exact type is known (`string`, `fixnum`, etc.). As allocation functions are also written in the target language, we know during the compilation where the refined objects are introduced. This information is propagated by the algorithm given in Section 1.2 so that it is often possible to infer the type of a variable. Since only foreign calls can introduce refined objects, the propagation process is initialized in the *0cfa-foreign-app*, so far left undefined. The reader can find below a simplified version of this function.

0cfa-foreign-app(f, A_1, ..., A_n) =
$\forall i \in [1, n]$ *set-top!*(A_i),
{ `type-of-foreign-result`(f) }

Note: Some library functions have, to a certain extent, a special status. The compiler must know them. This is the case for example, for the two functions `push-trace` and `pop-trace`; they allow to keep a stack remembering execution trace of functions. When the compiler is invoked in debugging mode, it inserts the first one of these functions at the start and the second one at the exit of all the program functions. A debugged function has the following skeleton :

`(lambda (x) (push-trace` *id*`) (pop-trace` $id\downarrow_{body}$`))`

The two functions `push-trace` and `pop-trace` have not been satisfactorily written in Scheme. We have been obliged to implement them in the target language. As a result, if the *0cfa* does not take them into account, all the analysis will become irrelevant because all values will have $\top$ as approximation. This example points out that the function *0cfa-foreign-app* has to discriminate some foreign functions.

[1] Non computed functional calls can easily be checked during compilation.

Once all variables approximations have been collected by the control analysis, the applications related to type checkers (`integer?`, `procedure?`, ...) whose results are now known are removed. These functions, as well as the constructors, are written in the target language. As a consequence, they appear in the calls of foreign functions.
Let us look at the whole Scheme program of Figure 1.

```
(define (foo f x)
   (if (integer? x)
       (integer? (f x))
       (if (real? x)
           (integer? (f (inexact->exact x))))))

(define (bar)
   (define (square y)
      ;;  *fx denotes integer multiplication
      (*fx y y))
   (foo square 4))

(bar)
```

Figure 1:

This program written in the language defined in Section 1.1 and, after macro expansion and insertion of type checking is presented in Figure 2. Figure 3 shows the same program after exploitation of *0cfa*'s results.

```
(define (foo f x)
   (let ((obj (funcall f x)))
      #t))

(define (bar)
   (labels ((square (y) (*fx y y)))
      (foo (function square) 4)))

(bar)
```

Figure 3:

Two iterations were necessary to obtain this result. Here is the way the algorithm ran :

1. '`(bar)`' (line *20*) $\Rightarrow$
   inspection of the body of `bar`
2. *add-app!*(`square`, *square*)
   where *square* $\equiv \lambda y.(*fx\ y\ y)$
3. '`(foo (function square) 4)`' (line *18*) $\Rightarrow$
   add-app!(`f`, *square*), *add-app!*(`x`, *integer*)
4. '`(let ((obj (funcall ...))) ...`' (line *3*) $\Rightarrow$
   add-app!(`y`, *integer*),
   inspection of the body of `square`
5. '`(*fx y y)`'

```
01: (define (foo f x)
02:    (if (integer? x)
03:       (let ((obj (funcall (let ((f (if (procedure? f) f (failure))))
04:                             (if (= (procedure-arity f) 1) f (failure)))
05:                           x)))
06:            (integer? obj))
07:        (if (or (integer? x) (flonum? x))
08:            (let ((obj (funcall (let ((f (if (procedure? f) f (failure))))
09:                                  (if (= (procedure-arity f) 1) f (failure)))
10:                                (if (flonum? x)
11:                                    (flonum->fixnum (if (flomun? x) x (failure)))
12:                                    x))))
13:               (integer? obj))
14:            #f)))
15:
16: (define (bar)
17:    (labels ((square (y) (if (integer? y) (*fx y y) (failure))))
18:       (foo (function square) 4)))
19:
20: (bar)
```

Figure 2:

6. *add-app!*(`obj`, *integer*)

