Removing Value Encoding using Alternative Values in Partial Evaluation of Strongly-Typed Languages

Denis Bechet

CRIN-CNRS & INRIA–Lorraine
Boîte Postale 239
54506 Vandœuvre-les-Nancy, France
e-mail: Denis.Bechet@loria.fr

Abstract. There is a main difference between a program which is interpreted by an interpreter written in a strongly-typed language and a compiled version. Such an interpreter usually uses a universal domain for the values it manipulates. A value encoding is necessary. A compiled program works directly on values. A layer of interpretation for value representation is inserted. On the other hand, a way to derive automatically a compiler from an interpreter is to use a partial evaluator applied to the interpreter and the interpreted program. This leads to a problem when we want that this technique removes all the layer of interpretation because value encoding must disappear. It is not the case for conventional partial evaluator. This paper proposes to introduce a new domain for partial evaluators called *alternative values* and a new algorithm of specialization (based on *events*) which can solve this problem of removing value encoding. We conclude by reporting a successful specialization of an interpreter written in a strongly-typed language by the partial evaluator LaMix which is based on those ideas.
keywords: partial evaluation, alternative values, type specialization, interpreters, strongly-typed language.

1 Introduction

Usually a compiled program is more efficient than interpreting it. A compiler removes the layer of interpretation. For interpreters written in a strongly-typed language, the difference is much more distinct because values manipulated during interpretation need to be encoded into a universal type. Three main differences separate a compiled program and its interpretation:

1. A compiler transforms source expressions into commands of the object language.
2. A compiler introduces variables but an interpreter uses a data structure for environments.
3. A compiler works on typed expressions but an interpreter needs to encode the values it manipulates.

In the framework of partial evaluation [6, 2], a typical application consists in specializing an interpreter when the interpreted program is known. This process looks like a compilation of the interpreted program into the language in which the interpreter is written [4, 12].

If we want that this mechanism acts as a compiler, it must solve the above points. This leads to the following operations.

1. Specialization of the eval loop of the interpreter with respect to the interpreted program. Control points of the interpreted program (a data) become points of the residual program.
2. Elimination of structures like environments which have to be replaced by variables or products of values (in particular, strings used to identify the variables of the interpreted program must disappear).
3. Specialization of values manipulated by the interpreter. Because interpreters usually use a universal datatype to describe values associated to variables or expressions, a compilation needs to introduced typed expressions.

The first problem has been solved by partial evaluation a long time ago. The idea consists in specializing the functions of the interpreter with respect to the value (and sub-values) of the data representing the interpreted program. The solution offered by partial evaluation is particularly simple because it just consists in propagating data coming from the interpreted program through the interpreter.

The second operation which transforms structures like environments into stores and references to new variables is solved by the simple mechanism shown above when the interpreter is written specially for this purpose (by dividing environments in two structures, one holding a list of identifiers, a second holding their values). More generally, this problem has been solved by the partial evaluators that can handle *partially known structures* [9] when environments are represented by association lists.

Practically no work in the framework of partial evaluation tackles the third problem. The technique of *tag removal* [7] consists in eliminating constructors when the value associated to an expression is always built with a unique constructor. Mogensen has also introduced the notion of constructor specialization [10] and [3] has extended his technique. New types are created but their partial evaluators either creat dead code or loop too frequently. This third problem is however very important if we want to have a residual program at least as efficient as a compiled version, especially when the interpreted program have several phases and use intermediary data structures. For instance, a lot of programs start by analyzing its input (a parser) then they compute a certain function on this internal representation, leaving an internal representation of the result. A last phase transforms this representation into more readable data (a printer). A partial evaluator must introduce specialized representations for those intermediary data.

Another reason for trying to solve this problem is due to Launchbury in [8] for the construction of self-applicable partial evaluators for strongly-typed languages. It is known that the self-application of a partial evaluator leads to compiler and compiler generator using the Futamura projections [4]. For this goal, a main problem is that a partial evaluator, like interpreters, uses a universal domain for representing values associated to expressions. In fact, it is the general problem of coding programs and data when they need to be manipulated. Sometime a double encoding is performed (for instance, for the self-application of a partial evaluator, the partial evaluator is encoded as a parameter of itself and the second argument which may be an interpreter is encoded twice). Eliminating internal encoding results in a better residual program.

This paper is organized as follows. After introducing partial evaluation and the problem for strongly-typed languages, Section 3 introduces the domain of *alternative values*. Its use and utility are explained in Sections 4 and 5. Section 6 introduces the problem of comparison between alternative values and Section 7 solves it by using a mechanism based on *events*. Section 8 shows a successful example of specialization using this technique. Finally, Section 9 concludes.

