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

An ASM Implementation of a Self-Applicable Partial Evaluator

by

Vladimir O. Di Iorio
Roberto S. Bigonha

LLP 004/2000

Caixa Postal 702
30.161-970 - Belo Horizonte
Minas Gerais - Brazil
February 2000

Abstract

Partial evaluation is a technique for specializing programs with respect to parts of their input. We describe an offline partial evaluator for Abstract State Machines, written in the ASM language itself. The partial evaluator receives an input ASM specification and generates a residual ASM specification. Suitable internal representations for these specifications are presented. The partial evaluator is self-applicable, so some problems related to self-application are discussed. We show how compiler generation can be achieved using the partial evaluator and an interpreter written in ASM.

1 Introduction

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input [4]. 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, 11, 14], logical [16] and imperative [1, 15] 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.

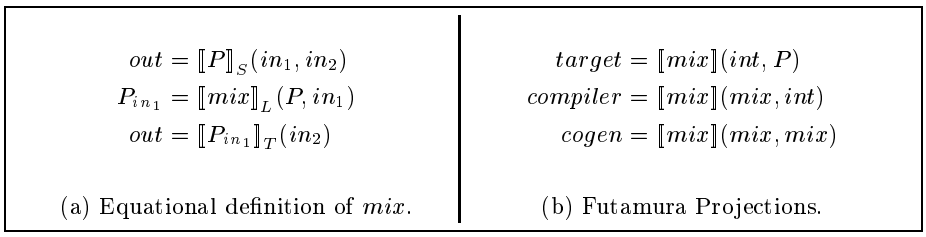
Abstract State Machines are a formal specification method created by Yuri Gurevich with the goal of simulating algorithms in a direct and coding-free way [7]. ASM has been used to describe the semantics of several programming languages [3, 8, 18]. The description usually consists of an interpreter for the programming language.

We present an offline partial evaluator for Abstract State Machines, written in the ASM language itself. The description can be seen as a formal specification of a partial evaluator for ASM. The partial evaluator is self-applicable, allowing compiler generation.

2 Partial Evaluation

A partial evaluator, when given a program and some of its input data, produces a so-called *residual* or *specialized program* [12]. 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.



(a) Equational definition of mix.

(b) Futamura Projections.

Fig. 1. Partial evaluation equations.

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 [12]. 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 [12]. 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 [12], 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 [13].

2.3 Partial Evaluation and ASM

Huggins and Gurevich present an offline partial evaluator for sequential ASM in [9, 10]. It performs specialization of basic ASM rules, but self-application is not addressed.

In [6] we present an offline partial evaluator for ASM, written in Java, which performs specialization of ASM rules and also user-defined recursive functions. We have also implemented another partial evaluator mix_{ASM}, written in ASM itself, which makes self-application possible.

In this work, we describe the partial evaluator mix_{ASM}, which was not discussed in detail in [6]. Special attention is devoted to the problems related to self-application.

3 An Offline Partial Evaluator for ASM

The architecture of an offline partial evaluator for ASM specifications is presented in this section. The terms *static* and *dynamic* have different meanings in ASM. To avoid name conflicts with established notation, we will use *positive* instead of *static* and *negative* instead of *dynamic*. This notation has been suggested by Huggins and Gurevich [9].

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

The partial evaluator receives as input the ASM specification to be specialized, the initial division and the values for the positive external functions. The first stage of the partial evaluator is called *binding time analysis*. All functions used in the ASM specification are classified either as positive or negative, computing a *BTA division*. Then, an annotated specification is generated, with all strutuctures marked either as positive or negative, according to the BTA division. The second stage is called

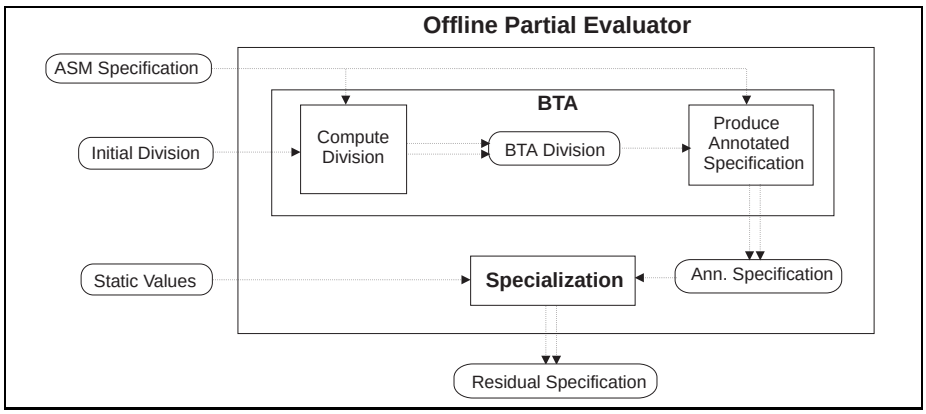


Fig. 2. Architecture of an offline partial evaluator for ASM.

specialization. The annotations are strictly followed to produce a residual specification, using the values of the positive external functions. Figure 2 shows a representation of this process.

