Abstract Compilation: a New Implementation Paradigm for Static Analysis

Dominique Boucher and Marc Feeley

Département d'informatique et de recherche opérationnelle (IRO)
Université de Montréal
C.P. 6128, succ. centre-ville, Montréal, Québec, Canada H3C 3J7
E-mail: {boucherd,feeley}@iro.umontreal.ca

Abstract. For large programs, static analysis can be one of the most time-consuming phases of the whole compilation process. We propose a new paradigm for the implementation of static analyses that is inspired by partial evaluation techniques. Our paradigm does not reduce the complexity of these analyses, but it allows an efficient implementation. We illustrate this paradigm by its application to the problem of control flow analysis of functional programs. We show that the analysis can be sped up by a factor of 2 over the usual abstract interpretation method.

Keywords: Abstract interpretation, static analysis, partial evaluation, compilation, control flow analysis.

1 Introduction

As the trend in designing higher level languages continues, it is increasingly becoming important to design compilation techniques to implement them efficiently. Optimizing compilers for such languages must typically perform a variety of static analyses to apply their optimizations. Most of these analyses are very time-consuming. It is therefore essential to perform them as efficiently as possible. Speed of analysis is the issue addressed in this paper.

For the class of first-order imperative languages, several techniques for static analysis have been designed and are now well established [1, 6]. Static analysis of higher-order functional languages is more difficult because the control flow graph (call graph) is not known at compile-time. Nevertheless, several kinds of analyses have been designed [2, 7, 15] and some have been successfully integrated in real compilers [13, 15].

1.1 A New Paradigm

A popular approach for implementing static analyses is non-standard interpretation. Even traditional data-flow analysis can be viewed as an interpretation layer, the flow graph being the abstract program to be "executed". The more recent analyses, devised in the abstract interpretation framework, are implemented as true interpreters (for example, see [13]).

But interpretation is costly because it adds a layer of abstraction to the analysis process. We propose to go one step further and perform what we call *abstract compilation.* This new paradigm is based on a simple idea: instead of interpreting (in some sense) the source program, we compile it into a program which computes the desired analysis when it is executed.

More formally, suppose we want to compute some static analysis $\mathcal{S}$. $\mathcal{S}$ can be viewed as a function of two arguments. The first is the program p we want to analyze. The second is an initial abstract environment σ_0 that depends on the analysis to be performed. The result of the analysis, $\mathcal{S}(p, \sigma_0)$, is an abstract environment which contains the desired information. The *abstract compilation* of p, $\mathcal{C}(p)$, would then be a function of one argument such that:

$$\mathcal{C}(p)(\sigma_0) = \mathcal{S}(p, \sigma_0).$$

Essentially, $\mathcal{C}$ is nothing more than a curried version of $\mathcal{S}$. But, as we will see, only the "real computational part" of $\mathcal{S}(p, \sigma_0)$ can be kept in the code of $\mathcal{C}(p)$. For instance, there is no traversal of the abstract syntax tree of p in $\mathcal{C}(p)$. This way, all the overhead of interpretation is eliminated. The abstract compilation process is really a kind of *ad hoc* partial evaluation. In fact, the abstract compiler $\mathcal{C}$ can be seen as a partial evaluator specialized for the static analysis $\mathcal{S}$.

Devising $\mathcal{C}$ directly made us aware of several interesting optimizations that can be performed to further speed up the analysis. Our results show that, using the technique of abstract compilation, the analysis can be sped up by over a factor of 2.

1.2 Overview

In this paper, we demonstrate our paradigm by showing how the control flow analysis (**cfa**) of higher-order functional programs can be compiled. We first describe the analysis and the language we want to analyze. Then, two different compilation strategies are presented. The first compiles the analysis into a textual program which is then executed using a general interpretation procedure. The second shows how we can use closures to produce a more efficient analysis program. Finally, we compare our results with more conventional implementations of the **cfa**.

Throughout the text, the Scheme programming language is used, mainly for our examples. But note that the compilation algorithms presented here do not rely on any particular language although they use some Lisp-like notation.

2 Control Flow Analysis

In higher-order functional languages, functions are first-class objects, i.e. they can be passed as arguments to other functions, returned as the result of functions, stored in data structures, and so on. It is thus more difficult to predict at compile-time the behavior of programs making heavy use of higher-order functions. One way to do so is the control flow analysis (**cfa**), of which there exists several variants [2, 8, 12]. The *0cfa* [15] computes, for each call site (α ...) of a program, the set of functions that could be bound to α at runtime.

To appreciate the usefulness of *0cfa*, consider the Scheme program of Fig. 1. The *0cfa* would find that in the `map` function, the only function that can be bound to `f` results from the evaluation of `(lambda (y) (+ y x))` (the result of applying `adder` to the value `1` or `2`). Knowing this, the compiler can optimize the runtime representation of the closure and the call to `f`. Rather than being a record with a code pointer and environment, the "closure" could simply be the value of `x` (`1` or `2`) and the call to `f` can be replaced by a jump to the body of the `lambda` expression with `x` as an argument.