2 Partial Evaluation

Let us start by presenting briefly the concept of partial evaluation. Starting with a function of two arguments $f : A \times B \to C$ and a value $a \in A$, a partial evaluator returns a *residual program* defining a function $f_a : B \to C$ such that,

$$\forall b \in B \; f_a(b) = f(a, b)$$

The residual program for f_a is supposed to be an optimized version of the original one using the fact that the first parameter of f has been fixed to a. For instance, f may represent the function "power" which takes two integers x and n and returns x^n. A ML program (we use the dialect Caml-Light [13]) that computes such a function is:

```
let rec power x n = if n = 0 then 1 else x * (power x (n-1)) ;;
```

One can specialize this algorithm when n is known. For instance the cube of x is obtained for $n = 3$. By propagating n through `power`, a partial evaluator returns the residual program which defines the function `cube`:

```
let cube x = x * x * x ;;
```

It is obvious that `cube` is more efficient than calling `power` with its second parameter equals to 3. The fixed data is called *static* and the remainder *dynamic*. Here, n is static and x dynamic.

2.1 Partial Evaluators and Interpreters

This technique is particularly interesting when the specialized program is an interpreter and the static data is the interpreted program. Let us consider *mix* as a partial evaluator for programs written in $\mathcal{L}$, *int* as an interpreter for a language $\mathcal{S}$ defined by a $\mathcal{L}$ program and P as a $\mathcal{S}$ program. In fact *int* needs two arguments: the first one is a $\mathcal{S}$ program and the second one is the data for this program. If $\overline{int_P} = mix(\overline{int}, \overline{\overline{P}})$ [1] is the residual program obtained by specializing *int* for P, we have the equality:

$$[\![\overline{int_P}]\!]\,(d) = \left[\!\!\left[mix(\overline{int}, \overline{\overline{P}}) \right]\!\!\right](d) = int(\overline{P}, d)^2$$

Thus, int_P represents a "compiled" version of P in the language $\mathcal{L}$. This ability of partial evaluators to create automatically a compilation process using an interpreter is known since a long time [4, 12].

2.2 A Compiler from an Interpreter

Moreover, if the partial evaluator is written in the same language that it specializes, we can create a "compiler" for $\mathcal{S}$ programs if we apply it to itself and to *int*. If $\overline{comp_{int}} = mix(\overline{mix}, \overline{\overline{int}})$ then we have:

$$[\![\overline{comp_{int}}]\!]\,(\overline{\overline{P}}) = \left[\!\!\left[mix(\overline{mix}, \overline{\overline{int}}) \right]\!\!\right](\overline{\overline{P}}) = mix(\overline{int}, \overline{\overline{P}}) = \overline{int_P}$$

The function $comp_{int}$ gets a $\mathcal{S}$ program and returns an equivalent $\mathcal{L}$ program. It performs compilation of $\mathcal{S}$ programs into $\mathcal{L}$ programs.

As a consequence, if everything was perfect, all what we need to build a compiler for a language $\mathcal{S}$ into a language $\mathcal{L}$, is an interpreter for $\mathcal{S}$ programs written in $\mathcal{L}$ and a partial evaluator for $\mathcal{L}$ programs written in $\mathcal{L}$. We can summarize the motivation in this case by: it is usually easier to write an interpreter for a new language $\mathcal{S}$ (for instance, from denotational semantics) than to write a specific compiler for it. Moreover, a unique partial evaluator can build different compilers for various new languages (from several interpreters).

In practice, this process succeeds if *mix* is sufficiently powerful to eliminate the layer of interpretation introduced by *int*. The program *mix* has to solve the three problems introduced at the beginning of the paper. Here we are focused on the third problem: the removing of value encoding used by interpreters.

2.3 Type Specialization

To illustrate the significance of this problem, let us introduce a concrete example. Suppose we have a program that derives symbolic expressions. It takes, as input, a list of symbols representing a function (in prefix notation) and returns a list of symbols (with the same convention) representing the derivative of the input. This program may be split into four phases.

- Reading an input list of symbols and building a tree representing this expression (this is a parsing phase).

[1] An overlined function means a representation of its definition

[2] $[\![\overline{p}]\!]$ means the function defined by the program $\overline{p}$ (it is p)

– Deriving the abstract tree. This phase returns a new tree for the derivative.
– Simplification of the resulting expression (for instance, a multiplication by one may disappear).
– Transforming the abstract tree into a list of symbols (this is a printing phase).

