A Self-Applicable Partial Evaluator for ASM

Vladimir O. Di Iorio[1], Roberto S. Bigonha[2], and Marcelo A. Maia[3]

[1] Universidade Federal de Viçosa, email: vladimir@dcc.ufmg.br

[2] Universidade Federal de Minas Gerais, email: bigonha@dcc.ufmg.br

[3] Universidade Federal de Ouro Preto, email: marcmaia@dcc.ufmg.br

Abstract. This paper presents an offline partial evaluator for Abstract State Machines. Self-application is possible by means of a simplified version of the partial evaluator 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

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input [10]. The main goal is efficiency improvement, so it is expected that the specialized program runs faster than the original one. Partial evaluators have been successfully built for functional [2, 12], logical [14] and imperative [1] languages.

An interpreter for a language L is usually a program with two inputs: a source program S, written in L, and the input data for S. Specializing the interpreter with respect to a given source program yields a compiled program, i.e., a program written in the interpreter's implementation language, with the semantics of the source program. A partial evaluator is also a program with two inputs: the program to be specialized and part of its input. So a partial evaluator itself can be specialized (self-application). The specialization of a partial evaluator with respect to an interpreter yields a compiler. Finally, specializing a partial evaluator with respect to itself yields a compiler generator.

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 used to describe the semantics of several programming languages [3, 7, 16]. The description usually consists of an interpreter for the programming language.

Huggins and Gurevich present an offline partial evaluator for ASM in [8, 9], which allows the specialization of conditional instructions and update blocks. This paper presents a partial evaluator which extends that work mainly in two aspects: it allows the specialization of user-defined functions (*derived functions* [4]) 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.

$out = [\![P]\!]_S(in_1, in_2)$ $P_{in_1} = [\![mix]\!]_L(P, in_1)$ $out = [\![P_{in_1}]\!]_T(in_2)$	$target = [\![mix]\!](int, P)$ $compiler = [\![mix]\!](mix, int)$ $cogen = [\![mix]\!](mix, mix)$
(a) Equational Definition of mix.	(b) Futamura Projections.

Fig. 1. Partial evaluation equations.

2 Partial Evaluation

A partial evaluator, when given a program and some of its input data, produces a so-called *residual* or *specialized program* [10]. In other words, a partial evaluator is a program specializer. The parts of the subject program's input data used in the specialization process are known as *static data*. The remaining input data are known as *dynamic data*.

A partial evaluator performs a mixture of code execution and code generation, so it is sometimes called *mix*. Let P be a program with two inputs, in_1 and in_2, written in a language S. Let $[\![P]\!]_S$ represent the semantics of P. Figure 1(a) presents an equational definition of a partial evaluator mix for S-programs.

The languages involved are S (the language of the programs processed by the partial evaluator), L (mix implementation language) and T (the language of the programs produced by mix). S and T are usually the same language. P_{in_1} is the *residual program*, i.e., the result of the specialization of P with respect to the first input in_1.

2.1 Compilation and Compiler Generation

Futamura was the first to realize that partial evaluation can be used for compilation and compiler generation [10]. Let int be an interpreter for a language L_{int}, written in a language S. Let mix be a partial evaluator for S-programs. The equations presented in Figure 1(b) are known as the *Futamura Projections*.

The First Futamura Projection shows compilation by the partial evaluation of an interpreter with respect to a given source program. The Second Futamura Projection shows how a compiler can be generated by the self-application of a partial evaluator. The Third Futamura Projection shows the generation of a compiler generator *cogen* by specializing a partial evaluator with respect to itself. The second and third equations require that mix be written in its own input language.

2.2 Online and Offline Partial Evaluation

The specialization process can be carried out either following the *online* or the *offline* approach. If the values computed during program specialization can affect

the execution flow of the partial evaluator, the strategy is online, otherwise it is offline [10]. In practice, in the online approach, specialization is performed in a single stage. Almost all offline partial evaluators, on the other hand, perform specialization in two stages.

The first stage of an offline partial evaluator is called *binding time analysis (BTA)*. An annotated program is produced, with all structures marked either as dynamic or static, according to their dependence on the input data. The second stage is the specialization itself. The annotations are strictly followed to produce the residual program.

The first partial evaluators were all online [10], but they did not produce good results on compiler generation by self-application. Offline partial evaluation was invented in 1984 and made self-application feasible in practice [11].

2.3 Partial Evaluation and ASM

Huggins and Gurevich present an offline partial evaluator for sequential ASM in [8, 9], written in C. It performs specialization of basic ASM rules, but self-application is not addressed.