Section 4 discusses an internal representation for ASM specifications. ASM rules describing *binding time analysis* and *specialization* are presented in sections 5 and 6, respectively. Self-application of the partial evaluator is discussed in Section 7.

4 Representation of Specifications

An ASM specification is the input of the partial evaluator, annotated specifications are produced by the BTA phase, and residual specifications are generated by the specialization phase. In this section, we discuss the way the partial evaluator encodes these syntactical objects into an internal representation. Three different representations will be presented: *input specifications*, *annotated* and *residual specifications*.

The partial evaluator deals only with basic ASM transition rules: *update instructions*, *block constructors* and *conditional constructors*. The abstract syntax of these rules is presented below, where f denotes function names, t denotes terms and R denotes rules:

$$
\begin{aligned}
R ::= \ & f(t_1, \ldots, t_r) := t \\
R ::= \ & R_1 \ldots R_k \\
R ::= \ & \texttt{if} \quad t \quad \texttt{then} \quad R_1 \quad \texttt{else} \quad R_2
\end{aligned}
$$

4.1 Input Specifications

To encode the input specifications, we will use definitions similar to those presented in [5]. An *input specification SPEC*, associated to an algebra $\mathcal{A}_{in}$, is defined as $SPEC = (\Upsilon_{in}, \mathcal{D}, Init, Prog)$, where

Υ_{in} is a vocabulary containing at least all function names occurring in $\mathcal{D}$, *Init* and *Prog*.

$\mathcal{D}$ is a set of function definitions. They may be *constructive definitions*, which the partial evaluator is able to evaluate, or *interface declarations*, defining externally alterable oracle functions.

Init is a block of update instructions which define part of the initial state S_0 of $\mathcal{A}_{in}$.

Prog is the transition rule.

An implicitly given part of the initial state S_0 of $\mathcal{A}_{in}$ results from default interpretations of predefined function names in Υ_{in} which are considered as built-in functions of the partial evaluator. This set of built-in functions contains at least the basic logic names. Part of the initial state is defined by *Init*. It is interesting to use a block of rules to represent *Init* because it can be specialized by the partial evaluator.

As in [5], we define a mapping "$[\![.]\!]$" that yields, for each syntactical object, the corresponding element of one of the following domains defined by the partial evaluator: *FNAME* (function names), *TERM* (terms), *RULE* (transition rules) and *DEF* (abstract function definitions). We assume that the input specification is syntactically correct and the encoding has been carried out by a preprocessing step.

For function names f in Υ_{in}, we have $[\![f]\!] = \psi(f)$, where ψ is an injective function from Υ_{in} into *FNAME*. For each function name f in Υ_{in}, we assume that Υ_{PE}, the vocabulary of the partial evaluator, contains a nullary function name "f" :*FNAME* such that "f" $= \psi(f)$.

The elements of the domain *TERM* are defined by means of a *term* constructor function. Using this constructor, different terms are inductively mapped to different elements of *TERM* in the following way:

$$term : (FNAME \times TERM^*) \rightarrow TERM$$
$$[\![f\,(t_1, \ldots, t_r)\,]\!] = \quad term\,([\![f]\!], \langle [\![t_1]\!], \ldots, [\![t_r]\!] \rangle)$$

The mapping of transition rules can be defined in a similar way, using the following constructor functions, which are used to encode the *Init* and

Prog components of *SPEC*:

$$update\_instr : (TERM \times TERM) \rightarrow RULE$$
$$block\_cons : RULE^* \rightarrow RULE$$
$$cond\_cons : (TERM \times RULE \times RULE) \rightarrow RULE$$

4.2 Annotated Specifications

Annotated specifications are produced by the BTA phase. They are very similar to input specifications, except that a BTA tag is associated to each syntactical object. BTA tags can assume one of the values: *BTAPOS* and *BTANEG*. The value *BTAPOS* indicates that the syntactical object can be computed at specialization time, while *BTANEG* indicates a "residualizable" object.