This algorithm uses an internal representation for symbolic expressions on which the main process is performed. Now, let us suppose that this program P_{derive} is written in a language $\mathcal{S}$, that *int* is an interpreter for $\mathcal{S}$ programs written in $\mathcal{L}$ and *mix* is a partial evaluator for $\mathcal{L}$ programs. If the interpreter uses a universal datatype to represent all values it manipulates then the internal representation for symbolic expressions needs to be encoded in this universal domain. As a consequence, if the partial evaluator can not perform type specialization, the layer of value-encoding will stay in the residual program.

This fact is especially important if $\mathcal{L}$ is a strongly typed language because the interpreter needs to encode the values it calculates by adding tags giving the kind of value it manipulates. For instance, an interpreter may use this sum type for representing values:

```
type Values = Int of int
            | String of string
            | Cons of string * Values list
            | Tuple of Values list ;;
```

The first constructor `Int` says that it is an integer, the second `String`, a string, the third `Cons`, a constructor whose name is given (a string) and whose arguments are put in a list. The fourth constructor `Tuple` serves to introduce products.

Now, since *int* uses this type, the value that represents a particular symbolic expression must be coded. Suppose symbolic expressions are as follows:

$$\begin{array}{lll|l}
E \to & Z & & [\![Z]\!] = \lambda x.0 \\
\mid & S(e) & e \in E & [\![S(e)]\!] = \lambda x.1 + [\![e]\!](x) \\
\mid & X(e) & e \in E & [\![X(e)]\!] = \lambda x.x * [\![e]\!](x) \\
\mid & M(e) & e \in E & [\![M(e)]\!] = \lambda x. - [\![e]\!](x) \\
\mid & P(e_1, e_2) & e_1, e_2 \in E & [\![P(e_1, e_2)]\!] = \lambda x. [\![e_1]\!](x) + [\![e_2]\!](x) \\
\mid & I(e) & e \in E & [\![I(e)]\!] = \lambda x.1/[\![e]\!](x) \\
\mid & T(e_1, e_2) & e_1, e_2 \in E & [\![T(e_1, e_2)]\!] = \lambda x. [\![e_1]\!](x) * [\![e_2]\!](x)
\end{array}$$

The constructors are the constant zero, the successor, a variable (times an expression — for instance x^3 is $X(X(X(S(0)))))$, the opposite, the inverse, the addition and the multiplication.

To encode them in the type `Values`, a string is choose for each constructor and arguments are put in a list:

$Cons(``0", [])$	for	Z	$Cons(``S", [e])$	for	$S(e)$
$Cons(``x", [e])$	for	$X(e)$	$Cons(``-", [e])$	for	$M(e)$
$Cons(``+", [e1; e2])$	for	$P(e_1, e_2)$	$Cons(``/", [e])$	for	$I(e)$
$Cons(``*", [e1; e2])$	for	$T(e_1, e_2)$			

If *mix* does not perform type specialization those representations will stay in the residual program. Removing this layer of interpretation means that a new datatype for symbolic expression is introduced like this one:

```
type Expr = Zero
          | Succ of Expr
          | Var of Expr
          | Minus of Expr
          | Plus of Expr * Expr
          | Inverse of Expr
          | Times of Expr * Expr ;;
```

3 A New Domain of Specialization

Since we are interested by intermediary results returned by specialized functions, we need to describe this kind of values. For instance, the function of P_{derive} that parses the input returns values belonging to the domain D_{parse}:

$$\begin{array}{rll} D_{parse} \rightarrow & Cons(\text{“}0\text{”}, []) & \\ | & Cons(\text{“}S\text{”}, [d]) & d \in D_{parse} \\ | & Cons(\text{“}x\text{”}, [d]) & d \in D_{parse} \\ | & Cons(\text{“}-\text{”}, [d]) & d \in D_{parse} \\ | & Cons(\text{“}+\text{”}, [d_1; d_2]) & d_1, d_2 \in D_{parse} \\ | & Cons(\text{“}/\text{”}, [d]) & d \in D_{parse} \\ | & Cons(\text{“}*\text{”}, [d_1; d_2]) & d_1, d_2 \in D_{parse} \end{array}$$

Because *mix* must describe the value returned by the specialized version of *int* that corresponds to the parsing function of P, the domain of values that *mix* manipulated has to support this form of description.

