Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Laboratório de Linguagens de Programação

A Self-Applicable Partial Evaluator for ASM

by

Vladimir O. Di Iorio
Roberto S. Bigonha
Marcelo A. Maia

LLP 011/99

Caixa Postal 702
30.161-970 - Belo Horizonte
Minas Gerais - Brazil
September 1999

Abstract

This paper presents an offline partial evaluator for Abstract State Machines. In order to allow specialization of ASM programs containing user-defined functions (*derived functions*), the implementation combines techniques from partial evaluation of imperative and functional languages. Self-application of the partial evaluator is possible by means of a simplified version written in ASM itself. Using self-application, we have generated compilers for small languages from their interpreter definitions. We also present techniques for describing the semantics of programming languages in a way suitable for partial evaluation.

1 Introduction

A partial evaluator is an algorithm that, when given a source program and part of its input, produces a new program designated as *specialized program* [9]. The specialized program, when given the rest of the input, yields the same result that the original program would have produced with the entire input. Efficiency is an important goal in this process, so it is expected that the specialized program runs faster than the original one.

Abstract State Machines are a formal specification method created by Yuri Gurevich with the goal of simulating algorithms in a direct and coding-free way [6]. ASM has been often used to describe the semantics of programming languages [7, 12, 2]. The description usually consists of an interpreter for the programming language. Specializing the interpreter with respect to a source program yields an specific ASM for that program.

Huggins and Gurevich presented an offline partial evaluator for ASM in [8], which allows the specialization of conditional instructions and update blocks. This paper presents a partial evaluator that extended that work mainly in two aspects: it also allows the specialization of user-defined functions (*derived functions* [3]) and the self-application is possible, due to a version of the partial evaluator written in ASM itself. In addition, this paper presents some suggestions on how to describe the semantics of programming languages in ASM, in a way suitable for partial evaluation.

Derived functions are a mechanism to define functions by giving an expression to calculate them. This definition can be recursive, but side effects are not allowed. The implementation of partial evaluation of derived functions required the use of techniques for specialization of functional programs.

It is possible to compile by partially evaluating an interpreter with respect to a source program, yielding a target program written in the partial evaluator's output language. Partially evaluating a partial evaluator (self-application) with respect to an interpreter yields a compiler. This is possible if the partial evaluator and its input are both written in the same language. One can describe the semantics of a programming language by defining an interpreter for it, so self-application makes semantics-directed compiler generation possible. We have implemented a simplified version of the partial evaluator in ASM. Specializing the simplified version with respect to interpreters for small languages, we have generated compilers from those languages to ASM.

When describing the semantics of programming languages in ASM, important concerns are correctness and readability [7, 12, 2]. The interpreters are provided on several abstraction levels which make them easier to understand, but they are usually not suitable for partial evaluation. Without loss of readability, we propose small changes to these descriptions in order to achieve much better specialization results.

This paper is organized as follows. In section 2, we discuss the formal background for compiler generation by means of partial evaluation and its limitations and drawbacks. In section 3, the techniques used to implement our partial evaluator for ASM are presented and illustrated with examples. In section 4, we

discuss techniques that can make the description of the semantics of programming languages more suitable for partial evaluation. We present some results of self-application of the partial evaluator in section 5. Section 6 contains the conclusions and a discussion about future work.

2 Partial Evaluation and Compiler Generation

An offline partial evaluator consists of two phases. The first one is called *binding-time analysis* (BTA) [9], in which all objects and structures are classified either as *static* or *dynamic*, according to their relationship with the static (known) and dynamic (unknown) parts of the input. In the second phase the actual program specialization takes place, making use of the static inputs to the extent determined by the previous phase. The static structures are computed, and code is generated for the dynamic ones.

A partial evaluator performs a mixture of execution and code generation actions. This is the reason why Ershov called the process "mixed computation" [4], and the partial evaluator is usually called `mix`.

If P is a program written in a language L, we will use $[P]_L$ to denote its semantics. Futamura was the first researcher to suggest compiler generation by self-application of a partial evaluator [5]. Therefore, the equations describing compilation, compiler generation and compiler generator generation are called the *Futamura Projections*. Suppose that `int` is an interpreter for a language L_{int}, written in a language S:

$$\texttt{target} = [\texttt{mix}]_L\,(\texttt{int}, \texttt{source}) \tag{1}$$

$$\texttt{compiler} = [\texttt{mix}]_L\,(\texttt{mix}, \texttt{int}) \tag{2}$$

$$\texttt{cogen} = [\texttt{mix}]_L\,(\texttt{mix}, \texttt{mix}) \tag{3}$$

Equation 1 shows that compilation can be achieved by partial evaluation of an interpreter, with respect to a specific source program. The language of the programs submitted to the partial evaluator `mix` is S (*input language*), which is usually also the language of the programs produced by `mix` (*output language*). The partial evaluator `mix` is written in L (implementation language).

Equation 2 shows compiler generation by self-application of `mix`, specialized with respect to a specific interpreter. In this case, a compiler from L_{int} to S is generated. This way of performing compiler generation requires that `mix` be written in its own input language, i.e., that $S = L$.