Our partial evaluator has been written in Java. It performs partial evaluation of sequential ASM specifications containing basic rules and also *derived functions* [4]. 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.

In order to achieve the results of the Second Futamura Projection, we have built a second partial evaluator in the ASM language itself. We have called those two partial evaluators `Jmix` and `ASMix`, respectively. The partial evaluator `ASMix` is a simplified version of `Jmix` implementing only the specialization phase of an offline partial evaluation method.

We present `Jmix` in Section 3. An example of compilation using the First Futamura Projection is presented in Section 4. In Section 5, we discuss implementation issues related to `ASMix` and evaluate the results of compiler generation (Second Futamura Projection) using small interpreters. We have not investigated the application of the Third Futamura Projection yet.

3 An Offline Partial Evaluator for ASM

In this section, we present the techniques used to develop an offline partial evaluator for sequential ASM specifications. The terms *static* and *dynamic* are frequently used in partial evaluation, but they have different meanings in ASM. To avoid name conflicts with established notation, we will use *positive* instead of *static*, for the parts of the input used in the specialization process, and *negative* instead of *dynamic*, for the remaining input data. This notation has been suggested by Huggins and Gurevich [8].

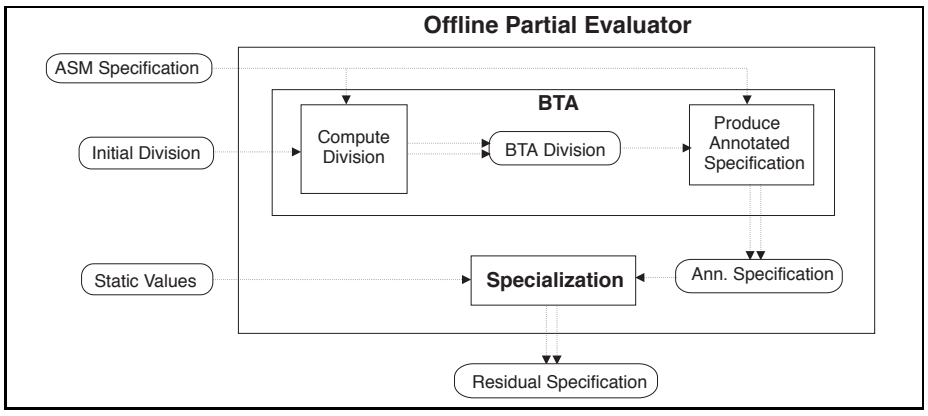

Fig. 2. Architecture of an offline partial evaluator for ASM.

The input for the partial evaluator are ASM specifications containing only basic rules. The input language includes *update instructions*, *conditional commands*, *blocks* and also *derived functions*. An abstract syntax is presented below, where the syntactical categories are represented in the following way: f (function names), t (terms), G (guards) and R (rules).

$$
\begin{aligned}
R &::= f(t_1, \ldots, t_r) := t \\
R &::= \texttt{if } G \texttt{ then } R_0 \texttt{ else } R_1 \\
R &::= R_1 \ldots R_k \\
&\texttt{derived function } f(\texttt{param}_1, \ \ldots, \ \texttt{param}_k) \ = \ t
\end{aligned}
$$

The partial evaluator assumes that input to ASM specifications is provided by *external (oracle) functions*. So each external function must be previously classified either as positive or negative, indicating which values are known in advance. The division of the external functions into positive and negative is called the *initial division*.

The partial evaluator receives an ASM specification to be specialized, an initial division and values for the positive external functions. The first stage is called *binding time analysis*. All functions used in the ASM specification are classified either as positive or negative, computing a *BTA division*. An annotated specification is generated, with all structures marked either as positive or negative, according to the BTA division. The second stage is called *specialization*. The annotations are strictly followed to produce a residual specification, using the values of the positive external functions. Figure 2 shows this process.

3.1 Binding-Time Analysis (BTA)

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 specification 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 external (oracle) functions. Initially, the user indicates which external functions are negative and which are positive. 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.

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 [10].

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.

Our division algorithm is not very sophisticated, so we allow users to help the classification of functions with annotations.

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

- a reference to an external function is classified as positive or negative according to the user annotations (initial division);
- 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 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.