In fact, each constructor are build by a particular expression during the specialization of *int*. So, we can index each constructor by the code that creates it. We suppose for the moment that the specialization of *int* corresponding to the parsing function creates only one instance for each kind of above structures. Thus there exists a unique place in the specialized functions of *int* that builds for each of the 7 constructors $Cons$, the 7 strings “0”, “S”,... and the different constructors of list (7 empty lists, 9 “cons” lists). Each code may be indexed by a unique index:

$$\begin{array}{rll} D_{parse} \rightarrow & Cons^A(\text{“}0\text{”}^1, []^a) & \\ | & Cons^B(\text{“}S\text{”}^2, d ::^b []^c) & d \in D_{parse} \\ | & Cons^C(\text{“}x\text{”}^3, d ::^d []^e) & d \in D_{parse} \\ | & Cons^D(\text{“}-\text{”}^4, d ::^f []^g) & d \in D_{parse} \\ | & Cons^E(\text{“}+\text{”}^5, d_1 ::^h d_2 ::^i []^j) & d_1, d_2 \in D_{parse} \\ | & Cons^F(\text{“}/\text{”}^6, d ::^k []^l) & d \in D_{parse} \\ | & Cons^G(\text{“}*\text{”}^7, d_1 ::^m d_2 ::^n []^o) & d_1, d_2 \in D_{parse} \end{array}$$

Now, this domain may be described by an alternative between different constructors (we say *constructor clones* for a constructor and its index) and a function which maps those indexed constructors to their arguments. Let $\mathcal{I}_\mathcal{C}$ be the set of constructor index, $\mathcal{C}^{\mathcal{I}_\mathcal{C}}$ the set of constructor clones (i.e. the set $\mathcal{C} \times \mathcal{I}_\mathcal{C}$) and $\mathcal{V}$ the type of values used by *mix*. Let $CArg : \mathcal{C}^{\mathcal{I}_\mathcal{C}} \times \texttt{int} \rightarrow \mathcal{V}$ the function which returns the value of the n-th argument of a constructor clone. The value which describes D_{parse} may be defined by:

$$\begin{array}{lll}
D_{parse} & = & Cons^A | Cons^B | Cons^C | Cons^D | Cons^E | Cons^F | Cons^G \\
CArg(Cons^A, 1) & \mapsto & \text{“}0\text{”}^1 \\
CArg(Cons^A, 2) & \mapsto & []^a \\
CArg(Cons^B, 1) & \mapsto & \text{“}S\text{”}^2 \\
CArg(Cons^B, 2) & \mapsto & ::^b \\
CArg(::^b, 1) & \mapsto & Cons^A | Cons^B | Cons^C | Cons^D | Cons^E | Cons^F | Cons^G \\
CArg(::^b, 2) & \mapsto & []^c \\
\ldots & \ldots & \ldots
\end{array}$$

We use the term *alternative values* for this kind of values. For instance, D_{parse} is an alternative between 7 constructor clones. Alternative values is a lattice with union and intersection (in the sense of the set of real values that they define) as least and upper bounds.

In comparison with the domains used by current partial evaluators, this structure (given by a value of $\mathcal{V}$ and the function *CArg*) is quite original. Usually partial evaluators have a domain which can describe completely known values (i.e., a constant) or completely unknown (symbolized by a particular constant noted $\perp$ or $\top$). Partial evaluators using partially static structures have the possibility to incorporate the unknown constant as a sub-value of a value (for instance $(\top, 1)$ is a partially static structure representing a pair whose first component is unknown and its second is 1). In [5], Haraldsson introduces a notion of alternative of values which associates different possible constants to an expression. A value is then described by a set of constants (for instance a value may be one of the integers 1 and 2). This domain can not describe a set of values like D_{parse} which can be seen as a context-free grammar description. In [3], a similar domain is used to describe values associated to an expression (they are called subdomain properties) but non-terminals are not the same as ours.

4 Conditional Specialization

The dual of constructor clones is *conditional clones*. If the original program has the conditional:

```
match e with
    c_1(x_1) ->  e_1
    ...          ...
    c_n(x_n) ->  e_n
```

a conditional clone of it, is a structure where index are added:

$$\begin{array}{lll}\texttt{match}\ e\ \texttt{with} & & \\ \quad c_1^{I_{1,1}}(x_1) & \texttt{->} & e_1^{1,1} \\ \quad \ldots & & \ldots \\ \quad c_n^{I_{n,m_n}}(x_n) & \texttt{->} & e_n^{n,m_n}\end{array}$$

As constructors are used by conditionals for selecting a branch, constructor clones select a *branch clone*. A branch clone is constituted of a set of constructor clones which can activate this branch and a clone of the expression corresponding to this branch. Branch clones are put together to create a conditional clone. The element $c_i^{I_{i,j}}$ are the set of constructor clones corresponding to this branch clone. Those constructor clones come from the alternative value associated to e. Thus, for each branch clone corresponding to the i-th original branch, we know the alternative value of x_i (with the function *CArg*) and the expression clone $e_i^{i,j}$ may use this fact.