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

(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))))

(let ((lst '(1 2 3 4 5 6)))
  (append (map (adder 1) lst)
          (map (adder 2) lst)))
```

Fig. 1. A small program

$\Pi \in$ Prog
$C \in$ Call
$L \in$ Lam
$F \in$ Fun
$A \in$ Arg
$V \in$ Var
$K \in$ Const
$P \in$ Prim (primitive functions: `if`, `+`, etc.)

$\Pi ::=$ C
$C ::=$ `(`$F\ A_1\ \ldots\ A_n$`)`
  | `(letrec ((`$V_1\ L_1$`)` $\cdots$ `(`$V_n\ L_n$`))` C`)`
$L ::=$ `(`λ `(`$V_1\ \ldots\ V_n$`)` C`)`
$F ::= L \mid V \mid P$
$A ::= K \mid V \mid L$

Fig. 2. Abstract syntax

We will now see how we can compute this **cfa**. Figure 2 describes the abstract syntax of our source language. It is a continuation passing style (*CPS*) λ-language. We assume that all programs are fully alpha-converted. We use CPS to simplify the analysis. Special forms like `if` can then be considered as primitive functions and all intermediate results are given names. Since only `lambda`-expressions, variables, and primitives can appear in the operator position of a call site, the *0cfa* problem is equivalent to the one of finding, for each variable v occurring in the program, the set of functions that can be bound to v. Our use of CPS carries no loss of generality since any non-CPS program can be easily converted to an equivalent CPS program.

Figure 3 gives the functionalities of the abstract interpretation algorithm[1] for *0cfa* shown in Fig. 4. The following terminology is assumed. First, $l \downarrow_{\text{formals}_i}$ stands for the ith formal parameter of procedure l. Similarly, $l \downarrow_{\text{body}}$ is the body of procedure l (a call site). The abstract environments are functions from syntactic domain *Var* and deliver results in domain 2^{Lam}. The empty environment is denoted σ_0 ($\sigma_0(v) = \emptyset$ for all v) and $[v \mapsto S]$ stands for the environment σ such that $\sigma(x)$ is S if $x = v$ and $\emptyset$ otherwise. Finally, environments can be joined using the $\sqcup$ operator, defined by $(\sigma \sqcup \sigma')(v) = \sigma(v) \cup \sigma'(v)$.

$$
\begin{aligned}
\text{0cfa-program} &: \text{Prog} \times \widehat{\text{Env}} \to \widehat{\text{Env}} \\
\text{0cfa-call} &: \text{Call} \times \widehat{\text{Env}} \to \widehat{\text{Env}} \\
\text{0cfa-app} &: \text{Fun} \times \text{Arg}^* \times \widehat{\text{Env}} \to \widehat{\text{Env}} \\
\text{0cfa-abstract-app} &: 2^{\text{Lam}} \times \text{Arg}^* \times \widehat{\text{Env}} \to \widehat{\text{Env}} \\
\text{0cfa-args} &: \text{Arg}^* \times \widehat{\text{Env}} \to \widehat{\text{Env}} \\
\text{0cfa-prim} &: \text{Prim} \times \text{Arg}^* \times \widehat{\text{Env}} \to \widehat{\text{Env}} \\
\text{lookup} &: \text{Arg} \times \widehat{\text{Env}} \to \widehat{\text{Env}}
\end{aligned}
$$

$$
\widehat{\text{Env}} = \text{Var} \to 2^{\text{Lam}}
$$

Fig. 3. Functionalities

The *0cfa* of a program p is computed by finding an environment σ such that $\sigma =$ *0cfa-program*(p, σ). This can be done iteratively by successive approximation, starting with σ_0. It can easily be shown that this process eventually terminates. The approximations $\sigma_0, \sigma_1, \ldots$ form an ascending chain (taking $\sigma \sqsubseteq \sigma'$ to mean $\sigma(v) \subseteq \sigma'(v)$ for all v), since we only add elements to the environment. Also, since every program is finite, $\sigma(v)$ must be finite for all v. Thus our algorithm will find σ in a finite number of steps.

This is the usual way the *0cfa* is implemented. For example, [13] describes the analysis performed in the Bigloo compiler [14]. It is essentially the same as the one we have presented. It is also very close to the one presented by Shivers

[1] For the sake of simplicity, we do not include any error-detection mechanism to the *0cfa*. We thus assume that all programs are syntactically valid.

$$\text{0cfa-program}(p,\sigma) = \text{0cfa-call}(p,\sigma)$$

$$
\begin{array}{l}
\text{0cfa-call}([\![\texttt{(}f\ \ a_1 \ldots a_n\texttt{)}]\!],\sigma) = \\