When the classification algorithm ends, the entire specification 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 specification has the format `f(`$\texttt{exp}_1$`,...,`$\texttt{exp}_k$`)`, where each $\texttt{exp}_i$, $1 \leq i \leq k$, has already been classified either as positive or negative. Let `(`$\texttt{S}_1$`, ..., `$\texttt{S}_k$`)` represent the classification of those expressions, where $\texttt{S}_i$, $1 \leq i \leq k$, is either *positive* or *negative*. A different version of `f` is generated for each different tuple `(`$\texttt{S}_1$`, ..., `$\texttt{S}_k$`)`, 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 specification has been divided into positive and negative, and an annotated specification has been produced. The annotated specification 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 specification, in order to make the specializer simpler. These transformations increase the specification size, possibly exponentially. At the end of the process, the specification rules are 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 specification 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.

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 Section 3.3.

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.

3.3 Specializer

Specialization is executed over the specification 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 specification 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 already been 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 processed, transition rules are 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 $f(t_1, \ldots, t_k) := t_0$ generates code for an update instruction with all positive information within $t_0, \ldots, t_k$ 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.
- 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 rule 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 specification submitted to the partial evaluator. Each command `dummy` is replaced by `pe_flag` $:= i$, where i is the state referenced by it. The final residual specification 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 specification. 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 the residual functions associated with a derived function `f` usually have names with the format `f_`i`_`j in the residual specification. The indexes i and j identify the different versions of `f`.

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 `if pe_flag =` i `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 that can be executed simultaneously without modifying the semantics of the specification.

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 specification has been generated. Although the strategy chosen is 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 [10].

The number of residual derived functions is also usually very high. In many cases, a function is residualized as a simple expression, without recursive calls.

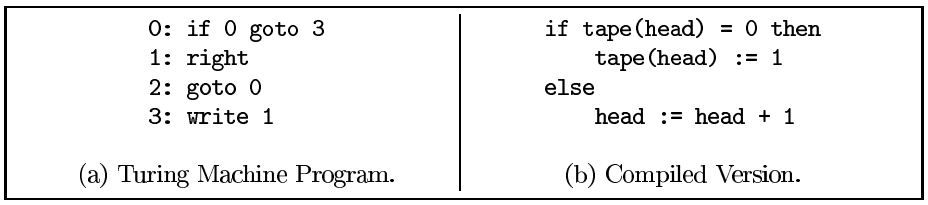

Fig. 3. Turing Machine Program and Compiled Version.

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 residual 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 residual 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.

4 Example: a Turing Machine Interpreter

We will show a very simple example of specializing an interpreter with respect to a specific source program, which we have borrowed from [10].

Consider a version of the *Turing Machine* with alphabet $\{0, 1, \mathtt{B}\}$ and a program $[\mathtt{I}_0\mathtt{I}_1 \ldots \mathtt{I}_n]$, where each $\mathtt{I}_i$, $0 \leq i \leq n$, is one of the following instructions: `right`, `left`, `write` a, `goto` i, `if` a `goto` i. 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 and `goto` i is an unconditional jump. The instructions are executed in sequence, unless a jump is executed.

In Figure 3(a), an example of a TM program is presented. The program changes the first 0 it finds in the tape to 1, or goes into an infinite loop if no 0 is found.

In Figure 4, we show the ASM transition rules which implement an interpreter to the language described above. The machine tape is represented by function `tape`. The program instructions are fetched by the external 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

```
if code(pc) = "RIGHT" then
    pc := pc + 1,
    head := head + 1
endif,
if code(pc) = "LEFT" then
    pc := pc + 1,
    head := head - 1
endif,
if code(pc) = "GOTO" then
    pc := par2(pc)
endif,
```

```
if code(pc) = "WRITE" then
    pc := pc + 1,
    tape(head) := par1(pc)
endif,
if code(pc) = "IFGOTO" then
    if par1(pc) = tape(head) then
        pc := par2(pc)
    else
        pc := pc + 1
    endif
endif
```

Fig. 4. Turing Machine Interpreter Written in ASM.

number of the current instruction is represented by `pc` and the tape head is represented by `head`.

Suppose that the interpreter of Figure 4 is submitted to the partial evaluator, to be specialized with respect to the TM program presented Figure 3(a). The external functions `code`, `par1` and `par2` represent the input TM program, so they are classified as positive. The BTA phase will classify `pc` as positive, while `tape` and `head` will be classified as negative. The result of the specialization is showed in Figure 3(b). We have omitted the initialization code.

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 in Figure 3(b) is exactly that produced by the partial evaluator.

5 Self-Application of the Partial Evaluator

As discussed in Section 2.3, we have built a simplified version of our partial evaluator in ASM itself. The main partial evaluator, which we have called `Jmix`, is divided into two phases: BTA and specialization. The simplified partial evaluator has been named `ASMix` and implements only the specialization phase.