In a partial evaluator for ASM the input language S is the ASM language. Our partial evaluator was implemented in Java, while that presented by Huggins and Gurevich was implemented in C. Let $\texttt{mix}_1$ be our partial evaluator. To achieve the results of equation 2, we have built $\texttt{mix}_2$ in ASM language, which is a simplified version of $\texttt{mix}_1$ implementing only the specialization phase of an offline partial evaluation method. The simplification is highly desirable in this case, otherwise the result of applying $\texttt{mix}_1$ to $\texttt{mix}_2$ would be time-consuming and generate longer and more inefficient programs.

As $\texttt{mix}_2$ does not implement the binding time analysis phase, its inputs are an annotated program P_{an} and the static inputs of P_{an}. The annotations define the static and dynamic structures of the program, established by an external BTA process, previously executed. To achieve compiler generation, we partially evaluate $\texttt{mix}_2$ with respect to a specific interpreter:

$$\texttt{compiler} = [\texttt{mix}_1]\,(\texttt{mix}_2, \texttt{int}_{an})$$

The annotated program is submitted to $\texttt{mix}_2$ in an abstract syntax tree format [1], so that $\texttt{mix}_2$ does not have to worry about syntax. Since the structures used to implement $\texttt{mix}_1$ have been also used to generate P_{an}, only little additional code has been necessary to implement the annotated program generator.

We present $\texttt{mix}_2$ in section 5, where we discuss implementation issues and evaluate the results of applying the self-applicable partial evaluator to small interpreters. We have not investigated the application of equation 3 yet.

3 A Partial Evaluator for ASM

In this section, we present the techniques used to develop an offline partial evaluator for Sequential ASM. The language includes update instructions, conditional commands and also user-defined functions. First, we show a small difference between the language we deal with and that of [3]: a new command named `stop`, used to interrupt the program execution. In some ASM implementations the program execution stops when no update is executed after firing the transition rules, but we decided to use this command for efficiency reasons.

The command `stop` can be easily translated into pure ASM. Let $\mathcal{A}$ be an ASM with vocabulary Υ and transition rule $\mathcal{R}$ containing `stop`. Let $\texttt{fstop} \notin \Upsilon$ be the name of a new basic function. The ASM $\mathcal{A}'$ with vocabulary $\Upsilon \cup \{\texttt{fstop}\}$, where `fstop` is initialized with `false`, and transition rule "`if not fstop then` $\mathcal{R}'$ `endif`" has the same semantics as $\mathcal{A}$, if $\mathcal{R}'$ results from the replacement of every command `stop` in $\mathcal{R}$ by "`fstop := true`".

3.1 Binding-Time Analysis (BTA)

The partial evaluation process starts with the definition of what parts of the input are static, i.e., known in advance. The terms *static* and *dynamic* have different meanings in ASM, so we will follow Huggins and Gurevich's suggestion to use *positive* instead of *static* and *negative* instead of *dynamic*. Our ASM interpreter offers predefined functions to do input and output, and the programmer can use annotations to define what parts of the input will be considered positive.

The purpose of the BTA phase is to classify all functions as either positive or negative, according to their relationship with the input. Then it classifies each command and function application, generating an annotated program that will be used in the subsequent phases.

Binding Time Identification We have implemented an algorithm similar to that used in [8] to compute the division of functions into positive and negative:

1. Input is done via an input file with a finite number of primitive values. The program gets these values using a predefined function `input(intval)`, where `intval` is an integer representing a position within the input file. The user indicates which positions are negative and which are positive. Initially, only the references to the predefined `input` function are classified as positive or negative, according to the user annotations. All other functions are left unclassified.
2. A function f is classified as negative if there is an ASM update rule $f(\bar{t}) := t_0$ such that $\bar{t}$ or t_0 references a negative function. Both $\bar{t}$ and t_0 may depend on derived functions. If, for all updates, every reference in $\bar{t}$ and t_0 is positive and f is not already negative, it is classified as positive.

   A derived function `f` has the following format:

$$\texttt{derived function f (param}_1\texttt{, ..., param}_k\texttt{) = expression}$$

where `expression` contains references to the formal parameters and possibly recursive invocations of `f`. Each invocation of `f` is classified as positive or negative according to the classification of the expressions used as parameters.

The second step of the algorithm is repeated until a fixpoint is reached. All remaining unclassified functions are classified as negative. Then this step is repeated until reaching a fixpoint by the second time.

It is important to classify a sufficient number of functions as negative to ensure finiteness of the specialization algorithm, but classifying more negative functions than necessary leads to poor specialization. The problem of finding an optimal division is not computable [9].

The classification algorithm above does not handle circular dependencies. The following example, borrowed from [8], illustrates this problem. Consider the following transition rules:

```
if Num > 0 then Num := Num + 1 endif
if MyList ≠ Nil then MyList := Tail(MyList) endif
```

Supposing that the initial values of `Num` and `MyList` are known, there is no problem in classifying `MyList` as positive. But classifying `Num` as positive will lead the specialization algorithm into an infinite loop, because the number of different values `Num` can assume is infinite and the specialization algorithm tries to generate code using each one of them.