The universe *TBTA* represents the BTA tags. Annotated terms and rules are elements of the universes *TTERM* and *TRULE*, where "*T*" means "tagged". Constructors including annotations are described below. A special constructor *lift* is used to indicate a positive term that occurs in a negative context (see Section 5).

$$tterm : (TBTA \times FNAME \times TTERM^*) \rightarrow TTERM$$
$$tupdate\_instr : (TBTA \times TTERM \times TTERM) \rightarrow TRULE$$
$$tblock\_cons : TRULE^* \rightarrow TRULE$$
$$tcond\_cons : (TBTA \times TTERM \times TRULE \times TRULE) \rightarrow TRULE$$
$$lift : TTERM \rightarrow TTERM$$

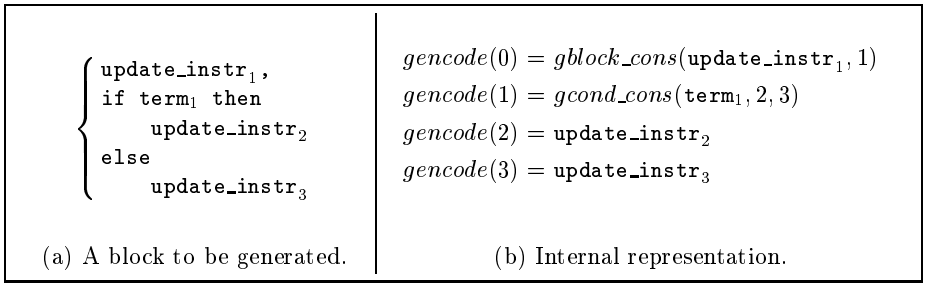
4.3 Residual Specifications

Residual rules are elements of the universe *GRULE*. These elements can be defined by the previous constructors *term* and *update_instr*, together with the new constructors *gblock_cons* and *gcond_cons*:

$$gblock\_cons : (null \cup (GRULE \times GVAL)) \rightarrow GRULE$$
$$gcond\_cons : (TERM \times GVAL \times GVAL) \rightarrow GRULE$$

A residual block can be either an empty block (*null*) or a residual rule together with a *GVAL* value that indicates the next rule in the sequence. A residual conditional rule is represented by the guard (a term) and two *GVAL* values that indicate the *then* and *else* rules.

A *gencode* function will be used by the partial evaluator to build links among the generated rules. An example is presented in Figure 3, where



(a) A block to be generated. (b) Internal representation.

Fig. 3. Example of use of the *gencode* table.

`if` $(\exists x \in CSET)$ `then`
`choose` $x \in CSET$
$CSET(x)$`:=` $false$`,`
`ProcessBlock(`x`)`
	`if` $\neg(\exists x \in CSET)$ `then`
	`if change then`
	$CSET$(`prog`)`:=` $true$`,`
	`change:=` $false$

Fig. 4. Rules for computing the BTA division.

GVAL is *INT*, for simplicity. Figure 3(a) shows a block with two rules. In Figure 3(b), the table *gencode* is used to store this block.

5 Binding Time Analysis

The binding time analysis phase is divided into two sequential steps. The first step computes a division of all functions into either positive or negative. In the second step, an annotated specification is generated.

5.1 Computing a BTA Division

Figures 4 and 5 show the rules that formalize the computation of the BTA division. Initially, all functions of the input specification are classified as positive, except the oracle (external) functions. All update instructions $f(t_1, \ldots, t_r) := t$ are sequentially examined. If t or any $t_i, 1 \leq i \leq r$ is negative, then f must also be classified as negative. The division algorithm iterates until a fixpoint is reached.

A unary relation $CSET : RULE \rightarrow BOOL$ is used, as in [5], to identify the instances of subrules that are being considered in a given step. The functions `init` and `prog` extract the initial and the transition rules of the input specification, respectively. The function `bta_val` associates, to each function f in Υ_{in}, a *TBTA* value (*BTAPOS* or *BTANEG*). The boolean nullary function `change` is used to determine when a fixpoint is reached.