5 The Utility of Alternative Values

The utility of alternative values comes from the technique of *tag removal* [7]. We can also say that they lead to type reconstruction then type simplification. With the example above, the domain D_{parse} is described by an alternative of 7 constructor clones.

- A type reconstruction says that its new type is constituted by 7 constructors (which are instances of the original constructor `Cons`).
- The first component of $Cons^A$ is the unique string "0"[1]. Its new type is constituted by a restriction of the string type to this unique element.
- The type reconstruction of the second component of $Cons^A$ is a restriction of the list type constituted of the unique element $[]^a$.
- The type of the second component of $Cons^B$ is constituted by the unique constructor of list $::^b$.
- For $::^b$, the first component is an alternative between several constructor clones (the same as for D_{parse}) and the second component has only one element $[]^c$.
- ...

The type reconstruction takes the different alternative values of each expression and tries to find new type definitions witch are compatible with type constraints. To simplify, for our example this leads to the "definitions":

```
type Values_parse = Cons_A of string_1 * list_a
                  | Cons_B of string_2 * list_b
                  | Cons_C of string_3 * list_d
                  | Cons_D of string_4 * list_f
                  | Cons_E of string_5 * list_h
                  | Cons_F of string_6 * list_k
                  | Cons_G of string_7 * list_m
    and  string_1     = "0"_1
    and  list_a       = []_a
    and  string_2     = "S"_2
    and  list_b       = Values_parse ::_b list_c
    and  list_c       = []_c
       ....
```

Here, a liberal notation is used for constructor clones. For instance, `"0"_1` means a constructor whose value is the string `"0"`, `::_b` means a constructor (which infix notation) similar to a list constructor.

The second step consists, with this new types, to simplify them. The principle is very simple. When a reconstructed type has only one constructor, this constructor may be removed (tag removal). For product type, constant components are removed. For the type `Values_parse` it gives the simplified definition:

```
    type Values_parses = Cons_A
                       | Cons_B of Values_parse
                       | Cons_C of Values_parse
                       | Cons_D of Values_parse
                       | Cons_E of Values_parse * Values_parse
                       | Cons_F of Values_parse
                       | Cons_G of Values_parse * Values_parse ;;
```

Of course, we need to transform expressions using this type:

– A constructor from a reconstructed type with only one element is replaced by a product of its arguments (if any).

$$c^i(e) \implies e$$

– A conditional of which its test expression belongs to a reconstructed type with only one element is replaced by a binding of local variables and the unique branch of this conditional.

$$\texttt{match}\ e_1\ \texttt{with}\ c^i(x)\ \texttt{->}\ e \implies \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e$$

Thus, this type specialization gives a more efficient program by removing the building of unnecessary constructors and by simplification of conditional with only one branch. Products and projections are specialized with regard to their constant arguments. Only non-constant parts are retained.

6 Problem of Equality Between Values

Now that the domain used by *mix* is defined, we can try to extend the technique of partial evaluator for alternative values. There exists two problems:

- A partial evaluator must identify certain specialized versions of the original functions. This mechanism usually compares the known parts of the arguments of the specialized functions and identifies them when they are equal. To exploit the same mechanism, an equality test between alternative values must be found. But, how can this comparison be made? An equality based on structural equivalence of context-free grammars [11] is complex. Moreover, there is a natural order $\sqsubset$ on grammars saying than $G_1 \sqsubset G_2$ if $\mathcal{L}(G_1) \subset \mathcal{L}(G_2)$. For instance, if a parameter is constructed with $Cons^A$ and another one with $Cons^A$ or $Cons^B$, an element of the first one is an element of the second. Since $Cons^A \sqsubset Cons^A | Cons^B$, a specialized function with an argument described by $Cons^A | Cons^B$ may be used for the parameter $Cons^A$.
- We also need to compare two constructor clones when they appear in the alternative value associated to the test expression of a conditional. Some time several constructor clones must activate the same branch clone. In fact, during specialization, there are no reason that the number of constructor clones is bound. As a consequence, if each constructor clone is said to be distinct to the other ones, the partial evaluator may loop by creating an infinite number of branch clones. For instance, if `f` is defined by:

  ```
  let rec f x = match x with true -> f false | false -> f true;;
  ```

  This function is not realistic since it always loops. However, such a scheme may appear in a real function. Suppose that `f` is specialized for x equals to $true^0$.
  - A first branch clone in `f` is created for $true^0$.
  - Then, `f` calls itself with an argument equals to $false^1$. The constructor clone $false^1$ appears in the "$true^0$" branch of `f`. A new branch is inserted for $false^1$.
  - This new branch calls `f` with a new constructor clone $true^2$ as argument. This clone creates another branch.
  - ...