To summarize, the following approximations are provided by the algorithm :

$\mathcal{A}$(`square`) = { *square* }
$\mathcal{A}$(`f`) = { *square* }
$\mathcal{A}$(`x`) = { *integer* }
$\mathcal{A}$(`y`) = { *integer* }
$\mathcal{A}$(`obj`) = { *integer* }

Let us see another example in which the *0cfa* allows an efficient compilation. Let us consider the code of `read-char`, a Scheme library function. One implementation can be as given in Figure 4, `read-char/port` being a function physically reading the stream indicated by its parameter.

```
(define (read-char . port)
   (let ((port (if (null? port)
                   (current-input-port)
                   (car port))))
      (if (input-port port)
          (read-char/port port)
          (failure))))

(define (user)
   (read-char))
```

Figure 4:

For efficiency purposes, the compiler can systematically inline the specific function `read-char`. After this inlining, the Figure 5 shows the `user` function code.

After control analysis and elimination of unecessary tests, the code would be as shown in Figure 6 (the presented code will be submitted to other optimizations such as *constant folding*, which removes the `(let ((port '())) ...)` in the final code).

```
(define (user)
   (let ((port '()))
      (let ((port (if (null? port)
                      (current-input-port)
                      (car port))))
         (if (input-port port)
             (read-char/port port)
             (failure)))))
```

Figure 5:

```
(define (user)
   (let ((port '()))
      (let ((port (current-input-port)))
         (read-char/port port))))
```

Figure 6:

The examples of Figures 1 and 4 illustrate the relevance of the analysis we have described. The results are not always so convincing; we give in section 4 the ratio of type tests we can eliminate in real situations.

3 Closures allocation

As mentioned before, 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.

3.1 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 :

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

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

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.*[2]

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

3.2 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*) :
  - 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.
  - 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.
  - 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* ...) :
  - 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.
  - 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.
- (`funcall` *f* ...) or (*f* ...) : 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.[3]

3.3 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 4 is devoted to an analysis of the improved performances obtained by the techniques previously described.

[3]Of course, this raises the problem of execution termination pointed out by G. Rozas. For example, after the analysis, programs written with the fix point operator $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ may stop when they should fall in an infinite loop.

3.3.1 $\mathcal{S}$: First order functions

The example given in Figure 7 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.

```
(define (fibo x)
   (letrec ((fib (lambda (x)
                   (if (< x 2)
                       1
                       (+ (fib (- x 1)) (fib (- x 2)))))))
      (if (number? x)
          (fib x)
          (error "fib" x))))

(fibo 20)
```

Figure 7:

This optimization is very important since like `fib` and `fibo`, many functions satisfy the $\mathcal{S}$ predicate. The measures of section 4 will prove this.

3.3.2 $\mathcal{X}$: Fix point operator

The example of Figure 8 is taken from G.J. Rozas' paper [11]. It has been rewritten only by replacing the `lambda` forms by `labels` forms. One of the consequences of this transformation is that the procedures are named.

```
01: (let ((fac (labels ((y (f)
02:                       (labels ((g (x)
03:                                  (funcall
04:                                     f
05:                                     (labels ((lam-2 ()
06:                                                (funcall x x)))
07:                                       (function lam-2)))))))))
08:                  (y (labels ((lam-0 (fg)
09:                                (labels ((lam-1 (n)
10:                                           (if (= n 0)
11:                                               1
12:                                               (* (funcall
13:                                                     (funcall fg)
14:                                                     (- n 1)) n))))
15:                                   (function lam-1))))
16:                           (function lam-0))))))
17:    (funcall fac 10))
```

Figure 8:

In this program, all functions satisfy the $\mathcal{X}$ predicate, that is, all calls will be direct calls. This is true for function `lam-1`, whose code after analysis is given in Figure 9. The expression (`funcall (funcall fg) (- n 1)`) (lines *12* and *13*) has been replaced by a direct recursive call.

```
(define (lam-1 lam-1-env n)
    (if (= n 0) 1 (* (lam-1 lam-1-env (- n 1)) n)))
```

Figure 9:

3.3.3 $\mathcal{X}$: Currying

The Figure 10 shows an example of a program written in "ML style". In this language, all functions are unary; a natural way to express functions with a higher arity is currying. This example illustrates the improvement made when a function satisfies the $\mathcal{X}$ predicate. 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 as shown in Figure 11. One can see that, in this case, no further allocation is needed.