5.1 The Simplified Partial Evaluator

ASM specifications are submitted to `ASMix` in an abstract syntax tree format. First, they are pre-processed as described in Section 3.2 and receive BTA annotations. The simplified partial evaluator `ASMix` performs only the specialization phase of the offline method, analyzing each command and following strictly the associated annotations.

When `ASMix` 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 ASMix, we have first built an ASM self-interpreter, i.e., an interpreter for ASM written in ASM itself. In Section 7 we discuss how this self-interpreter has been used to evaluate the performance of the main partial evaluator Jmix.

In this section, we describe how ASMix behaves while processing the three sections of its input specifications: function declarations, initialization and transition rules. A detailed description of ASMix can be found in [5].

Function Declarations The first task in ASMix is to determine a list *POS* of positive functions used to build the different states. This task is performed while ASMix process the function declarations of the input specification. 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 4, the only positive function that is updated by the transition rules is pc, so $POS = [\text{"pc"}]$.

Code is generated for the declarations of negative functions. The residual declarations have exactly the same format of the input specification, except for the annotations. All derived functions are residualized, because ASMix does not handle specialization of derived functions yet.

Initialization An ASM function FVal stores the values of the functions used in the input specification. FVal associates a function name and a list of values to another value. Using the example of Section 4, $\texttt{Tape}(1) = 0$ would be represented by ASMix as $\texttt{FVal}(\text{"Tape"}, [1]) = 0$. As FVal is updated with 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 $\texttt{FVal}(\text{"Tape"}, [1])$ 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.

Each state produced by ASMix is represented by a list of values, associated with the functions listed in *POS* by their positions. In the example of Section 4, the initial state would be the list [0], which means that initially pc has value 0.

Transition Rules The states produced during the specialization process are stored in two sets: *PENDING* and *MARKED*. The set *PENDING* contains the states for which residual rules have not been generated yet. The set *MARKED* indicates the states which have already been processed. Initially, *PENDING* contains only the initial state of the input specification.

Sates removed from *PENDING* are inserted in *MARKED* and processed. The specification transition rules are analyzed, with all positive information computed using the values of the current state removed from *PENDING*. The

process is very similar to that described in Section 3.3, where the specialization algorithm of `Jmix` is presented.

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 `Jmix`, but the code is generated in an abstract tree format, the same used for the input specification.

When a new state is produced, it is inserted in *PENDING* only if it is not in *MARKED*. The process stops when *PENDING* is empty.

The simplified partial evaluator still has some limitations. It does not specialize derived functions and most of the optimizations present in `Jmix` (see Section 3.4) have not been implemented yet.

5.2 Compiler Generation

Suppose that `int` is an interpreter for a language L_{int}, $\texttt{int}^{ann}$ is an annotated version of `int` and `source` is a program written in L_{int}. To satisfy the following equations, `Jmix` and `ASMix` should be equivalent:

$$\texttt{compiler} = [\texttt{Jmix}]_{Java}\,(\texttt{ASMix}, \texttt{int}^{ann}) \tag{1}$$

$$[\texttt{compiler}]_{ASM}\,(\texttt{source}) = [\texttt{Jmix}]_{Java}\,(\texttt{int}, \texttt{source}) \tag{2}$$

Equation 1 shows how a compiler can be generated by means of self-application of the partial evaluator. The simplified version `ASMix` is specialized with respect to an annotated version of `int`, yielding a compiler from L_{int} to ASM. The compiler is an ASM specification that receives a program P written in L_{int} and generates an ASM specification with the same semantics of P.

The left hand side of Equation 2 is the result of compiling `source` to ASM with a 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 2 will produce better results until optimizations are introduced in `ASMix`.

Because of the restrictions discussed above, we have only conducted compiler generation experiments with simple interpreters until now. The compiler generated by specializing `ASMix` 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 of the lack of transition compression optimizations in `ASMix`.

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

6 Interpreters Suitable for Partial Evaluation

The description of the semantics of programming languages in ASM usually consists of an interpreter for the language, written in ASM. Important concerns are correctness and readability [3, 7, 16]. The interpreters are provided on several abstraction levels which make them easier to understand, but they are usually not suitable for partial evaluation.

In this section, we discuss how small changes in a specification may achieve much better specialization results.

6.1 Case Study: Function Return in C

In the example showed in Section 4, all structures related to the TM program were classified as positive, so none of them appeared in the residual specification. 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 4: it indicates the current task to be performed. 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 those of Section 4.

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.