We treat circular dependencies in a way similar to that of [8]. We classify as positive the functions that depend only upon themselves in a bounded manner. For example, `MyList` will eventually be reduced to `Nil` and remain at that value. We apply the same criteria to classify as positive the derived functions that depend upon themselves (recursive) in a bounded manner.

For our division algorithm is not very sophisticated, we allow users to help the classification of functions with annotations. We intend to eliminate this possibility in the future, building a totally automated process.

Generation of the Annotated Program All expressions and commands are also classified as either positive or negative, generating an annotated program. This process is performed observing the following situations:

- a reference to the predefined input function is classified as positive or negative according to the user annotations. It is the beginning of the entire classification process;
- output is implemented by an `output(expression)` command. It is always classified as negative;
- a term $f(\bar{t})$ is classified as negative if any of the terms in $\bar{t}$ is negative; it is positive if all terms in $\bar{t}$ are positive and f is not already negative;
- an update command $f(\bar{t}) := t_0$ is negative if f is negative, and is positive if f is positive;
- a conditional command `if (condition) then ...` is negative if the boolean expression `condition` is negative, and is positive if `condition` is positive;
- the command `stop` is always classified as negative.

When the classification algorithm ends, the entire program is annotated. The last step consists of producing different versions of the user-defined (derived) functions, according to the classification of the expressions used as parameters in each invocation.

An invocation of a derived function `f` inside the transition rules of the program has the format `f (exp`$_1$`, ..., exp`$_k$`)`, where each `exp`$_i$, $1 \leq i \leq k$, was already classified either as positive or negative. Let `(S`$_1$`, ..., S`$_k$`)` represent the classification of those expressions, where `S`$_i$, $1 \leq i \leq k$, is either *positive* or *negative*. A different version of `f` is generated for each different tuple `(S`$_1$`, ..., S`$_k$`)`, and the structures inside the derived function expression are classified according to these values. Each invocation of `f` is translated into an invocation of one of the new versions produced.

At the end of the BTA phase, the set of functions referenced in the program was divided into positive and negative, and an annotated program is produced. The annotated program contains one or more versions of each derived function, according to the process described above.

3.2 Pre-Processing

In [8], a pre-processor performs some transformations over the original program, in order to make the specializer simpler. These transformations increase the program size, possibly exponentially. At the end of the process, the program is represented as a binary tree, where leaves are update blocks and internal nodes are boolean guards.

We have decided to postpone the pre-processing of the program until after the BTA phase. The information produced by that phase is used to minimize code expansion. In the sequel, we explain how transformations are performed.

In each block, all rules are reordered so that all conditional commands lie at the end of the block. If there are at least two conditional commands in the block,

the last one is removed and inserted simultaneously into the THEN and ELSE clauses of one of the other conditional commands. This process is recursive and is repeated until all blocks contain at most one conditional command. At the last recursion level, a special `dummy` command is inserted into each block. This command will be used later by the specializer to indicate the generation of a new state. Only blocks containing a `stop` command will not have a `dummy` inserted, because they will not generate new states during specialization.

An important difference between our algorithm and that of [8] is that we avoid many duplications of update instructions. In our algorithm, the innermost blocks contain a special `dummy` command indicating the generation of a complete state, while in [8] they contain all possible update instructions. The purpose of the `dummy` command will be clear when we discuss the specializer algorithm in the next section.

We also avoid the duplication of some conditional commands, with the help of BTA information. A positive conditional command whose clauses contain only update instructions is not duplicated, avoiding, in some cases, the code expansion imposed by the algorithm above. One might argue that this is not a very frequent case, but our experience with interpreters for programming languages written in ASM has shown the opposite.

3.3 Specializer

Specialization is executed over the program produced by the pre-processor. Let Fp be the set of functions classified as positive. The specializer works on an ASM $\mathcal{A}_{Fp}$ whose vocabulary is restricted to Fp, trying to establish all possible states which are reachable from the initial state.

The initial state is represented by the initial values of the positive functions. In each iteration, the entire program is analyzed. Code is generated for the negative structures, and positive update instructions generate new states. Each new state is processed, producing an associated code and generating more states, which may have been already processed. This procedure ends when all possible states have been processed. The way BTA has been done ensures that the procedure will eventually end, unless there is an infinite loop produced by the positive functions.

To generate the code associated with a state and define the new states to be generated, each transition rule of the program is processed in the following way:

- If the rule is a block, each command in the block is processed.
- A positive update command adds an update to the set of current updates.
- A negative update command $f(\bar{t}) := t_0$ generates code for an update instruction with all positive information within $\bar{t}$ and t_0 computed.
- In a positive conditional command, first the condition is evaluated. If it is true, the THEN clause is processed, otherwise the ELSE clause is processed.
- A negative conditional command generates code for a conditional command where all positive information of the condition expression is computed and the THEN and ELSE clauses are processed. In this case, however, the set of

current updates is duplicated. A copy of this set will be used when processing the THEN clause, and the other, when processing the ELSE clause.
- Commands `stop` and `output` simply generate code for themselves. All the positive information in the `output` expression is computed.
- A `dummy` command indicates the point where all updates that build a new state have already been processed. All registered updates are fired at the current state, in parallel, and a (possibly new) state is generated. The generated code is a `dummy` command together with information that references the generated state, as we will show soon.