  Finally an infinite number of versions of $true$ and $false$ is built, leaving the (infinite) program:

  ```
  let rec f x = match x with
      true_0  -> f false_1
    | false_1 -> f true_2
    | true_2  -> f false_3
        ....
  ```

To solve those problems, we have introduced a new mechanism for identifying specialized functions and constructor clones. This mechanism is not based on comparison of alternative values but rather on a resource analysis needed by each specialized expression.

7 Notion of Event

To introduce the mechanism that identifies functions and constructor clones, we start with a simple example. The function below `append` concatenates two lists:

```
let rec append x y = match x with e :: l -> e :: append l y | [] -> y ;;
```

This function may be specialized when `x` is bound to the list $v = [1;2]$ and `y` is unknown. It gives three versions of `append` corresponding to the three lists $[1;2]$, $[2]$ and $[]$. In this case, the traditional identification based on equality between the known part of function arguments works perfectly:

- The partial evaluator starts with a first version of `append` with `x` bound to $[1;2]$. This function is called `append_0`. `append_0` calls `append` with its first parameter equals to $[2]$.
- The partial evaluator creates a new version of `append` (`append_1`) with `x` bound to $[2]$. Then, `append_1` calls `append` with the first argument equals to the empty list $[]$.
- A third version of `append` is created for this new value of `x` (`append_2`).
- The partial evaluator stops because no new version of `append` needs to be specialized.

This process leaves the residual program:

```
let append_2 y = y ;;
let append_1 y = 2 :: append_2 y ;;
let append_0 y = 1 :: append_1 y ;;
```

If functions are unfolded, a more efficient program is returned:

```
let append_0 y = 1 :: 2 :: y ;;
```

Because, with alternative values, we do not want to compare values, we can not use this simple mechanism. If we look at the *events* that occur when `append` is evaluated with its first argument equals to $[1;2]$, we can see that this list is matched by the conditional of `append`. Then, during the first recursive call, the sub-list $[2]$ is tested. At the second recursive call, it is the empty list.

Now, suppose that the constructors of the list $[1;2]$ have been indexed (they are constructor clones): $1 ::^a 2 ::^b []^c$. Then the first call to `append` matches the constructor $::^a$, the second call matches $::^b$ and the third one matches $[]^c$. This fact gives a mechanism to distinguish the three calls to `append` without comparing the known argument:

- `append_0` is the initial call.
- The call to `append_1` is due to the *event* between $::^a$ and the conditional of `append`.
- The call to `append_2` is due to the *event* between $::^b$ and the conditional of `append`.

Thus, function identification is made by comparing the *events* that occurs between two calls. A similar mechanism may be applied for identification of constructor clones. This system try to know which part of the original value is needed to create a particular constructor. For instance, with the previous example, the list constructor appearing in `append_0` depends on the event between the conditional of `append` and the constructor $::^a$. The list constructor of `append_1` depends on two events: the one needed for the previous constructor (from $::^a$) and the one between the conditional of `append` and the constructor $::^b$.

We can restrict this resource analysis by computing, for each expression, which part of the initial data is needed to "activate" it. With an interpreter, this mechanism associates to each expression the part of the data structure representing the interpreted program which is needed at this point.

8 An Example of specialization

This section reports the successful partial evaluation of an interpreter for an applicative language when the interpreted program has three phases and performs symbolic derivative of arithmetics expression.

This experiment was made with the partial evaluator LaMix [1] that implements the ideas presented in this article. LaMix specializes programs written in a subset LaML of Caml-Light. LaML is a pure first order functional language with global function definitions, algebraic type declarations and a limited form of pattern-matching. LaMix is an experimental on-line partial evaluator based on alternative values and events.