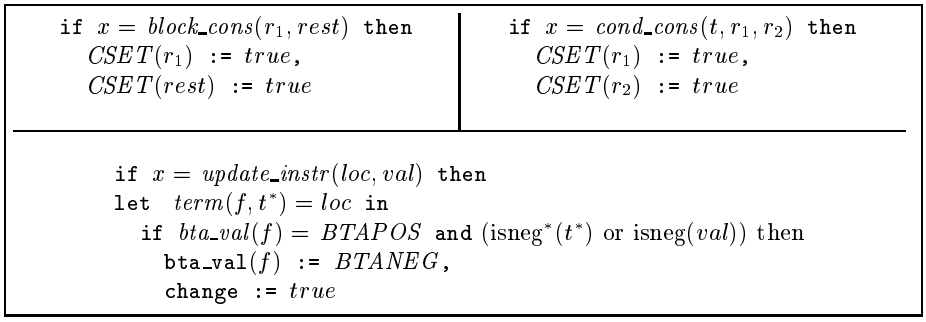


Fig. 5. Rule `ProcessBlock(`x`)`.

The function $\texttt{isneg}(t)$ indicates if the term t is negative, and $\texttt{isneg}^*(t^*)$ indicates if one of the terms of the list t^* is negative.

For each function f in Υ_{in}, $\texttt{bta\_val}(f)$ is initialized to $BTAPOS$, except for the oracle functions, whose classification is given as input. Other initial values: $CSET = \{\texttt{init}, \texttt{prog}\}$ and $\texttt{change} = false$.

5.2 Generating an Annotated Specification

After computing a BTA division, an annotated specification is generated. The functions used to build the annotated rules are `gentr` and `gentt`, whose definition is presented in Figure 6. The function `gentr` describes a mapping from rules to annotated rules, and `gentt` describes a mapping from terms to annotated terms. A third function $\texttt{gentt}^*$ is applied to lists of terms.

If a positive term occurs in a negative context, the result of the computation must be residualized. To indicate this situation to the specialization phase, a *lift* constructor is used in the annotated representation. The function `gentt` uses its second argument to determine if a *lift* is necessary.

Observe that an annotated block constructor is always generated, even when a conditional constructor or update instruction is processed. The only reason for this is to simplify the specialization algorithm.

6 Specialization

The specialization algorithm uses the annotated specification produced by the BTA phase and the values of the positive external functions to generate a residual specification. The technique used is *polyvariant specialization*. The process consists on computing the set of all reachable *specialized program points* [12]. A specialized program point, in this case,

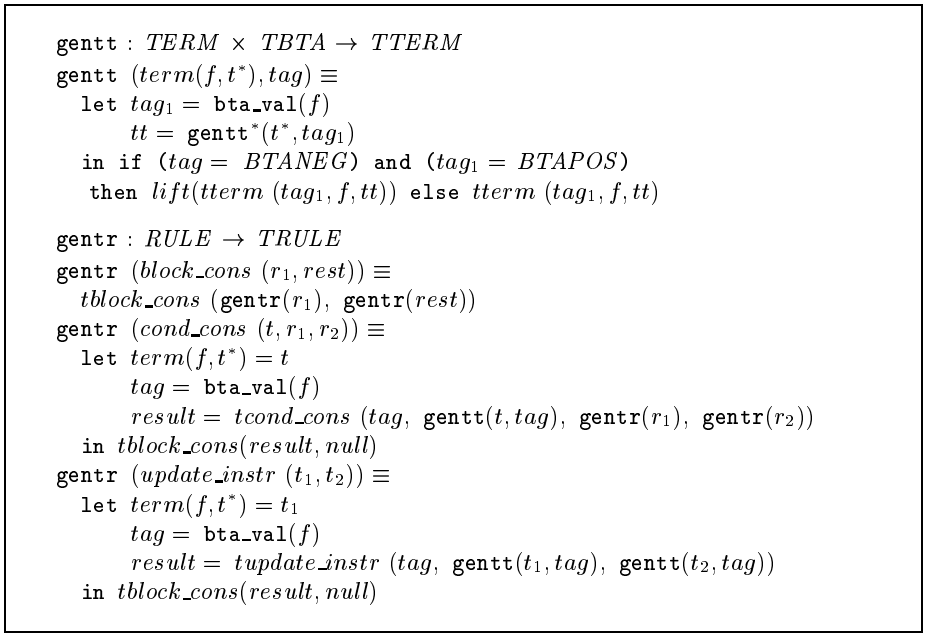


Fig. 6. Functions `gentt` and `gentr`.

is identified by a sequence of values for the positive functions. Each different sequence of values is a different program point. These sequence of values will be designated as *positive states*.