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

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

(foo 1 2)
```

Figure 10:

```
(define plus
   (lambda (x)
      x))

(define lambda-1
   (lambda (env y)
      (+ env y)))

(define foo (lambda (x y)
               (lambda-1 (plus x) y)))

(foo 1 2)
```

Figure 11:

3.3.4 $\mathcal{T}$: Denotational semantics

Let us illustrate the usefullness of $\mathcal{T}$ by studying a denotational semantics. The figure 12 is a fragment of a Scheme denotational semantics, itself written in Scheme.
Each one of `meaning-...` functions (`meaning-reference`, `meaning-quotation`, `meaning-alternative`, etc.) returns closures with three arguments (`env`, `cont` and `store`). Let M the set of these closures. Each of them can be invoked on all sites '`((meaning ...) ...)`' (lines *02* and *19*). We have :

$$\forall f \in M, \mathcal{T}(f)$$

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. Here is the optimization :
As the *0cfa* has identified that all functions in M satisfy the $\mathcal{T}$ predicate, it is possible, at compilation time, to change the way they are represented. For example, one can decide to give them a unique identifier and to represent them as a pair $< num \times env >$. All '`((meaning ...) ...)`' calls would become an indexed jump to the first projection of these pairs. These first projections could then be seen as "byte-codes" and the '`((meaning ...) ...)`' calls could be seen as "byte-code" interpreters.
In Bigloo, we have adopted a slightly better solution. Rather than allocating pairs $< num \times env >$, we allocate for each function in M a pair of the form $< entry\ point \times env >$. In this way, we avoid indexed jumps as well as code growth. Section 4, will give some figures presenting the gain of this transformation.

4 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)**. Compilations have been interrupted each time before C code generation. We have then measured the size of compiled files, still written in our intermediate language. For all the compilation modes so far described, we have evaluated the number of type checks done, and the number of closures allocations. We have also measured the compilation time with and without the C compiler (GNU-cc [16] always used with the -O compiler option). 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).
Figure 13 presents the programs used to make our measures.