8.1 A Tiny Interpreter int_{TML} for TML

The interpreter int_{TML} is written in LaML. It is essentially untyped but accepts constructed values and has a simple mechanism of pattern-matching. The language TML that it recognizes is defined by:

$$
\begin{array}{lll}
\mathcal{P} = & d_1 \cdots d_n & d_i \in \mathcal{D} \\
\mathcal{D} = & \mathtt{def}\ f(x_1, \ldots, x_n)\ \mathtt{=}\ e & f \in \mathcal{F},\ x_i \in \mathcal{V},\ e \in \mathcal{E} \\
\mathcal{E} = & \mathtt{var}\ x & x \in \mathcal{V} \\
\mid & \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 & x \in \mathcal{V},\ e_1, e_2 \in \mathcal{E} \\
\mid & \mathtt{true} & \\
\mid & \mathtt{false} & \\
\mid & \mathtt{constr}\ c(e_1, \ldots, e_n) & c \in \mathcal{C},\ e_i \in \mathcal{E} \\
\mid & \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 & e_i \in \mathcal{E} \\
\mid & \mathtt{match}\ e\ \mathtt{with} & \\
 & \quad c_1(x_{1,1}, \ldots, x_{1,m_1})\ \mathtt{->}\ e_1 & \\
 & \quad \ldots & \\
 & \quad \mathtt{|}\ c_n(x_{n,1}, \ldots, x_{n,m_n})\ \mathtt{->}\ e_n & e, e_i \in \mathcal{E},\ c_i \in \mathcal{C},\ x_{i,j} \in \mathcal{V} \\
\mid & \mathtt{call}\ f(e_1, \ldots, e_n) & f \in \mathcal{F},\ e_i \in \mathcal{E}
\end{array}
$$

The sets $\mathcal{F}$, $\mathcal{V}$ and $\mathcal{C}$ contain respectively identifiers for functions, variables and constructors. $\mathcal{P}$ is the set of TML programs, $\mathcal{D}$ corresponds to function definitions and $\mathcal{E}$ to expressions.

An expression is either a variable (a parameter of the current function or a variable bound by pattern-matching or let-expression), a let-expression which binds a variable to the value calculated for an expression, the truth values **`true`** and **`false`**, a constructor built from an identifier and a list of expression, an if-then-else, a pattern-matching conditional which computes an expression and following its value choose a case after binding the variables of the pattern to the values of the arguments of the constructor. The last construction calls a defined function. Variables are local to functions.

8.2 A TML Program for Deriving Symbolic Expressions

The TML program below performs a symbolic derivation of arithmetic expressions. It takes, as input, a stream of symbols representing a function in the prefix notation. The first phase builds an internal representation from the input stream as a tree of arithmetic operators and constants (the function **`parse`**, not defined here). Then, a function derives this tree returning the derivative abstract tree (the function **`derive`**). A last phase constructs the stream of symbols representing the abstract tree using prefix notation (the function **`print`**, not defined below).

```
def derive(e) = match e with
      Z         -> constr Z
    | S(e)      -> call derive(e)
    | X(e)      -> constr P(e,constr X(call derive(e)))
    | M(e)      -> constr M(call derive(e))
    | P(e1,e2) -> constr P(call derive(e1),call derive(e2))
    | I(e1)     -> constr M(constr T(call derive(e1), constr T(e,e)))
    | T(e1,e2) -> constr P(constr T(call derive(e1),e2),
                           constr T(e1,call derive(e2)))
def main(i) = print(derive(parse(i)))
```

8.3 Partial Evaluation of int_{TML}

The interpreter int_{TML} may be specialized for this program. Here, the main function is called **`exec`**. LaMix returns the following residual program (we have retained the code generated for **`derive`**):

```
type Value_0 = Tag_R | Tag_Q | Tag_P of Value_0 | Tag_O of Value_0
 | Tag_N of Value_0 | Tag_M of Value_0 | Tag_L of Value_0 * Value_0
 | Tag_K of Value_0 * Value_0 | Tag_J of Value_0 | Tag_I of Value_0 * Value_0
 | Tag_H of Value_0 * Value_0 | Tag_G of Value_0 | Tag_F of Value_0 * Value_0
 | Tag_E of Value_0 * Value_0 | Tag_D of Value_0 * Value_0
 | Tag_C of Value_0 * Value_0 | Tag_B of Value_0 | Tag_A of Value_0 * Value_0 ;;

let rec bind_2 vs = match vs with
     Tag_R   -> Tag_Q
   | Tag_P p -> Tag_A(p,Tag_B(bind_2 p))
   | Tag_O p -> bind_2 p
   | Tag_N p -> Tag_J(bind_2 p)
   | Tag_M p -> Tag_G(Tag_H(bind_2 p,Tag_I(vs,vs)))
   | Tag_L p -> Tag_F(bind_2(fst p),bind_2(snd p))
   | Tag_K p -> Tag_C(Tag_D(bind_2(fst p),snd p),Tag_E(fst p,bind_2(snd p))) ;;

let rec exec s = bind_3(bind_2(bind_1(s)) ;;
```