The initial positive state, which is defined by the initial values of the positive functions, is inserted into a set *PENDING*. This set comprises the positive states that have not been processed yet. On each iteration, an element is picked up from *PENDING* and processed.

When a positive state is processed, an associated code is generated and new positive states are produced. Those not processed yet are inserted into *PENDING*. This process is carried out until *PENDING* becomes empty. At this point, a generated rule R_k is associated to each different positive state k. The residual specification will have an additional function `CURSTATE` and its residual transition rule will be a block of rules of the form "`if CURSTATE` $= k$ `then` R_k".

Figure 7 shows ASM rules that describe the specialization algorithm. A unary relation *GSET* identifies the instances of subrules that are being considered in a given step, in a way similar to *CSET* in Section 5. The difference is that elements of *GSET* are a tuple formed by a rule, an integer number used as a link by the generated code, and a set of collected updates. The initial value of *GSET* is $\{\langle \texttt{init}, 0, null\rangle\}$. A set

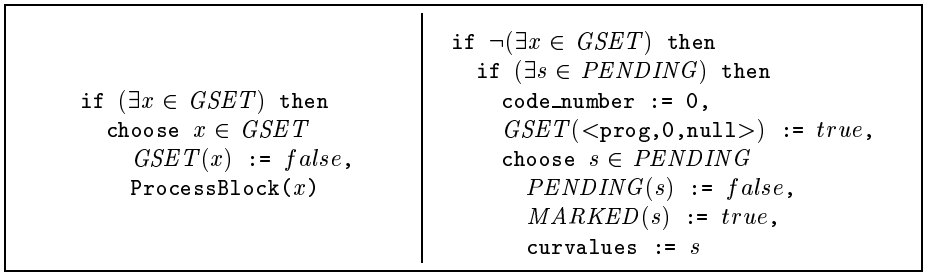


Fig. 7. Specialization Rules.

MARKED, initially empty, identifies the positive states that have already been processed. The function `codenumber` is used by the residual code. The function `curvalues` represents the current positive state being processed, and its initial value is a positive state in which the functions are undefined at all points. The functions `init` and `prog` now denote the initial and transition rules of the **annotated** specification.

Before showing the complete description of the specializer, we discuss how positive states are represented and computed.

6.1 Representation of States

A universe $VALUE = BOOL \cup \{undef\} \cup \ldots \cup N$ provides interpretations for standard function names. Let `posfuncs` denote the list of function names classified as positive by BTA, not including the external functions. If `posfuncs` $= \langle f_1, \ldots, f_k \rangle$, then a positive state $ps \in PSTATE$ is a list $\langle fv_1, \ldots, fv_k \rangle$, where fv_i represents a *function value* for $f_i, 1 \leq i \leq k$. A *function value* is a sequence of pairs $\langle VALUE^*, VALUE \rangle \in FVALUE$, representing the finitely many updated values of a function.

To build new positive states, the partial evaluator must compute values using positive terms of the annotated specification. To specify the evaluation of these terms, the following auxiliary functions are introduced, which are similar to those presented in [5]:

- $g_1, \ldots, g_r$ represent the predefined functions.
- $apply : DEF \times VALUE^* \rightarrow VALUE$ is a function which produces values for the functions defined by the input specification, given a function definition and a sequence of arguments. This definition can be constructive or an interface for an external function.
- $cont : FNAME^* \times FVALUE^* \times LOCATION \rightarrow (VALUE \cup nil)$ is a function which produces values, given a sequence of function names, a

positive state and a location, in the following way: $cont\,(pf, ps, \langle f, t^*\rangle)$ identifies the position of the function name f in the sequence pf, obtaining the correctly associated element $fv \in FVALUE$ from ps. Using fv and the terms t^*, a value (possibly $undef$) is produced. The special nullary function nil indicates that the function name f is not in the sequence pf.

Interpretations to the annotated terms are provided by the evaluation function $Val : FNAME^* \times FVALUE^* \times TTERM \rightarrow VALUE$ using values defined by a positive state and its associated list of positive function names. This function, which will be applied only to positive terms, is defined by the equation

$$
\begin{array}{l}
Val\ (pf, ps, tterm\,(tag, f, \langle t_1, \ldots, t_n\rangle)) = \\
\quad \texttt{let}\ \langle \bar{x}\rangle = \langle\, Val\,(t_1), \ldots, Val\,(t_n)\rangle\ \texttt{in} \\
\quad \texttt{if}\ f = \text{“}g_1\text{”}\ \texttt{then}\ g_1\,(\bar{x}) \\
\quad \ldots \\
\quad \texttt{else if}\ f = \text{“}g_r\text{”}\ \texttt{then}\ g_r\,(\bar{x}) \\
\quad \texttt{else if}\ cont\,(pf, ps, \langle f, \langle \bar{x}\rangle\rangle) \neq nil\ \ \texttt{then}\ cont\,(pf, ps, \langle f, \langle \bar{x}\rangle\rangle) \\
\quad \texttt{else if}\ def\,(f) \neq undef\ \text{then}\ apply\,(def\,(f), \langle \bar{x}\rangle) \\
\quad \texttt{else}\ undef
\end{array}
$$