```
01:  ;; eval: form * env * cont * store --> val
02: (define (eval e env cont store) ((meaning e) env cont store))
03:
04:  ;; meaning: form --> env * cont * store --> val
05: (define (meaning e)
06:   (if (atom? e)
07:       (if (symbol? e) (meaning-reference e) (meaning-quotation e))
08:       (case (car e)
09:         ((quote) (meaning-quotation (cadr e)))
11:         ((if) (meaning-alternative (cadr e) (caddr e) (cadddr e)))
10:         ((lambda) (meaning-abstraction (cadr e) (cddr e)))
12:         ((begin) (meaning-sequence (cdr e)))
13:         ...
14:
18: (define (meaning-alternative e1 e2 e3)
19:   (lambda (r k s) ((meaning e1) r (lambda (v s) (if v ((meaning e2) r k s) ((meaning e3) r k s))) s)))
20:
21: (define (meaning-quotation v)
22:   (lambda (r k s) (k v s)))
...
```

Figure 12:

program	author	size	iterations	description
Queens	L. Auguston	97	3	Number of solutions of the queens problem.
Semantics	C. Queinnec	231	5	A pattern matching language semantics [9].
Pp	M. Feeley	524	4	A Scheme pretty printer.
Conform		569	5	A program making heavy use of higher-order procedures.
Peval	M. Feeley	632	3	A simple partial evaluator.
Earley	M. Feeley	661	5	An earley parser implementation.

Figure 13: The benchmarks programs description: **size** is expressed in number of lines and **iterations** is the number of iterations required by the *0cfa* to reach its fix point.

For each of them, we have made several measures which are described in Figure 14.

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 %
size (lines)	9143	8190	3816	53 %
.o size (Kbytes)	144	136	52	61 %
failure	718	626	203	68 %
type error	459	355	132	63 %
proc	94	94	5	95 %
$\mathcal{T}$	-	-	0	-
$\mathcal{X}$	-	-	16	-
$\mathcal{S}$	-	-	73	-
run	73.5 s	61.9 s	24.7 s	60 %
run unsafe	52.2 s	43.4 s	15.8 s	90 %

Conform uses many functions, but few of them require to be allocated. The closure optimization has a great impact on this program. This explains why the gain with the *0cfa* is more important when compiling without dynamic type tests than with dynamic type tests (95 % in **unsafe** mode against 60 % in **safe** mode).

Pp		**O**	**O2**	δ
compil.	6.1 s	5.2 s	35.7 s	-586 %
compil.+cc	56.5 s	48.6 s	59.8 s	-23 %
size (lines)	6555	5771	3018	48 %
.o size (Kbytes)	87	78	30	62 %
failure	471	404	97	76 %
type error	318	251	81	68 %
proc	52	52	14	73 %
$\mathcal{T}$	-	-	5	-
$\mathcal{X}$	-	-	3	-
$\mathcal{S}$	-	-	30	-
run	12.6 s	11.4 s	8.4 s	26 %
run unsafe	12.6 s	9.9 s	6.3 s	36 %

Pp being less functional than the others, weaker results are not surprising. The number of optimized closures is the smallest (73 %). This probably explains why the speed up is also the smallest of our benchmarks programs (26 % and 36 %).

compil.	Time in seconds to produce C code.
compil.+cc	Global compilation time (with linking).
size	Number of lines of the program just before the C code generation (expressed in our intermediate language).
.o size	Size of the file obtained after the C compilation (expressed in Kilo bytes).
failure	Number of times the `failure` form has been introduced during type and arity checking.
type error	Number of `failure` that can be produced due to typing errors.
proc	Number of procedure allocations without any optimizations. In the **O2** column, it represents the number of closures which are not optimized.
$\mathcal{T}$	Number of procedures that have been simplified since they satisfy the $\mathcal{T}$ predicate.
$\mathcal{X}$	Number of procedures eliminated since they satisfy the $\mathcal{X}$ predicate.
$\mathcal{S}$	Number of procedures eliminated since they satisfy the $\mathcal{S}$ predicate.
run	Execution time of the program.
run unsafe	Execution time of the same program compiled with special option where no dynamic type checking had been carried on.

Figure 14: Measure explanations

Earley		**O**	**O2**	δ
compil.	7.4 s	6.8 s	73.9 s	-986 %
compil.+cc	73.3 s	56.5 s	100.5 s	-77 %
size (lines)	10679	7965	3637	54 %
.o size (Kbytes)	120	90	33	63 %
failure	859	547	214	61 %
type error	626	314	118	62 %
proc	75	75	4	95 %
$\mathcal{T}$	-	-	0	-
$\mathcal{X}$	-	-	0	-
$\mathcal{S}$	-	-	71	-
run	44.4 s	41.6 s	10.2 s	75 %
run unsafe	33.7 s	33.2 s	7.0 s	79 %

For this program (**Earley**), the *0cfa* seems to be very time consuming. Compilation time are increased of 986 % without C compilation and 77 % with C compilation. This is due to the dynamic type checks. Here are the compilation times in *unsafe* mode:

Earley	**O**	**O2**	δ
compil.	5.7 s	11.9 s	-108 %
compil.+cc	21.5 s	21.1 s	2 %