Let k be the number of different generated states. After all states have been processed, a set of rules $\{R_1, \ldots, R_k\}$ has been generated, each associated with a different state. Suppose that R_1 was produced when processing the initial state. Each `dummy` command inside R_1 contains a reference to one of the states, describing which are the possible rules to be executed in the next step. The `dummy` command defines the dynamic flow of control and could be translated into a `goto`, if the language had such a command.

Let `pe_flag` be a function name that does not belong to the vocabulary of the original ASM submitted to the partial evaluator. Each command `dummy` is replaced by `pe_flag` $:= i$, where i is the state referenced by it. The final program has the following format:

```
Initial values:
    pe_flag := 1
Transition rules:
    IF pe_flag = 1 THEN R1
    ...
    IF pe_flag = k THEN Rk
```

The specialization of the derived functions demands other procedures. Suppose that `f` is a derived function used inside the transition rules of the program. During BTA, one or more annotated versions of `f` have been generated. Let $\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$ be this set of annotated versions. For $1 \leq i \leq n$, suppose that the number of $\mathtt{f}_i$ parameters classified as positive is N_{f_i} and build a tuple $(p_1^{\mathtt{f}i}, \ldots, p_{N_{f_i}}^{\mathtt{f}i})$ with these parameters. The invocations of $\mathtt{f}_i$ inside $\{R_1, \ldots, R_k\}$ are used to identify all possible different values for this tuple. Each different tuple value originates a new specialized version of $\mathtt{f}_i$.

As we have shown, the specialization of derived functions is carried out in two steps, generating two levels of specialized versions. We have used indexes to identify each version, so when a derived function f is residualized in the final program, its name usually has the format `f_`i`_`j, where i and j are indexes used for identification.

3.4 Optimizations

The code generated by the specializer is usually very inefficient, there are many opportunities for code optimizations. Some of them were implemented in [8] and also in our partial evaluator.

A rule with the format

$$\texttt{if pe\_flag} = i \texttt{ then pe\_flag :=} j$$

can be deleted, if all references to state i are replaced by j. Another important optimization is combining the code of two rules that would be executed consecutively, but whose codes can be executed simultaneously without altering the semantics of the program.

The two optimizations listed above are usually known as *transition compression*, what, in some imperative languages, would mean elimination of redundant `gotos`. We have decided to do transition compression *on the fly*, i.e., during the specialization process, instead of doing it as a separate phase after the whole residual program has been generated. Although the strategy chosen is a little bit more complicated, it can be much more efficient, due to the great number of superfluous rules that are usually generated by the specialization process. Another good reason to choose this strategy is that it improves the results of self-application significantly [9].

The number of residualized derived functions is also usually very high. In many cases, the function is residualized as a simple expression, without recursive calls. In these cases, each function call is replaced by the associated expression, with the correct values substituted for the parameters. The function definition does not appear in the final code.

Suppose that a negative function f with arity n is always used with positive information for its k-th parameter, $1 \leq k \leq n$. The residualized functions associated with f will have arity $n-1$, and the positive values will be used to build the names of the new functions. For example, if the first parameter of a negative function `func` is always positive, a call `func("key",x)` will be residualized as `func_key(x)`. If the values of the parameters are not appropriate to build names, a different integer number is associated with each different value, and this number is used to build the new function name. This process can be extended to multiple positive parameters. The partial evaluator will use as many positive parameters as possible.

Other simple optimizations have been implemented. Some of them use also online information.

3.5 A Turing Machine Semantics

We will show a very simple example of specializing an interpreter with respect to a specific source program, which we have borrowed from [9]. More sophisticated examples are treated in section 4. From now on, we will use $[e_1\ e_2\ \ldots\ e_n]$ to denote a list with n elements. The associated operations on lists are `car`, `cdr`, and `isnull`. The empty list is [ ] or `null`.

Consider a version of the *Turing Machine* with alphabet $\{0, 1, \texttt{undef}\}$ and a program $[\texttt{I}_0\texttt{I}_1\ \ldots\texttt{I}_n]$, where each $\texttt{I}_i$, $0 \leq i \leq n$, is one of the following instructions: `right`, `left`, `write` a, `goto` i, `if` a `goto` i, `stop`. The machine has a tape head indicating the current scanned tape cell. The semantics of the

instructions are: `write` a changes the scanned cell to a, `right` and `left` move the tape head, `if` a `goto` i causes the next instruction to be $\mathtt{I}_i$ if the scanned cell contains the value a, `goto` i is an unconditional jump, and `stop` interrupts the program execution. The instructions are executed in sequence, unless a jump is executed.