When a negative term is processed by the partial evaluator, the residual generated term must have all positive information computed. The function $Reduce : FNAME^* \times FVALUE^* \times TTERM \rightarrow TERM$ produces these residual *reduced terms*. Like *Val*, *Reduce* uses values defined by a positive state and its associated list of postive function names. To produce a residual term, negative annotated terms are simply converted to terms without annotations. Positive terms are evaluated using the function *Val*. When a *lift* constructor is found, denoting a positive term inserted in a negative context (see Section 5), the positive value is computed and converted to a *term* constructor.

6.2 Processing Blocks

Figure 8 shows ASM rules to process the elements inserted in *GSET*. These elements are formed by a block of annotated rules, an integer number used in code generation, and a sequence of collected updates. The sequence of collected updates is formed only by syntactical objects (each update is represented by a pair of *TTERM* elements).

On each iteration, the first annotated rule of the block is processed, inserting new elements in *GSET* (see the description of `ProcessRule`).

```
let <thisblock, cn, updates>= x in
  if thisblock = tblock_cons(r1, rest) then
    ProcessRule (r1, cn, rest, updates)
  else    //...thisblock is an empty block
  let newvalues = compute_updates (posfuncs, curvalues, updates) in
    gencode(<curvalues,cn>) := gen_next_step (newvalues),
    if ¬(newvalues ∈ MARKED) then
      PENDING(newvalues) := true
```

Fig. 8. Rule `ProcessBlock(`x`)`.

If the block is empty, a new positive state is produced using its collected updates and appropriate code is generated.

A new positive state `newvalues` is computed by `compute_updates`, using the values defined by the current positive state `curvalues` and the sequence of syntactical objects representing the collected updates of the block. We do not present here the description of `compute_updates`, but it may be easily defined using the function *Val*, previously described. If the new positive state has not been processed yet, it is inserted in *PENDING*.

The function `gencode`, as explained in Section 4, associates a residual rule to each processed block. A block is identified by the value of the current positive state and a code number `cn`. The function `gen_next_step(v)` generates the code "`CURSTATE := v`", which defines the flow of control in the residual transition rule (see Section 6.5).

6.3 Processing Conditional Constructors

Annotated conditional constructors *tcond_cons* $(tag, cond, rthen, relse)$ are processed by the ASM rules shown in Figure 9. To append two blocks of rules,.the function `merge_blocks` is used.

If the rule is positive, then the condition is evaluated using the function *Val*. Depending on the result (*true* or *false*), the new subrule to be processed in the next step is *rthen* or *relse*, merged with the rest of the current block.

If the rule is negative, two new subrules are inserted in *GSET*. Observe that new code numbers are assigned to these subrules. The generated residual code is a conditional constructor which is linked to the code numbers of the new subrules. All positive information of the residual condition is computed by the *Reduce* function.

```
if r1 = tcond_cons (tag, cond, rthen, relse) then
  let merged_rthen = merge_blocks (rthen, rest),
      merged_relse = merge_blocks (relse, rest)
  in if tag then
    if Val (posfuncs, curvalues, cond) then
      GSET (<merged_rthen, cn, updates>) := true
    else GSET (<merged_relse, cn, updates>) := true
  else
    GSET (<merged_rthen, codenumber+1, updates>) := true,
    GSET (<merged_relse, codenumber+2, updates>) := true,
    codenumber := codenumber + 2,
    gencode(<curvalues,cn>) := gcond_cons (
      Reduce (posfuncs, curvalues, cond),
      <curvalues,codenumber+1>, <curvalues,codenumber+2>)
```

Fig. 9. ProcessRule(r_1, cn, *rest*, updates), for conditional constructors.

6.4 Processing Update Instructions

Figure 10 shows ASM rules to process annotated update instructions $tupdate\_instr\,(tag, t_1, t_2)$.