\qquad \text{0cfa-app}(f, \langle a_1,\ldots,a_n\rangle,\ \text{0cfa-args}(\langle a_1,\ldots,a_n\rangle,\sigma)) \\
\text{0cfa-call}([\![\texttt{(letrec ((}v_1\ \ l_1\texttt{)}\ \cdots\ \texttt{(}v_n\ \ l_n\texttt{))}\ c\texttt{)}]\!],\sigma) = \\
\qquad \textbf{let}\ \ \sigma' = \sigma \sqcup [v_1 \mapsto \{l_1\}] \sqcup \cdots \sqcup [v_n \mapsto \{l_n\}] \\
\qquad\qquad \sigma'' = \text{0cfa-args}(\langle l_1,\ldots,l_n\rangle,\sigma'') \\
\qquad\quad \textbf{in}\ \text{0cfa-call}(c,\sigma'')
\end{array}
$$

$$
\begin{array}{l}
\text{0cfa-app}(f, \langle a_1,\ldots,a_n\rangle,\sigma) = \\
\qquad \textbf{cond} \\
\qquad\quad \mathit{isVar}(f): \\
\qquad\qquad \text{0cfa-abstract-app}(\sigma(f), \langle a_1,\ldots,a_n\rangle,\sigma) \\
\qquad\quad \mathit{isPrim}(f): \\
\qquad\qquad \text{0cfa-prim}(f, \langle a_1,\ldots,a_n\rangle,\sigma) \\
\qquad\quad \mathit{isLam}(f): \\
\qquad\qquad \textbf{let}\ \ \sigma' = \sigma \sqcup\ [f\downarrow_{\text{formals}_1} \mapsto \text{lookup}(a_1,\sigma)] \sqcup \cdots \\
\qquad\qquad\qquad\qquad\quad \cdots \sqcup [f\downarrow_{\text{formals}_n} \mapsto \text{lookup}(a_n,\sigma)] \\
\qquad\qquad\quad \textbf{in}\ \text{0cfa-call}(f\downarrow_{\text{body}},\sigma')
\end{array}
$$

$$
\begin{array}{l}
\text{0cfa-abstract-app}(\emptyset, \langle a_1,\ldots,a_n\rangle,\sigma) = \sigma \\
\text{0cfa-abstract-app}(S, \langle a_1,\ldots,a_n\rangle,\sigma) = \\
\qquad \textbf{let}\ \ l = \text{some member of S} \\
\qquad\qquad \sigma' = \sigma \sqcup\ [l\downarrow_{\text{formals}_1} \mapsto \text{lookup}(a_1,\sigma)] \sqcup \cdots \\
\qquad\qquad\qquad\quad \cdots \sqcup [l\downarrow_{\text{formals}_n} \mapsto \text{lookup}(a_n,\sigma)] \\
\qquad\quad \textbf{in}\ \text{0cfa-abstract-app}(S - \{l\}, \langle a_1,\ldots,a_n\rangle,\sigma')
\end{array}
$$

$$
\begin{array}{l}
\text{0cfa-args}(\langle\rangle,\sigma) = \sigma \\
\text{0cfa-args}(\langle a_1,\ldots,a_n\rangle,\sigma) = \\
\qquad \textbf{let}\ \ \sigma' = \textbf{if}\ \mathit{isLam}(a_1) \\
\qquad\qquad\qquad\quad \textbf{then}\ \sigma \sqcup\ \text{0cfa-call}(a_1\downarrow_{\text{body}},\sigma) \\
\qquad\qquad\qquad\quad \textbf{else}\ \sigma \\
\qquad\quad \textbf{in}\ \text{0cfa-args}(\langle a_2,\ldots,a_n\rangle,\sigma')
\end{array}
$$

$$
\begin{array}{l}
\text{0cfa-prim}([\![\texttt{+}]\!], \langle a_1,\ldots,a_3\rangle,\sigma) = \text{0cfa-args}(\langle a_1,\ldots,a_3\rangle,\sigma) \\
\text{0cfa-prim}([\![\texttt{if}]\!], \langle a_1,\ldots,a_3\rangle,\sigma) = \text{0cfa-args}(\langle a_1,\ldots,a_3\rangle,\sigma) \\
\ldots
\end{array}
$$

$$
\begin{array}{l}
\text{lookup}(e,\sigma) = \\
\qquad \textbf{cond} \\
\qquad\quad \mathit{isConst}(e): \emptyset \\
\qquad\quad \mathit{isVar}(e): \sigma(e) \\
\qquad\quad \mathit{isLam}(e): \{e\}
\end{array}
$$

Fig. 4. 0cfa abstract interpretation algorithm

in [15]. We will now show how we can compile the analysis, by extending the interpretation algorithm.

3 A First Abstract Compiler

When many iterations are needed for the algorithm to reach a fixed point, a lot of work is done which does not have a direct impact on the result of the analysis. The reason for this is that each iteration requires a traversal of the entire syntax tree, examining each node to see if it is an application, an abstraction, etc. This is the interpretation overhead. When we consider the interpretation algorithm of Fig. 4, we notice that only three functions can actually influence the result of the analysis: *0cfa-call* when applied to a `letrec` special form, *0cfa-app* when f is a λ-expression, and *0cfa-abstract-app*.

What we are interested in is a way to remember only those computations which affect the final result of the analysis. Consider the sample CPS program of Fig. 5, where each λ-expression has been numbered from 1 to 7 (we will later refer to these expressions as λ_1 to λ_7). It defines a currified version of `apply`, a function such that `((apply f) x)` = `(f x)`. The program then computes `((apply (apply (λ (x) (+ x 1)))) 2)`.