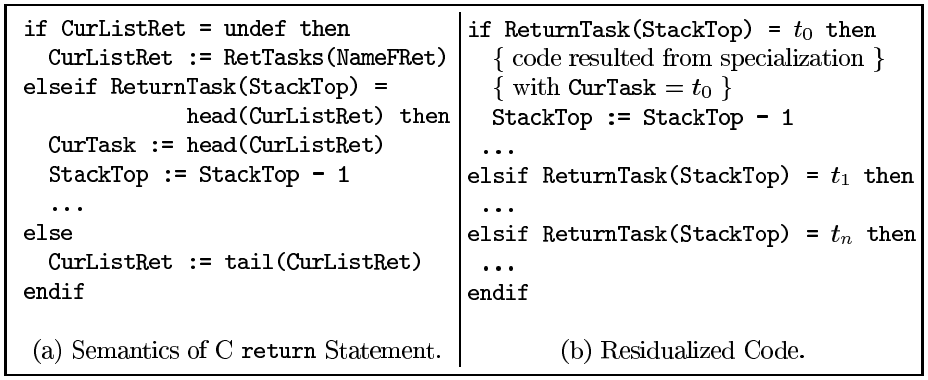

Fig. 5. Semantics of C `return` Statement and Residualized Code.

A solution for the problem above is to use a *pointwise division* in BTA [10]. In a pointwise division, functions can receive different classifications in different points of the specification 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 specification. The annotations in the specification produced after BTA would suffer some changes, and the implementation of the specializer would be more complicated.

6.2 Modifying the C Interpreter

Neither our partial evaluator, nor the one presented in [8] implements pointwise division. In the example of Section 6.1, it is possible to achieve good specialization results without pointwise divisions. All that is necessary is to change slightly the C interpreter. The changes we propose will make BTA classify `CurTask` as positive.

We will use $[e_1 \ldots e_n]$ to denote a list with n elements. The associated operations on lists are `head` and `tail`.

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 previously computed. The list of possible next tasks always takes on finitely many values, so we say that it has *bounded static variation* [10].

Let `RetTasks` 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 `RetTasks("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 `NameFRet` be an ASM function that represents the name of the C function which is terminated by the current C `return` statement. Part of the new code for the semantics of the C `return` statement is presented in Figure 5(a). The ASM function `CurListRet` stores the current list of the possible next tasks. We assume that

`RetTasks` is well built, so it is not necessary to include code to handle the case where `CurListRet` becomes empty.

Note that now `CurTask` is updated with values from `CurListRet`. The function `RetTasks` 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 trick of exploiting *statically bounded values* can be used in many other situations. It is so commonly used in partial evaluation that it is known as *The Trick* [10]. The residual code, as presented in Figure 5(b), will consist of a sequence of conditional commands, each testing a possible value for `ReturnTask(StackTop)`.

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 with good compilation results.

7 Conclusions and Future Work

In order to evaluate the performance of a partial evaluator, a self-interpreter test is suggested in [10]. 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 5.1, we have built a self-interpreter for ASM. The self-interpreter has been specialized with respect to many long ASM specifications. The residual specifications produced have been exactly the same given as input, except for some function renaming.

Results similar to those presented in Section 4 have been achieved with more complex languages. Compilation of the entire semantics of C and a small subset of Java, using the First Futamura Projection, have produced very good results. To accomplish this, the techniques presented in Section 6 have played a fundamental role. On the other hand, experiments with compiler generation using the Second Futamura Projection have not produced good results yet.

The Third Futamura Projection shows how a compiler generator *cogen* can be generated using self-application of a partial evaluator. For strongly typed languages, better results in compiler generation may be achieved using a hand-written *cogen* instead of a partial evaluator. In this case, *cogen* is known as a *generating extension generator* [15]. We intend to build a generating extension generator *cogen* for a strongly typed version of the ASM language. With this new approach, we expect much better results on semantics-directed compiler generation using ASM and partial evaluation techniques.

Partial evaluation of concurrent and parallel programs may also be a great source of research. Our future plans include extending the partial evaluator to deal with ASM extensions such as Distributed [6] and Interactive ASM [13].

References

1. L. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 54–61. New Haven, CT: Yale University, June 1992.
2. L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.
3. 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.
4. 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.
5. V. O. Di Iorio and R. S. Bigonha. An ASM Implementation of a Self-Applicable Partial Evaluator. Technical Report LLP-004-2000, Programming Languages Laboratory, DCC, Universidade Federal de Minas Gerais, 2000.
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. J. Huggins. An Offline Partial Evaluator for Evolving Algebras. Technical Report CSE-TR-229-95, EECS Dept., University of Michigan, 1995.
10. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
11. N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.
12. J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.
13. 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.
14. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*. Berlin: Springer-Verlag, Jan. 1993.
15. T. Æ. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.
16. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.