The following TM program changes the first 0 it finds in the tape to 1, or goes into an infinite loop if no 0 is found:

```
0: if 0 goto 3
1: right
2: goto 0
3: write 1
4: stop
```

Next, we show the ASM transition rules which implement an interpreter to the language described above. Suppose that the machine tape is represented by function `tape`. The program instructions are fetched by functions `code`, `par1` and `par2` such that, when given the instruction number, return the code, the first parameter and the second parameter of the instruction, respectively. The number of the current instruction is represented by `pc` and the tape head is represented by `head`:

```
IF code(pc) = "GOTO" THEN
    pc := par1(pc)
IF code(pc) = "IFGOTO" THEN
    IF par1(pc) = tape(head) THEN
        pc := par2(pc)
    ELSE
        pc := pc + 1
IF code(pc) = "WRITE" THEN BEGIN
    pc := pc + 1
    tape(head) := par1(pc)
END
IF code(pc) = "LEFT" THEN BEGIN
    pc := pc + 1
    head := head - 1
END
IF code(pc) = "RIGHT" THEN BEGIN
    pc := pc + 1
    head := head + 1
END
IF code(pc) = "STOP" THEN
    STOP
```

Suppose that this interpreter is submitted to the partial evaluator, to be specialized with respect to the TM program given in this section. The BTA phase will classify `pc`, `code`, `par1` and `par2` as positive, while `tape` and `head` will be classified as negative. The functions `pc` and `head` are examples of the

special BTA case described in section 3.1. A code analysis shows that pc depends upon itself in a bounded way, and so it is classified as positive. On the other hand, head can assume an infinite set of different values, and so it is classified as negative. The result of the specialization is showed next. We have omitted the initialization code.

```
IF tape(head) = 0 THEN BEGIN
    tape(head) := 1
    STOP
END
ELSE head := head + 1
```

The function pe_flag used in section 3.3 defines an order of execution for the set of rules. In the example above, the transition compression reduced the number of rules to 1, so pe_flag was not necessary. The code shown above is exactly that produced by the partial evaluator.

4 Interpreters Suitable for Partial Evaluation

We have assumed in section 3.1 that the BTA division of the functions into positive and negative is unique and remains the same during the whole specialization process. This is called an *uniform division* [9]. For most small programs, an uniform division is enough to guarantee good specialization results, but for larger programs it is sometimes overly restrictive, as we will show soon.

The description of the semantics of programming languages in ASM usually consists of an interpreter for the language, written in ASM. Most times, the interpreter is large enough to make the uniform division of the functions in a specialization process lead to poor results.

In this section, we will show two examples with a description of a programming language. The first one involves the description of C, and we discuss problems related to the function return mechanism. The second example involves the description of a subset of Java, and we discuss problems related to polymorphism. A direct approach to treat the function return mechanism and polymorphism make the uniform division be unsuitable for a good specialization. We show how small changes in the description can lead to good specialization results, even using an uniform division.

4.1 First Example: Function Return in C

In the example showed in section 3.5, all structures related to the TM program were classified as positive, so none of them appeared in the residualized program. We will consider now an interpreter for a more sophisticated language, with definition of recursive functions.

The description of the semantics of the C programming language presented in [7] includes the definition of C recursive functions. An ASM function CurTask plays a role similar to pc in the example of section 3.5: it indicates the current

task to be performed. From now on, suppose that the interpreter for the C programming language is specialized with respect to a specific source program. It is desirable that `CurTask` be classified as positive, so compiling from C to ASM can achieve results similar to that of section 3.5.

C functions may have several active incarnations at a given moment. In [7], the next task to be performed after a function returns is stored in a stack. This is implemented by the following ASM commands:

```
ReturnTask(StackTop+1) := CurTask
StackTop := StackTop + 1
```

where `StackTop` represents the top of the stack and `ReturnTask` will indicate the new value of `CurTask` when the execution of the current function terminates. These commands are executed immediately before the flow of control is transfered to the function body. The set of values `StackTop` can assume is infinite, so BTA will classify it as negative, and consequently `ReturnTask` will also be negative.

When the execution of the current function terminates, the next task to be performed is on the stack. `CurTask` is updated with this value. The following ASM commands are part of the semantics of the C `return` statement, that terminates a function execution:

```
CurTask := ReturnTask(StackTop)
StackTop := StackTop - 1
```

The first update above will make BTA classify `CurTask` as negative, creating an undesirable situation.

A solution for the problem above is to use a *pointwise division* in BTA [9]. In a pointwise division, functions can receive different classifications in different points of the program submitted to the partial evaluator. `CurTask` could be classified as negative in the few points where it is necessary, and positive in the rest of the program. The annotations in the program produced after BTA would suffer some changes, and the implementation of the specializer would be a little bit more complicated.

Neither our partial evaluator, nor the one presented in [8] implements pointwise division. We intend to add this possibility in the future, but we want to show here that it is still possible to achieve good specialization results with an uniform division. All that is necessary is to change slightly the C interpreter. The changes we propose will make BTA classify `CurTask` as positive, even in an uniform division.

Let f be a C function in a program P submitted to the C interpreter. The places where f is called inside P are all known before the execution of the program starts. It is an information that depends only on P, so a list $[t_0\ t_1\ \ldots\ t_n]$ of the possible next tasks after f returns can be computed before the execution of P starts. Let `ListRetTasks` be an ASM function that associates a function name with the list of its possible next tasks after returning. In this case, we would have `ListRetTasks("f")` $= [t_0\ t_1\ \ldots\ t_n]$. The function whose execution is terminated by each C `return` statement is also known before the program starts.