Earley 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*.

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 %
size (lines)	3444	2982	1597	46 %
.o size (Kbytes)	49	45	18	60 %
failure	248	210	25	88 %
type error	156	118	25	79 %
proc	54	54	8	85 %
$\mathcal{T}$	-	-	36	-
$\mathcal{X}$	-	-	4	-
$\mathcal{S}$	-	-	8	-
run	38 s	39 s	15.2 s	61 %
run unsafe	32.5 s	32.6 s	15.4 s	53 %

Semantics being a denotational semantics, the optimization described in Section 3.3.4 is applied. This explains why there is so many functions satisfying the $\mathcal{T}$ predicate in this example.

Peval		**O**	**O2**	δ
compil.	7.6 s	6.4 s	35 s	-446 %
compil.+cc	86.9 s	62.4 s	72.7 s	-16 %
size (lines)	9458	7550	4789	37 %
.o size (Kbytes)	144	120	54	55 %
failure	724	535	218	59 %
type error	566	377	213	43 %
proc	63	63	15	76 %
$\mathcal{T}$	-	-	8	-
$\mathcal{X}$	-	-	0	-
$\mathcal{S}$	-	-	40	-
run	17.7 s	17.4 s	10.7 s	38 %
run unsafe	13.4 s	13.4 s	8.1 s	39 %

The type investigation seems to fail on **Peval**. It may be due to the numerous side effects occuring in it. The only gain of the *0cfa* on this program is the closure optimization since the gain in both safe and unsafe mode are the same.

Queens		**O**	**O2**	δ
compil.	2.1 s	2.1 s	2.9 s	-38 %
compil.+cc	10.7 s	8.7 s	6.6 s	24 %
size (lines)	737	586	189	69 %
.o size (Kbytes)	13	12	4	67 %
failure	66	50	7	86 %
type error	47	31	7	77 %
proc	12	12	0	100 %
$\mathcal{T}$	-	-	0	-
$\mathcal{X}$	-	-	3	-
$\mathcal{S}$	-	-	9	-
run	31.3 s	29.5 s	12.0 s	59 %
run unsafe	24.8 s	24.5 s	10.7 s	56 %

Queens has been originally written in ML. The automatic currying creates closures which have been eliminated by the *0cfa*.

Several comments should be made on these results :

- 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 compilations have been processed with the flags: "`-On -Ob -Og -Ot -cc gcc -O`". This means that the compilation is `unsafe` and that fixnum arithmetic is used. For **compil.+cc**, the compilation is stopped after the C compilation.

scc	Queens	Semantics	Pp
compil.+cc	13.8 s	29.9 s	49 s
run unsafe	22.3 s	30.6 s	18.3 s

scc	Conform	Peval	Earley
compil.+cc	55 s	80.8 s	111.5 s
run unsafe	35.4 s	13.8 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 (**queens** 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 use the analysis in "real world" and we never found programs that require an polynomial number of iterations.

- The numbers of type checks and closures creations we give are static. Although they may not be ideal measures, they are of some help.

- When looking at the execution times in the so-called "`unsafe`" mode, one can have an idea of the efficiency of the procedure allocation optimizations, described in Section 3.1. Indeed, in this mode, Bigloo performs no dynamic type checks which could slow the execution. The main difference between modes **-O** and **-O2** is that, in the second case, the described optimizations used. The average improvement in the execution times of the programs is about 61 %.

- Since the "`unsafe`" mode execution time has been measured with no dynamic type checking, it corresponds to what a ML compiler would gain by using our optimizations.

These time figures show the actual performances gains that can be obtained with a *0cfa* analysis. In the examples given, 60 % of execution time is gained with a small increase in compilation time (20 %).

5 Related works

Three kind of works 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[13, 15]. 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 [7], 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. Our type analysis is close to his *"type recovery"*, but the other 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 dont 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 [11], 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 do not have 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 [10], 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 [8] and N. Séniak's one [12] 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 [4]. 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.
People worked on reducing type checking in dynamic typed language :

- Our type investigation differs from of the W algorithm of Damas and Milner [3]. Our goal is to compute only approximation of types rather than perform a total type computation. In particular a variable can be of several types. The W algorithm types functions *without* knowing where they are applied. Our algorithm types functions *only* knowing how they are used (knowing the types of the actual arguments). In general, our type investigation is not as precise as the W algorithm, but sometimes we get results which are impossible to obtain with W. For instance, our algorithm can type generic arithmetic (i.e. our algorithm recognizes fixnum and flonum).