If the rule is positive, the new subrule inserted in *GSET* is formed by the rest of the current block, with $\langle t_1, t_2 \rangle$ added to the sequence of collected updates. The code number is the same of the current block, because no residual code is generated. Observe that t_1 and t_2 are syntactical objects.

If the rule is negative, the new subrule inserted in *GSET* is formed by the rest of the current block, with the same sequence of collected updates. A new code code number is assigned to this subrule, because a residual block is generated using the current code number. A reduced (positive information computed) version of the update instruction is the first rule of the residual generated block, which is linked to the new subrule to be processed.

6.5 The Residual Transition Rule

After the last step of the specialization algorithm, the links established by the function `gencode` can be used to build the residual transition rule. Each positive state $ps \in MARKED$ has an associated residual rule R_{ps} defined by `gencode`(ps).

An additional function `CURSTATE` : *PSTATE* defines the flow of control in the residual specification. The initial value of `CURSTATE` is a positive

```
if r1 = tupdate_instr (tag, t1, t2) then
  if tag then let newupdates = cons (< t1,t2 >, updates) in
    GSET (<rest, cn, newupdates>) := true
  else
    GSET (<rest, codenumber+1, updates>) := true,
    codenumber := codenumber + 1,
    gencode(<curvalues,cn>) := gblock_cons (
      update_instr (Reduce (posfuncs, curvalues, t1),
                    Reduce (posfuncs, curvalues, t2) ),
      <curvalues,codenumber+1>)
```

Fig. 10. `ProcessRule(`r_1`, cn, `*rest*`, updates), for update instructions.`

state in which the functions are undefined at all points (the same initial value of `curvalues` in the specializer). The residual transition rule is a block of rules of the form "`if curstate =` ps `then` R_{ps}", for each $ps \in MARKED$. The flow of control is determined by the assignments to `CURSTATE` which are generated when an empty block is processed by the specializer (see Figure 8).

7 Self-Application and Compiler Generation

The offline approach simplifies the process of self-application because it divides partial evaluation into two separated phases. The partial evaluator for ASM ($\texttt{mix}_{\texttt{ASM}}$) is composed of two separated programs: *BTA* (binding time analysis) and *spec* (specialization). Using these programs, the Second Futamura Projection can be rewritten in the following way:

$$\begin{aligned} spec^{ann} &= [\![BTA]\!](spec, div_{spec}) \\ int^{ann} &= [\![BTA]\!](int, div_{int}) \\ compiler &= [\![spec]\!](spec^{ann}, int^{ann}) \end{aligned}$$

where p^{ann} denotes an annotated version of a program p and div_p denotes the division of the external functions of p into static (positive) and dynamic (negative).

The compiler generation process using the offline partial evaluator for ASM is performed in three steps. First, an interpreter for a language L is written in ASM. Second, an annotated version of this interpreter is generated using the BTA algorithms described in Section 5. Finally, a previously annotated version of the *spec* specialization algorithm (described in Section 6) is specialized with respect to the annotated interpreter. A

compiler from L to ASM is generated. Note that *BTA* is not included in the self-application of the specializer.

The specialization algorithm described in Section 6 processes only basic ASM rules. The text of the description uses simple pattern matching constructs and `let` expressions to enhance readability, but these structures can be easily translated into appropriate selector functions that extract the desired components. The `choose` constructor is also used, but its semantics is essentially deterministic. It can be translated into operations that extract elements from a list. So the entire description can be translated into a specification that uses only basic ASM rules, making self-application possible.

Not all programs are suitable for partial evaluation. The structures depending on static (positive) and dynamic (negative) values must be carefully separated. The positive structures will be computed during specialization time, and will not appear in the residual code. To show that the specializer presented in Section 6 is suitable for partial evaluation, it is necessary to analyze its annotated version, produced by submitting it to the BTA algorithm.

The specializer has two input data: an annotated specification and the values of its positive external functions. To access the annotated specification, external functions like `init` (initial rule), `prog` (transition rule) and `def` (definitions) are used. If the specializer is specialized with respect to the first input, these external functions are marked as positive. The second input, the values of the positive external functions, is accessed by means of the auxiliary function `apply` (see Section 6.1). This function is marked as negative.