Let NameFuncReturn be an ASM function that represents the name of such C function. Part of the new code for the semantics of the C `return` statement is showed next:

```
if CurListRet = undef then
    CurListRet := ListRetTasks(NameFuncReturn)
else if ReturnTask(StackTop) = Car(CurListRet) then begin
    CurTask := Car(CurListRet)
    StackTop := StackTop - 1
    ...
end
else
    CurListRet := Cdr(CurListRet)
```

We have used the ASM function CurListRet to store the current list of the possible next tasks. We assume that ListRetTasks is well built, so it is not necessary to include code to treat the case where CurListRet becomes empty.

Note that now CurTask is updated with values from CurListRet. The function ListRetTasks will be classified as positive because it depends only on static information from the source program P. Therefore CurListRet and CurTask will also be classified as positive. The residualized code will consist of a sequence of conditional commands, each testing a possible value for ReturnTask(StackTop):

```
if ReturnTask(StackTop) = t0 then begin
    { code resulted from specialization with CurTask = t0 }
    StackTop := StackTop - 1
    ...
end
else if ReturnTask(StackTop) = t1 then
    ...
```

In [7], the semantics of the C programming language is specialized with respect to the C function `strcpy`. To achieve the good results showed, only a part of the semantics of C is considered. The partial evaluator is given an interpreter that does not describe the semantics of function call and return and *automatic* variables. Using the techniques described in this section, we have been able to specialize the entire semantics of C.

4.2 Second Example: Polymorphism in Java

We have made some experiences with the description of the semantics of a small subset of Java. Our description includes creation of objects and implementation of polymorphism.

Suppose that C_1 and C_2 are Java classes, where C_2 is a subclass of C_1. Suppose that both classes have a function `f` with the same signature:

```
int f();
```

Let `x` be an object reference of class C_1. In the following function call, it is not possible to know in advance which `f` version is invocated:

```
int i = x.f();
```

It depends on the class of the object referenced by `x` in the moment of the call. The object referenced by `x`, in this case, can be either of class C_1 or class C_2. It is an information available only at execution time.

The following Java command creates an object of class C_2 and associates an object reference of class C_1 to it. We do not consider object initialization yet:

```
C1 x = new C2( );
```

When an object is created, memory is allocated to store its value. The object reference `x` is updated with the memory address of the newly created object. Information to associate this memory address with class C_2 is also stored.

As in last section, let `CurTask` be an ASM function to represent the current task to be performed. In a function call, `CurTask` must be updated to the first task inside the function body. Let `FuncFirstTask` indicate the first task of a function, when given the name of that function and the class the function is associated to. In the function call `x.f()`, `f` is associated to object reference `x`. So it is necessary to first discover the class of the object referenced by `x`.

Let `Obj_to_Class` be an ASM function that associates an object (represented by a memory address) with a Java class. Let `CurFunc` be the name of the function in the current function call, and let `CurObj` be the object reference associated with it. Finally, let `Memory` indicate the value stored in a given memory location. In a function call associated with an object reference, the following ASM command updates CurTask with the first task inside the function:

```
CurTask := FuncFirstTask (CurFunc, Obj_to_Class(Memory(CurObj)))
```

`Memory` stores the values of the Java objects during execution. If the Java interpreter is specialized with respect to a source program, BTA will classify `Memory` as negative. The ASM command showed above will make BTA also classify `CurTask` as negative. As we have discussed in the last section, this can lead to poor specialization results.

We can use the trick of the last section to make BTA classify `CurTask` as positive, even in an uniform division. The information about object references and class hierarchy depends only on the source program. The object reference `x` belongs to class C_1, so it can be associated, during execution, to an object of class C_1 or C_2.

Suppose that `CurListClasses` is initialized with a list containing the possible classes associated with the current object reference. In the example discussed, `CurListClasses` $= [C_1\ C_2]$. The classes can be represented by their names, for example. The following ASM rules define a different way to update `CurTask`:

```
if Obj_to_Class(Memory(CurObj)) = Car(CurListClasses) then
    CurTask := FuncFirstTask (CurFunc, Car(CurListClasses))
else
    CurListClasses := Cdr(CurListClasses)
```

BTA will classify `CurListClasses` as positive, because all the information necessary to build it depends only on the source program. So `CurTask` will also be classified as positive. The residualized code will be a sequence of conditional commands. For example:

```
if Obj_to_Class(Memory("x")) = C1 then
    CurTask := FuncFirstTask ("f", C1)
else if Obj_to_Class(Memory("x")) = C2 then
    CurTask := FuncFirstTask ("f", C2)
```

We have made some simplifications in the code above. For example, the value associated with `x` would also depend on the environment.

5 Self-Application of the Partial Evaluator

As discussed in section 2, we have built a simplified version of our partial evaluator in ASM itself. We have called the main partial evaluator $\texttt{mix}_1$, and the simplified one has been named $\texttt{mix}_2$. Good results on self-application of a partial evaluator have become possible only after offline methods have been developed. Dividing the process into BTA and specialization phases, it is possible to submit to the partial evaluator a simplified version that performs only the specialization phase. This is what we have done in $\texttt{mix}_2$.

Programs are submitted to $\texttt{mix}_2$ in an abstract syntax tree format. First, they are pre-processed as described in section 3 and receive BTA annotations. The same pre-processor and BTA procedures implemented to be used in $\texttt{mix}_1$ are also used to generate the transformations and annotations for the $\texttt{mix}_2$ input program. The generation process of the abstract syntax tree format is also automated.