  Another interesting comparison between W and *0cfa* is that both of them have high complexities but, in practice, they behave as if they were linear !

- Fritz Henglein presents an optimization which allows to remove useless tests [5]. He does not use control analysis but seems to eliminate more tests than our analysis do: the author claims to remove from 60 to 95 % of the tests. His analysis is entirely devoted to type checking, not to closure allocation improvement.

- Cartwright, Wright and Fagan in [2, 21] and Thatte in [19] show systems performing partial static type inference. Programs which are not correctly typed can still be executed but with dynamic type checking inserted. The goal of these papers is not to show how one can reduce the number of type checking but to investigate a new way of computing types with partial algorithms.

[4] Kranz is a bit more precise since closures requiring environment allocation are themself divided into two parts, the ones which are heap allocated and the ones which are stack allocated. Our compiler also performs stack allocations but since this optimization does not concern only closures, we do not address it here.

Conclusion

Using an already known analysis (the *0cfa*), this paper describes some new optimizations. The first optimization concerns closure allocations. For several "real programs" we measure that the new closures optimization removes 87 % of allocations. Applying the analysis to type investigation allows to remove 65 % of dynamic type checking. These different improvements represent a gain of 53 % on execution times (a speed up by a factor of two) with dynamic type checks, 59 % on execution times without any dynamic type checks and 61 % on the sizes of the object files produced (more than two times smaller) [5]. All modern functional languages such as ML and Scheme are in the scope of the presented optimizations.

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] R. Cartwright and M. Fagan. Soft Typing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 278–292, Toronto, Ontario, Canada, June 1991.

[3] L. Damas and R. Milner. Principle Type Inference for Functional Programs (extended abstract). In *9th ACM Symposium on Principle of Programming Langugages*, pages 207–212, 1982.

[4] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML.* MIT-press, 1990.

[5] F. Henglein. Global Tagging Optimization by Type Inference. In *ACM Conference on Lisp and Functional Programming*, 1992.

[6] 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.

[7] 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.

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

[5] Program performances depend on the size of cache memory. Reducing the object size allows them to fit more easily in this memory, so this gain in size could become, with cache size expansion, very important.

[9] C. Queinnec. Compilation of Non-Linear, Second Order Pattern on S-Expressions. In P. Deransart and J. Maluszyński, editors, *PLILP*, number 245 in LNCS, pages 340–357, August 1990.

[10] G.J. Rozas. Liar, an Algol like compiler for Scheme. S.b. thesis, Massachusetts Institute of Technology, Cambridge, Mass., Jan 1984.

[11] 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.

[12] 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.

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

[14] 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.

[15] O. Shivers. The Semantics of Scheme Control-Flow Analysis. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale University, New Haven, Conn., June 1991.

[16] R.M. Stallman. Using and Porting GNU CC. Technical report, Free Software Foundation, Inc., April 1989.

[17] IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[18] G.L. Steele. *COMMON LISP (The language).* Digital Press (DEC), Burlington MA (USA), 2nd edition edition, 1984.

[19] S. Thatte. Quasi-static Typing. In *Proceedings of the ACM Symposium of Principles of Programming Languages*, pages 367–381, 1990.

[20] P. Weis. The CAML Reference manual, Version 2.6.1. Technical Report 121, INRIA-Rocquencourt, 1990.

[21] A. Wright and R. Cartwright. A Practical Soft Type System for Scheme. In *Conference Record of the ACM Conference on Lisp and Functional Programming*, Orlando, Florida, US, June 1994.