```
((λ1 (apply k1)
    (apply (λ2 (x1 k2)
              (+ x1 1 k2))
           (λ4 (t2)
              (apply t2 (λ5 (t3) (t3 2 k1))))))

 (λ6 (f k3)
    (k3 (λ7 (x2 k4)
           (f x2 k4))))

 tl-cont)
```

Fig. 5. A small CPS program.

By carefully examining the program, we can determine the particular call sites where the control flow information will be propagated. The call (λ_1 λ_6 `tl-cont`) will add λ_6 to σ(`apply`) and `tl-cont` to σ(`k1`). This is the simplest case. But consider an inner call site, `(t3 2 k1)`, in λ_5. The analysis will take each $\lambda_i \in \sigma$(`t3`) and will add σ(`k1`) to $\sigma(\lambda_i \downarrow_{\text{formals}_2})$. In constrast, the call `(+ x1 1 k2)` adds no information and has no impact on the final result.

Note that only the call sites where the information is propagated are useful for the computation of the analysis. One way to implement the analysis would be to first traverse the syntax tree and store the useful call sites in some data structure and then traverse it at each iteration. But again, there still remains an interpretation layer, namely the computations needed to traverse the data structure.

Compilation can overcome this layer of interpretation by replacing the data structure representing the program to analyse by the control structure of another program (the "analysis program"). The only "interpretation" that remains is at the processor level but since this is unavoidable we will not count it. Figure 6 shows a first compilation algorithm for *0cfa*. We use a Scheme-like notation for the produced code. The function *comp-program* takes as input a program p and produces p', the analysis program[2] in source form. When p' is run, it performs the analysis by finding an abstract environment such that $\sigma = p'(\sigma)$, by the technique of successive approximation.

To see how it works, consider the following program:

```
((λ1 (f c1)
    ((λ2 (x c2)
        (f x c2))
     2
     c1))
 (λ3 (y c3)
    (+ y 1 c3))
 tl-cont)
```

Once compiled, we get the following analysis program:

```
    (λ (σ)
      ((λ (σ)
         ((λ (σ)
            ((λ (σ)
               ((λ (σ)
(1)               ((λ (σ) (0cfa-abstract-app σ(f) σ x c2))
(2)             σ ⊔ [x ↦ (lookup 2 σ)] ⊔ [c2 ↦ (lookup c1 σ)]))
             ((λ (σ) σ)
              σ)))
(3)       σ ⊔ [f ↦ (lookup λ3 σ)] ⊔ [c1 ↦ (lookup tl-cont σ)]))
         ((λ (σ) σ)
          σ)))
      ((λ (σ)
         ((λ (σ)
            ((λ (σ) σ)
             ((λ (σ) σ)
              σ)))
          ((λ (σ) σ)
           σ)))
       σ)))
```

---

2 We assume that *0cfa-abstract-app* and *lookup* can be "linked" in some way with the resulting program.

$$
\begin{array}{l}
\text{comp-program}(p) = \text{comp-call}(p) \\
\\
\text{comp-call}([\![\texttt{(}f \;\; a_1 \ldots a_n\texttt{)}]\!]) = \\
\quad \textbf{let } C_1 = \text{comp-args}(\langle a_1, \ldots, a_n \rangle) \\
\qquad\quad C_2 = \text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) \\
\qquad \textbf{in } [\![\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \;\; \texttt{(}C_2 \;\; \texttt{(}C_1 \;\; \sigma\texttt{)))}]\!] \\
\\
\text{comp-call}([\![\texttt{(letrec ((}v_1 \;\; l_1\texttt{)} \;\; \cdots \;\; \texttt{(}v_n \;\; l_n\texttt{))} \;\; c\texttt{)}]\!]) = \\
\quad \textbf{let } C_1 = \text{comp-call}(c) \\
\qquad\quad C_2 = \text{comp-args}(\langle l_1, \ldots, l_2 \rangle) \\
\qquad \textbf{in } [\![\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \;\; \texttt{(}C_1 \;\; \texttt{(}C_2 \;\; \sigma \sqcup [v_1 \mapsto \{l_1\}] \sqcup \cdots \sqcup [v_n \mapsto \{l_n\}]\texttt{)))}]\!] \\
\\
\text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{cond} \\
\qquad \textit{isVar}(f) : \\
\qquad\quad [\![\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \\
\qquad\qquad \texttt{(0cfa-abstract-app (}\sigma \;\; f\texttt{)} \;\; \sigma \;\; a_1 \;\; \ldots \;\; a_n\texttt{))}]\!] \\
\qquad \textit{isPrim}(f) : \\
\qquad\quad \text{0cfa-prim}(f, \langle a_1, \ldots, a_n \rangle) \\
\qquad \textit{isLam}(f) : \\
\qquad\quad \textbf{let } C = \text{comp-call}(f \downarrow_{\text{body}}) \\
\qquad\qquad \textbf{in } [\![\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \\
\qquad\qquad\qquad \texttt{(}C \;\; \sigma \sqcup [f \downarrow_{\text{formals}_i} \mapsto \texttt{(lookup } a_i \;\; \sigma\texttt{)}]\texttt{))}]\!] \\
\\
\text{comp-args}(\langle\rangle) = [\![\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \;\; \sigma\texttt{)}]\!] \\
\text{comp-args}(\langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{if } \textit{isLam}(a_1) \\
\qquad \textbf{then let } C_1 = \text{comp-call}(a_1 \downarrow_{\text{body}}) \\
\qquad\qquad\qquad C_2 = \text{comp-args}(\langle a_2, \ldots, a_n \rangle) \\
\qquad\qquad\quad \textbf{in } [\![\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \;\; \texttt{(}C_2 \;\; \texttt{(}C_1 \;\; \sigma\texttt{)))}]\!] \\
\qquad \textbf{else } \text{comp-args}(\langle a_2, \ldots, a_n \rangle) \\
\\
\text{comp-prim}([\![\texttt{+}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\text{comp-prim}([\![\texttt{if}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\ldots
\end{array}
$$

Fig. 6. 0cfa compilation algorithm

It is not hard to see that only lines *(1)*, *(2)*, and *(3)* will contribute to the abstract environment. We can also see that there are still a number of useless computations done by this analysis program. Two simple optimizations can further reduce the number of computations performed at each iteration.

We can first eliminate all the calls to the identity function $\texttt{(}\lambda \;\; \texttt{(}\sigma\texttt{)} \;\; \sigma\texttt{)}$ by performing η-reductions. This can be done at low cost by adding additional tests to the compilation process. For example, assuming that *Id-Funct?* is true if its argument is the code of the identity function, the *comp-call* function becomes:

$$
\begin{array}{l}
\text{comp-call}(\llbracket (f\ \ a_1 \ldots a_n) \rrbracket) = \\
\quad \textbf{let}\ C_1 = \text{comp-args}(\langle a_1, \ldots, a_n \rangle) \\
\qquad\ \ C_2 = \text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) \\
\quad\ \ \textbf{in}\ \textbf{if}\ \text{Id-funct?}(C_1) \\
\qquad\qquad \textbf{then}\ C_2 \\
\qquad\qquad \textbf{else}\ \textbf{if}\ \text{Id-funct?}(C_2) \\
\qquad\qquad\qquad \textbf{then}\ C_1 \\
\qquad\qquad\qquad \textbf{else}\ \llbracket (\lambda\ \ (\sigma)\ \ (C_2\ \ (C_1\ \ \sigma))) \rrbracket
\end{array}
$$

The second optimization comes from the behavior of `lookup`. When applied to a constant, it returns the empty set; when applied to a λ-expression, it returns the set containing only this expression. This leads to the following optimization. First, we can eliminate all the contributions of the form $[v \mapsto$ `(lookup` c σ`)`$]$, where v is a variable and c is constant. Also, we can remove the environments of the form $[v \mapsto \{\lambda_k\}]$ and add them to the initial environment. This saves one iteration, but more importantly, it simplifies the lookup mechanism and makes each iteration faster.

When these two optimizations are added to the compilation algorithm, the compiled code for the previous example now becomes