As one can see, a new types has been created. The type **`Value_2`** is a specialized version of the universal type **`Value`** for abstract tree constructors. The relation between the new constructors and the symbolic derivative program is as follows:

```
Tag_R          → Z              Tag_Q          → Z
Tag_P(e)       → X(e)           Tag_O(e)       → S(e)
Tag_N(e)       → M(e)           Tag_M(e)       → I(e)
Tag_L(e1,e2)   → P(e1, e2)      Tag_K(e1,e2)   → T(e1, e2)
Tag_J(e)       → M(e)           Tag_I(e1,e2)   → T(e1, e2)
Tag_H(e1,e2)   → T(e1, e2)      Tag_G(e)       → M(e)
Tag_F(e1,e2)   → P(e1, e2)      Tag_E(e1,e2)   → T(e1, e2)
Tag_D(e1,e2)   → T(e1, e2)      Tag_C(e1,e2)   → P(e1, e2)
Tag_B(e)       → X(e)           Tag_A(e1,e2)   → P(e1, e2)
```

The function `bind_1` corresponds to `parse`, `bind_2` to `derive` and `bind_3` to `print`. We can notice that there are different versions of the same initial constructor of the TML program. It is due to the different instances of them in the TML program. However, from the point of view of efficiency, the residual program is as quick as an optimal one (tests give an improvement in time of about 45–50).

Our goals as they were introduced at the beginning of the paper, are plainly reached. LaMix has been able:

- to transform completely the interpreted program into control points in the residual program,
- to eliminate environment used by $intr_{TML}$ and
- to introduce new types for intermediary results of residual functions. The encoding of TML constructors by $intr_{TML}$ has disappeared inside the residual program.
  Of course, the residual program needs to receive an encoded argument and it returns an encoded value. We cannot avoid this problem because the residual function and the original one must be equivalent.

9 Conclusion and Related Works

The alternative values and the mechanism of identification (events) has been able to remove all the layer of interpretation for the example shown here. In particular, a new type definition for intermediary results was generated. The value encoding of the interpreter has been completely removed. The residual program looks like a compiled version of the original one.

This technique realizes a complete compilation of the original program (as a compiler does) derived from an interpreter. Moreover, the use of events to identify specialized objects (by comparison to other partial evaluators) ensures termination of the compilation process for abstract datatypes.

The two main works on type specialization are [10] which introduces the notion of constructor specialization and [3] which extends this technique and uses an abstract specialization phase based on sub-domains properties. This last paper has similarities with our work since the objectives are the same and the different phases of this system are similar to our. However, there are some differences. The first one is that LaMix is an on-line partial evaluator and in [3] the system is off-line with a binding time analysis. As a consequence, specialized objects for this partial evaluator are identified by the known part of their arguments. In the same way, comparisons between context-free grammars (which describe sub-domain properties) are made. We have said that this comparisons are difficult to implement (and are time-consuming). It was one of the main reasons to use events. Another problem comes from the fact that this partial evaluator may loop very easily (more often than its predecessors because it preserves more static informations). Events prevent LaMix to run forever on recursive abstract datatypes since there are only a finite number of events relative to them.

LaMix may be extended in different ways. We work to add polymorphism, higher order function and side effects. Another important task will be to transform LaMix into an off-line partial evaluator.

References

1. D. Bechet. *Les valeurs alternatives et la notion d'événement dans l'évaluation partielle.* PhD thesis, Université de Paris VII, Paris, France, october 1995.
2. 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.
3. D. Dussart, E. Bevers, and K. de Vlaminck. Polyvariant constructor specialisation. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 54–65. ACM SIGPLAN, 1995.
4. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
5. A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.
6. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.
7. J. Launchbury. *Projection Factorisations in Partial Evaluation.* Cambridge: Cambridge University Press, 1991.
8. J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 145–164. ACM, Berlin: Springer-Verlag, 1991.
9. T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. Amsterdam: North-Holland, 1988.
10. T. Mogensen. Constructor specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 22–32. New York: ACM, 1993.
11. A. Salomaa. *Formal languages.* Computer Science Classics. Academic Press, Inc., 1987.
12. V.F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.
13. Pierre Weis and Xavier Leroy. *Le Langage CAML.* InterEditions, 1993.