We present now the application of the BTA algorithm to the specializer and the initial division described above. The function *Val* depends on the function apply, so any occurrence of *Val* is negative. The function `compute_updates` depends on *Val*, so it is also negative. Other negative functions: `curvalues`, `gencode` and the sets *PENDING* and *MARKED*. On the other hand, the components of the elements inserted in *GSET* are all positive: the subrules and the collected updates are extracted from the annotated program, and the code number depends only on itself. So *GSET* is classified as positive. The reason of using only syntactical objects to represent the collected updates now becomes clear (another possibility would be using computed values, but it would make *GSET* negative).

A residual compiler generated by the Second Futamura Projection has no occurrence of the positive functions enumerated above. The positive

and negative structures are satisfactorily separated, so the specializer can be considered suitable for self-application.

8 Conclusion

Self-application of a partial evaluator allows compiler generation, but imposes additional requirements on the specification. The partial evaluator must process its own text. Static (positive) and dynamic (negative) structures must be carefully separated.

The ASM specification of the partial evaluator presented is very simple and is few lines long. We have decided to use the offline approach because it simplifies self-application [12]. We have shown that it is possible to describe a self-applicable partial evaluator using only basic ASM rules. The ASM model has proved to have the necessary expressive power to specify the partial evaluator in a elegant way.

The offline partial evaluator for ASM differs from partial evaluators for imperative and functional languages in several aspects. For example, it must deal with parallel updating of functions and has a different concept of *specialized program points*.

In imperative languages, a specialized program point is defined by a pair $\langle l, vv \rangle$, where l is a program label and vv represents the values of the static variables. In functional languages, specialized program points are defined by a function name and the values of its static arguments. In ASM specifications, on the other hand, the entire transition rule is processed for each different set of positive (static) functions. So a specialized program point is represented only by the values of the positive functions. Processing the entire transition rule multiple times brings additional difficulties, because several new specialized program points may be produced at each iteration.

The parallel updating of functions is an important feature of ASM specifications. The partial evaluator is supposed to maintain a set of collected updates that are to be fired in parallel, when a new specialized program point is built.

We have found some difficulties to represent the code generated by the specializer. Residual generated specifications could be internally represented with the same structures used for input specifications. However, the top-down generating process used by the specializer makes the use of those structures unsuitable, specially for block and conditional constructor rules. The technique described in Section 4.3 is appropriate for top-down code generation.

Many optimizations may be implemented in the partial evaluator. The most important is *transition compression* [12]. With transition compression, compatible rules can be merged and unnecessary residual rules can be eliminated, producing a more efficient residual specification.

Some experiments involving compilation and compiler generation using ASM and partial evaluation are described in [6]. Using a powerful partial evaluator for ASM, implemented in Java, interpreters written in ASM are specialized with respect to source programs. Good results in compilation are reported, in experiments involving a simple Turing Machine interpreter and an interpreter for a subset of C.

The partial evaluator described in this document, which we call $\texttt{mix}_{\texttt{ASM}}$, was used in compiler generation experiments. The partial evaluator written in Java was used to specialize $\texttt{mix}_{\texttt{ASM}}$ with respect to a simple Turing Machine interpreter. The residual compiler processes Turing Machine programs and generates ASM code. The results, however, are not satisfactory, because of the lack of optimizations in $\texttt{mix}_{\texttt{ASM}}$.

A more promising alternative for compiler generation is being considered. A statically typed ASM language, called Machĭna, is presented in [17]. An efficient compiler from Machĭna to C is under construction. Another important work is a compiler generator *cogen*, which uses partial evaluation techniques. An integrated system will generate compilers in C, using as input interpreters written in Machĭna.

References

1. L. Andersen. C program specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 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. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 493–501. ACM, New York: ACM, 1993.
5. G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. K. Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191–214. Springer, 1996.
6. V. O. Di Iorio, R. S. Bigonha, and M. A. Maia. A Self-Applicable Partial Evaluator for ASM. Technical Report LLP-11-99, Programming Languages Laboratory, DCC, Universidade Federal de Minas Gerais, 1999.
7. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

8. 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.
9. 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.
10. J. Huggins. An Offline Partial Evaluator for Evolving Algebras. Technical Report CSE-TR-229-95, EECS Dept., University of Michigan, 1995.
11. N. Jones, C. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pages 49–58. New York: IEEE Computer Society, 1990.
12. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.
13. 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.
14. J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.
15. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.
16. 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.
17. F. Tirelo, R. Bigonha, M. A. Maia, and V. Iorio. Machīna: A Linguagem de Especificação de ASM (in portuguese). Technical Report 08/1999, Laboratório de Linguagens de Programação, Universidade Federal de Minas Gerais, 1999.
18. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.