```
(λ (σ)
  ((λ (σ)
     ((λ (σ)
        ((λ (σ)
           ((λ (σ) (0cfa-abstract-app σ(f) σ x c2))
         σ ⊔ [c2 ↦ (lookup c1 σ)]))
      σ))
   σ ⊔ [c1 ↦ (lookup tl-cont σ)])))
```

Starting with $\sigma'_0 = [\mathtt{f} \mapsto \{\lambda_3\}]$ (as computed by the second optimization), we can find that $\sigma_1 = [\mathtt{f} \mapsto \{\lambda_3\}]$ is a fixed point for this function in only one iteration.

This solution is not entirely satisfactory. The layer of abstraction is no longer present in the resulting code but the program must be executed in some way, thus requiring interpretation at another level. If, for example, we use a builtin interpretation procedure, like Scheme's `eval`, our experimentations reveal that it remains much more efficient to compute the analysis by means of abstract interpretation. But it is possible to do better.

4 Representing the Compiled Analysis with Closures

Many functional programming languages allow the user to create new functions via λ-expressions. When these expressions are evaluated, they return a **closure**, i.e. a function that remembers the current environment.

We will use closures here to overcome the interpretation overhead of the analysis program. The idea is to represent a compiled expression with a closure. When this closure is applied, it performs the analysis of the given expression. We will thus replace the "code generation" by a "closure generation" (as in the work of Feeley and Lapalme [5]). This leads to the compilation algorithm of Fig. 7 (without the optimizations discussed above).

$$
\begin{array}{l}
\text{comp-program}(p) = \text{comp-call}(p) \\[1em]
\text{comp-call}([\![\texttt{(}f\ \ a_1 \ldots a_n\texttt{)}]\!]) = \\
\quad \textbf{let}\ \ C_1 = \text{comp-args}(\langle a_1, \ldots, a_n \rangle) \\
\qquad\quad C_2 = \text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) \\
\qquad \textbf{in}\ \lambda\sigma.C_2(C_1(\sigma)) \\[1em]
\text{comp-call}([\![\texttt{(letrec ((}v_1\ \ l_1\texttt{)}\ \cdots\ \texttt{(}v_n\ \ l_n\texttt{))}\ \ c\texttt{)}]\!]) = \\
\quad \textbf{let}\ \ C_1 = \text{comp-call}(c) \\
\qquad\quad C_2 = \text{comp-args}(\langle l_1, \ldots, l_n \rangle) \\
\qquad \textbf{in}\ \lambda\sigma.C_1(C_2(\sigma \sqcup [v_1 \mapsto \{l_1\}] \sqcup \cdots \sqcup [v_n \mapsto \{l_n\}])) \\[1em]
\text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{cond} \\
\qquad \mathit{isVar}(f): \\
\qquad\quad \lambda\sigma.\text{0cfa-abstract-app}(\sigma(f), \sigma, \langle a_1, \ldots, a_n \rangle) \\
\qquad \mathit{isPrim}(f): \\
\qquad\quad \text{0cfa-prim}(f, \langle a_1, \ldots, a_n \rangle) \\
\qquad \mathit{isLam}(f): \\
\qquad\quad \textbf{let}\ \ C = \text{comp-call}(f \downarrow_{\text{body}}) \\
\qquad\qquad \textbf{in}\ \lambda\sigma.C(\sigma \sqcup [f \downarrow_{\text{formals}_i} \mapsto \text{lookup}(a_i, \sigma)]) \\[1em]
\text{comp-args}(\langle\rangle) = \lambda\sigma.\sigma \\
\text{comp-args}(\langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{if}\ \mathit{isLam}(a_1) \\
\qquad \textbf{then}\ \ \textbf{let}\ \ C_1 = \text{comp-call}(a_1 \downarrow_{\text{body}}) \\
\qquad\qquad\qquad\quad C_2 = \text{comp-args}(\langle a_2, \ldots, a_n \rangle) \\
\qquad\qquad\qquad \textbf{in}\ \lambda\sigma.C_2(C_1(\sigma)) \\
\qquad \textbf{else}\ \text{comp-args}(\langle a_2, \ldots, a_n \rangle) \\[1em]
\text{comp-prim}([\![\texttt{+}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\text{comp-prim}([\![\texttt{if}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\ldots
\end{array}
$$

Fig. 7. 0cfa compilation algorithm using closures

Comp-app would thus be implemented as:

```
(define (comp-app f args)
  (cond
   ((var? f)
    (lambda (env)
      (0cfa-abstract-app (env f) env args)))
   ...
   ))
```

It may seem that this new compilation scheme is not very different from the previous one; the main difference being that the generated code is no longer textual. This change of representation has two main advantages.

First, there is no longer a need for an interpretation procedure like `eval`. Any language that provides closures can be used to implement the abstract compiler. Secondly, and more importantly, both the abstract compiler and the analysis program run much faster because all the $\lambda\sigma.E$ expressions are also compiled (we assume that the abstract compiler is itself compiled). Only closures are created in the process of abstract compilation.

5 Results

We have implemented the *0cfa* using the abstract interpretation algorithm and the compilation algorithm using closures for code generation. Our implementations handle a larger subset of Scheme than the one presented here. Imperative constructs such as `set!`, `set-car!`, `set-cdr!`, etc., are treated.

Our implementations also handle the case of functions "escaping" to memory. By this we mean functions which are stored in data-structures and that could be later fetched and applied. In order to handle this case conservatively, we introduce a special variable, $\mathcal{ESC}$, that abstracts the memory. For example, if the program to analyze contains the call `(cons x y k)`, all the λ-expressions that can be bound to `x` and `y` at runtime are added to $\sigma(\mathcal{ESC})$. Conversely, a call `(car x (lambda (z)` E`))` will cause $\sigma(\mathcal{ESC})$ to be added to $\sigma($`z`$)$. Clearly, this approximation is very coarse, but conservative and easy to implement.

The front-end is the same for the three implementations. It performs the following operations:

1. It reads the Scheme program to analyze.
2. It performs a certain number of syntactic expansions to express the program using a minimum set of constructs (for example, `let` and `let*` special forms are expressed using only the `lambda` and `set!` special forms).
3. It CPS-converts the program.
4. It labels the λ-expressions and α-converts the program. An abstract syntax tree (AST) results from this last operation.

The *0cfa* is then computed directly from the AST.

The implementations have both been written in Scheme and they have been compiled with the Gambit-C compiler (which generates C code) on a DEC Alpha. Set operations have been implemented in C for efficiency reasons. Several representations for sets have been considered. A list representation was too costly and bit vectors consumed too much memory (typical sets contain very few elements and are very sparse). The representation we adopted consists of vectors in which the elements are sorted. Each set union operation allocates a new vector in which the elements are merged while being copied.

We ran the *0cfa* over the following set of programs:

`conform`	A program that manipulates lattices and partial orders.
`earley`	A parser generator for context-free languages based on Earley's algorithm.
`interp`	A small interpreter implementing call-by-need semantics.
`lambda`	A λ-calculus interpreter.
`lex`	A lexical analyzer generator.
`link`	The application linker for the Gambit-C compiler.
`ll1`	An $LL(1)$ parser generator.
`peval`	A small Scheme partial evaluator.
`source`	A parser for Scheme.

Figure 8 gives the execution times for both implementations on a 160MB DEC AXP3000 (a DEC Alpha microprocessor under OSF/1). The times are all given in seconds. The first column gives the number of lines in the program. The second column gives the number of iterations needed to reach the fixed point and the third column gives the average number of elements of $\sigma(v)$, where σ is the result of the analysis and v ranges over all the variables of the program. The interp column gives the execution time required by the abstract interpreter to perform the analysis. The closure column gives the time for the analysis programs generated by the second compilation algorithm, including all optimizations discussed. The numbers in parentheses give the speedup relative to the times given in the interp column. The last column (gen+closure) gives the time needed to generate and execute the analysis programs and to execute them. The numbers in parentheses give the speedup over interpretation.

We can see that the compilation process can speed up the analysis by a factor varying between 3 and 5 for most of these programs[3] The only exception is `interp`. We can observe that the average set length (in the last column) for this program is much higher than for the others (except `peval`) indicating that it makes heavy use of higher-order functions and/or that a larger number of functions are stored in data-structures. Since the sets contain more elements, relatively more time is spent in the set manipulation procedures compared to

[3] We consider that the time spent in the code generation phase can be amortized if the analysis program is to be run several times. Such a situation can arise in the global analysis of separately compiled modules.

	Number of lines	Number of iterations	Average set length	Execution times		
				interp	closure	closure+gen
`conform`	557	3	0.59	0.0648	0.0154 (4.2)	0.0416 (1.56)
`earley`	648	3	0.41	0.0603	0.0117 (5.2)	0.0366 (1.65)
`interp`	411	9	3.02	0.1230	0.0435 (2.8)	0.0612 (2.01)
`lambda`	617	4	0.60	0.1050	0.0326 (3.2)	0.0634 (1.66)
`lex`	1133	3	0.59	0.1290	0.0266 (4.8)	0.0752 (1.72)
`link`	1608	6	2.22	0.4687	0.1322 (3.5)	0.2116 (2.12)
`ll1`	613	5	0.43	0.0940	0.0226 (4.2)	0.0470 (2.00)
`peval`	618	5	6.80	0.1727	0.0508 (3.4)	0.0896 (1.93)
`source`	453	5	0.77	0.0773	0.0195 (4.0)	0.0400 (1.93)

Fig. 8. Comparison of two strategies.

the time spent to traverse the syntax tree. So it is not surprising that the interpretation overhead will be less significant and the speedup lower than the other programs. Note also that even if we consider the time required to generate the analysis programs (the gen+closure column), the overall speedup is close to 2 in almost all cases.

Figure 9 shows the relative benefits of the optimizations we have discussed, namely the η-reduction and the `lookup` optimization. The first column gives the execution times (in seconds) for the analysis programs when the abstract compiler does not perform any optimization. The next two columns give the percentage of work that is saved by each optimization. The last column gives the same information when both optimizations are performed. It proves that it is worth the effort to add them to the abstract compiler.