We have made some experiences with compiler generation by means of self-application of the partial evaluator. The simplified version $\texttt{mix}_2$ is specialized with respect to an annotated interpreter $\texttt{int}_{an}$ for a language L_{int}, yielding a compiler from L_{int} to ASM:

$$\texttt{compiler} = [\texttt{mix}_1]\,(\texttt{mix}_2, \texttt{int}_{an})$$

The compiler is an ASM program that receives as input a program P written in L_{int} and produces another program, written in ASM, with the same semantics of P.

In this section, we describe how the simplified partial evaluator works and present some results of compiler generation using it. First, we show a brief description of the format of the programs submitted to $\texttt{mix}_2$.

5.1 Input Programs

Lists are used to represent the abstract syntax tree format of the programs submitted to $\texttt{mix}_2$. Programs are divided into three sections: function declarations,

initialization and transition rules. These sections are represented by lists of declarations and commands. Each command, declaration and function application is also represented by a list, whose first element denotes its binding time: "+" for positive and "-" for negative. Next, we show a small example of the adopted format.

In the ASM program of section 3.5, the command

$$\texttt{pc} := \texttt{pc} + 1$$

would be represented as

```
["+" "update" "pc" null
    [ "func" "+" "plus" [
        [ "func" "+" "pc" null ]
        [ "int" 1 ]
    ]]
]
```

An update command has the format

$$[\textit{signal} \quad \text{"}\texttt{update}\text{"} \quad \textit{fname} \quad \textit{listexp} \quad \textit{expression}\ ]$$

where *signal* denotes the binding time associated, *fname* is the name of updated function, *listexp* is the list of parameters and *expression* is the value to be calculated. A function application has the format

$$[\ \text{"}\texttt{func}\text{"} \quad \textit{signal} \quad \textit{fname} \quad \textit{listexp}\ ]$$

This format is used in any function application, including those which have an empty list of parameters (for example, the ASM function `pc`).

In the example above, the ASM function `pc` has been classified as positive by the BTA phase. Annotations have been added to indicate that the update command and the function applications are also positive.

5.2 Inside $\texttt{mix}_2$

The simplified partial evaluator performs only the specialization phase, analyzing each command and following strictly the associated annotations. In this section, we describe how $\texttt{mix}_2$ behaves while processing the three sections of its input programs: function declarations, initialization and transition rules.

When $\texttt{mix}_2$ finds a positive command, it behaves like an ASM interpreter. All the information is computed according to ASM semantic rules. The simplified partial evaluator needs a complete ASM interpreter to compute the positive structures. So, before implementing $\texttt{mix}_2$, we have first built an ASM self-interpreter, i.e., an interpreter for ASM written in ASM itself. In section 6 we discuss how this self-interpreter has been used to evaluate the performance of the main partial evaluator.

Function Declarations The first task in $\texttt{mix}_2$ is to determine the list *POS* of positive functions used to build the different states. This task is performed while $\texttt{mix}_2$ process the function declarations of the input program. If a function is not updated after initialization, its value remains the same in all states, so it does not have to be considered in *POS*, even if it is positive. In the example of section 3.5, the only positive function that is updated by the transition rules is `pc`, so $POS = [\text{"}\texttt{pc}\text{"}]$.

Code is generated for the declarations of negative functions. The residualized declarations have exactly the same format of the input program, except for the annotations. All derived functions are residualized, because $\texttt{mix}_2$ does not treat specialization of derived functions yet. We are working now to introduce this feature.

Initialization An ASM function `FVal` is used to store all information related to the functions used in the input program. `FVal` associates a function name and a list of values to another value. Using the example of section 3.5, $\texttt{Tape}(1) = 0$ would be represented by $\texttt{mix}_2$ as $\texttt{FVal}(\text{"}\texttt{Tape}\text{"}, [1]) = 0$. As `FVal` is used to store the function values, it is classified as negative. The simplified partial evaluator is built in such a way that the first parameter of `FVal` is always positive, so a function call $\texttt{FVal}(\text{"}\texttt{Tape}\text{"}, [1])$ will be residualized as $\texttt{FVal\_Tape}([1])$. This transformation was described in section 3.4.

The initial values of the functions are defined by update instructions in the *initialization section*. Code is generated for the initialization of negative functions, and the initial values of the positive functions are computed. Using *POS*, the initial state for the specialization process is built.

The simplified partial evaluator keeps all states produced during the specialization process in a table, which is represented by an ASM function designated as `pe_states`. Each state is represented by a list of values, associated with the functions listed in *POS* by their positions. In the example of section 3.5, the initial state would be the list [1], which means that initially `pc` has value 1.

Transition Rules An ASM function `pe_index` indicates which of the states of the `pe_states` table is the current state being processed. The transition rules are analyzed, with all positive information computed using the values of the current state. The process is very similar to that described in section 3.3, where the specialization algorithm of $\texttt{mix}_1$ is showed.