Program	No opt.	η-reduct.	Lookup	All opt.
`conform`	0.0292	18.7%	28.8%	47.3%
`earley`	0.0274	26.4%	37.0%	57.3%
`interp`	0.0675	22.0%	15.7%	35.6%
`lambda`	0.0507	23.1%	18.6%	35.8%
`lex`	0.0547	21.6%	31.9%	51.4%
`link`	0.2137	20.6%	16.2%	38.3%
`ll1`	0.0410	21.5%	16.9%	44.9%
`peval`	0.0846	20.0%	17.7%	40.0%
`source`	0.0370	24.7%	18.5%	47.5%

Fig. 9. Relative benefits of each optimization.

6 Related Work

Although our work is novel in the field of static analysis, it is related to a number of other works.

Our work originated from the study of abstract interpretation, a framework well-adapted for the design of static analyses. A growing interest has been shown for this framework since the pioneering work of the Cousots [4]. It has been applied to a number of interesting analyses in the area of functional programming, including strictness analysis [11], reference counting [7], and control flow analysis [15]. In [2], Ayers presents several techniques for the efficient implementation of the *0cfa*. His "initial call sites" correspond to the calls where our technique can perform the `lookup` optimization.

Although we present here a more efficient implementation of *0cfa*, our paradigm is not restricted to analyses designed in the abstract interpretation framework. In fact, it can be applied to more conventional data-flow analyses [1, 6].

In [13], Serrano describes the control flow analyses performed in the Bigloo Scheme to C compiler [14]. His results show that these analyses allow significant optimizations to be performed. But they also show that it can take 4 to 9 times longer to compile a program when all the optimizations are enabled. Since his compiler produces C code, the time spent by the analysis and optimization process is often compensated by the time saved during the C compilation (the generated C code is easier to compile).

We argue that if the Bigloo compiler was to produce native code instead of C code, the time lost would not be compensated, thus revealing the real cost of control flow analysis (which has an $O(n^3)$ worst-case complexity).

Lin and Tan [10] show how to compile the dataflow analysis of logic programs. Although they named their technique "abstract compilation", they actually compile Prolog programs into code for the Warren Abstract Machine (WAM) using a (standard) compiler and that code is passed to an abstract interpreter for the WAM which computes the dataflow analysis. So there is no concept of analysis program in this technique, the abstract interpretation is being computed from another intermediate representation of the source program. Thus, optimizations of the analysis program, as those we described here, cannot be performed.

The use of closures for code generation has previously been proposed for compilation [5]. The latter describes an approach to compiling where each compiled expression is embodied by a closure whose application performs the evaluation of the given expression. This idea of replacing "code generation" by "closure generation" is essentially the same as we used in our compilation algorithm.

Closure generation is a form of runtime code generation. Leone and Lee [9] describe a technique for runtime code generation called *deferred compilation.* Their results also show that significant speedups can be obtained. For example, multiplication of sparse matrices is sped up by a factor of 2 using their technique.

7 Future Work

The analysis programs can be much faster if we consider generating machine code instructions instead of closures. Preliminary results show that they can typically be sped up by yet another factor of 4 in almost all cases. The only drawback is that the (abstract) compilation time also increases by a factor of 10, making the overall process slower than abstract interpretation if the analysis program is run a small number of times. The Scheme-to-C interface that we use is in part responsible for this increase. Also, the creation of a closure at runtime is much faster than the generation of a relatively long sequence of machine code instructions that do the same work. Nevertheless, we believe that the efficiency of the code generation can be further improved.

Our approach is most beneficial when the analysis is performed several times. This led us to the idea that the analysis program can be stored in a file and later reloaded together with other analysis programs in order to perform global control flow analysis of separately compiled multimodule programs. We are currently working on this idea [3].

8 Conclusion

We have presented a new way of implementing static analyses. It is based on the concept of *abstract compilation*. This paradigm is attractive for several reasons. It is conceptually simple, it is not restricted to any kind of static analysis, and more importantly, it can speed up the analysis.

As an example, we have described how to compile the control flow analysis of higher-order functional languages and our results have shown that the analysis can be sped up by over a factor of 2.

References

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

[2] Andrew E. Ayers. *Abstract Analysis and Optimization of Scheme.* PhD thesis, MIT, September 1993.

[3] Dominique Boucher and Marc Feeley. *Un système pour l'optimisation globale de programmes d'ordre supérieur par compilation abstraite séparée.* Technical report 992, Université de Montréal, september 1995.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles,* 1977, pp. 238–252.

[5] Marc Feeley and Guy Lapalme. Using closures for code generation. *Comput. Lang.* **12**, 47–66, 1987.

[6] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York, 1979.

[7] Paul Hudak. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* 351–363, 1986.
[8] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* Ph.D. thesis, Yale University, 1988.
[9] Mark Leone and Peter Lee. Lightweight Run-Time Code Generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,* pp. 97–106.
[10] I-P. Lin and J. Tan. Compiling Dataflow Analysis of Logic Programs. In *Proceedings of the 1992 ACM Conference on Programming Language Desing and Implementation,* pp. 106–115.
[11] Alan Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* Ph.D. thesis, University of Edinburgh, 1981.
[12] Guillermo Juan Rozas. Taming the Y operator. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming,* 226–234, 1992.
[13] Manuel Serrano. Control Flow Analysis: a compilation paradigm for functional languages. In *Proceedings of SAC 95.*
[14] Manuel Serrano. Bigloo User's Manual. Inria-Rocquencourt. March 1994.
[15] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. thesis, Carnegie Mellon University, Pittsburgh, 1991.