Positive updates are registered in a set of current updates. A negative conditional command makes the set of current updates be duplicated and used when processing the THEN and ELSE clauses. The `dummy` commands indicate that a (possibly new) state will be built by firing the set of current updates at the current state. The code generation for the negative structures is also similar to that of $\texttt{mix}_1$, but the code is generated in an abstract tree format, the same used for the input program.

The new states produced are inserted in `pe_states`. An ASM function `pe_last` indicates the last position of the `pe_states` table. The process stops when `pe_index` = `pe_last`.

Most of the optimizations presented in section 3.4 have not been implemented in $\texttt{mix}_2$ yet.

5.3 Compiler Generation

As we have stated before, the simplified partial evaluator still has some limitations. It does not specialize derived functions and most of the optimizations of $\texttt{mix}_1$ have not been implemented yet.

We are working in order to build a version of $\texttt{mix}_2$ that is equivalent to the specialization phase of $\texttt{mix}_1$. Supposing that `int` is an interpreter for a language L_{int} and source is a program written in this language, the following result can be accomplished, if the two specializers are equivalent:

$$\texttt{compiler} = [\texttt{mix}_1]_{Java}\,(\texttt{mix}_2, \texttt{int}_{annotated}) \tag{4}$$

$$[\texttt{compiler}]_{ASM}\,(\texttt{source}) = [\texttt{mix}_1]_{Java}\,(\texttt{int}, \texttt{source}) \tag{5}$$

The left hand side of equation 5 is the result of compiling `source` to ASM with the compiler generated by self-application of the partial evaluator. The right hand side is the result of compiling by means of partially evaluating an interpreter with respect to a source code. The right hand side of equation 5 will produce better results until optimizations are introduced in $\texttt{mix}_2$.

The discussion above is related only to the quality of the code generated by `compiler`. Other important issues are the size of the compiler generated with self-application and its efficiency (how fast it generates code). These properties are affected by the optimizations present in $\texttt{mix}_1$ and the way $\texttt{mix}_2$ is built. The examples of section 4 showed how appropriate changes in a program can lead to much better specialization results.

Because of the restrictions discussed above, we have only made compiler generation experiences with simple interpreters until now. The compiler generated by specializing $\texttt{mix}_2$ with respect to the Turing Machine interpreter is more than six times longer than the interpreter itself. In average, running a TM program compiled using this compiler is three times faster than interpreting the same program with the TM interpreter. The quality of the code generated by the compiler is worse than that generated by specializing the interpreter, specially because the lack of transition compression optimizations in $\texttt{mix}_2$.

We have also made some experiences with other small interpreters, specially the description of a small subset of C. The results produced have been similar to those described above.

The simplified partial evaluator $\texttt{mix}_2$ has been built in such a way that implementing transition compression is not a hard work. We intend to do this soon, improving the quality of the generated code. We expect that small changes in the code of $\texttt{mix}_2$ will lead to compilers not so long. After implementing specialization of derived functions, we will make experiences with more complex interpreters, for example, the entire semantics of the C programming language and Java.

6 Conclusions and Future Work

In order to evaluate the performance of a partial evaluator, a self-interpreter test is suggested in [9]. Suppose that an interpreter for the partial evaluator's input language S is written using the same language S. Specializing this interpreter with respect to a program P must yield the same program P, if the partial evaluator is powerful enough to remove all *interpretational overhead*.

As described in section 5.2, we have built a self-interpreter for ASM. The self-interpreter has been specialized with respect to many long ASM programs. The residualized programs produced have been exactly the same given as input, except for some function renaming.

Results similar to those presented in section 3.5 have been achieved with more complex languages. Compilation by means of partial evaluation of the entire semantics of C and a small subset of Java have produced very good results. To accomplish this, the techniques presented in section 4 have played a fundamental role.

The main weakness of the partial evaluator developed is the BTA phase. The algorithms we have used are not very sophisticated, so user annotations are sometimes needed. We intend to improve BTA soon, adding to it even the possibility of a *pointwise division* of the functions.

We have considered the results of the self-interpreter test and compilation by means of the *First Futamura Projection* very good. On the other hand, the simplified partial evaluator $\texttt{mix}_2$ built for self-application is still very simple. As stated in section 5, our plans include adding a lot of new features to $\texttt{mix}_2$ soon, namely: specialization of derived functions, transition compression and other optimizations. Then we will be able to apply the *Second Futamura Projection* to more complex languages. The simplified partial evaluator is the first step on developing a work on semantics-directed compiler generation using ASM and partial evaluation. The *Third Futamura Projection* (compiler generator generation) will be considered later.

Partial evaluation of concurrent and parallel programs may be a great source of research. Our future plans include extending the partial evaluator to deal with ASM extensions such as Parallel and Distributed ASM [6], and Interactive ASM [11, 10].

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.
2. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
3. G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

4. A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
5. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
6. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
7. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.
8. Y. Gurevich and J. Huggins. Evolving Algebras and Partial Evaluation. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 587–592, Elsevier, Amsterdam, the Netherlands, 1994.
9. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
10. M. Maia, V. Di Iorio, and R. Bigonha. Interacting Abstract State Machines. In *V International Workshop on Abstract State Machines*, Magdeburg, Germany, September 1998.
11. M. Maia, V. Iorio, and R. Bigonha. Interacting Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.
12. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.