Limix: a Partial Evaluator for Partially Static Structures

Denis Bechet
CRIN-CNRS & INRIA Lorraine
Campus scientifique, B.P. 239
54506 Vandœuvre-les-Nancy Cedex
France
e-mail: Denis.Bechet@loria.fr

Abstract

Partial evaluators essentially deal with values which can completely be computed during specialization. Some of them handle partially static structures. In this paper, we demonstrate that partial evaluation can be split into three main mechanisms. Moreover, this is suitable for partially static structures. The first mechanism specializes functions when a part of its arguments is known and finds an approximation of its result. The second one performs a use-analysis, erases unused expressions and specializes both constructors and their corresponding destructors. The third one implements an unfolding strategy. Limix, a partial evaluator for a first-order subset of ML is based on those ideas.

Keywords: partial evaluation, partially static structures, use-analysis, data specialization, functional programming.

1 Introduction

Program transformations are nowadays a common technique for improving the efficiency of programs. Partial evaluation (also known as program specialization) is a relatively simple technique belonging to the transformational framework. Its principle consists in evaluating a program when one or more of its inputs are known. Of course, since all inputs are not yet fixed, it returns a specialized program which can be executed later with the rest of the inputs. Classical specializers essentially deal with values which can be completely computed during specialization and with their propagation through the program. Some of them, like FUSE [WCRS91], Schism [Con88, Con93] or recently Similix [Bon93, Bon91a, BJ93a], try to take care of *partially static values* like a pair of which only the first component is known at specialization time. However, the first one can only use this information in the environment where those structures were built and are generalized otherwise. The second one uses partially static structures for carrying constant sub-expressions. Similix treats partially known structures only for special constructors.

To solve the problem brought by partially static values, we must first look at the interest of computing those values during specialization time and think on what can we do with them. Our thesis, here, is that partial evaluation should be split into three mechanisms. The first one looks like the traditional function specialization when one or more of their arguments are known (eventually not completely). The second one computes the set of values really needed, eliminates the parts that will never be used and transforms both constructors and their corresponding destructors. Finally, the third applies an unfolding strategy.

From a theoretical point of view, if f is a function of two arguments and if we know at specialization time that its first argument x is equal to v and its second y unknown, partial evaluators replace the call $(f\ x\ y)$ by a call to a specialized function f^*_v with only one argument: $f^*_v\ y$. The value v disappears completely. This substitution is based on two more primitive transformations. The first one replace a call $(f\ x\ y)$ by a call to a specialized function $(f_v\ x\ y)$. The first argument is then given to f_v. However, because x is known, the variable which is bound to v in the body of f_v is never used. So, this call is replaced by a call to the same function but with fewer arguments: $f^*_v\ y$ and the variable which is unused is also eliminated from the list of arguments of f_v. The first mechanism is a specialization but the second deals with elimination of unused expressions. Thus, we can distinguish two phases: a *specialization* mechanism and an *elimination* mechanism.

This distinction becomes important when partial evaluators handle also partially static structures. For instance, if f is an uncurried version of the previous function, the same call $f(x,y)$ computes a partially static pair. Its first component is v and its second unknown. The specialization phase returns $f_v(x,y)$. Thus the elimination phase deduces that x is no more needed and it transforms the pair. The residual call becomes $f^*_v\ y$. Standard specializers do not perform such a transformation thought this version of f and the previous one are very similar. Our proposal resolves this problem.

This problem is much more interesting when a function takes a value belonging to a *sum type*. For instance, let f be a function which takes a list as argument, during specialization process, we know that this list is partially known: its length is three but the three elements are unknown. Then, f can be specialized giving $f_{[x;y;z]}$. But, how many arguments has this function ? What are they ? What can be done with the code which computes this list ? A first solution raises the number of arguments of f. The call $f([x;y;z])$ is replaced by $(f^{*}_{[x;y;z]}\ x\ y\ z)$. However, this transformation does not always increase the efficiency. Even worse, it can decrease it. For instance, if this list is given as argument to other functions inside f or if it is needed as a part of the result of f. This remark becomes obvious when, instead of a list, the partially static structure represents an array the size of which is known but not its elements. In this case, this transformation replaces a call with the reference to this array by a call with all the values stored in this array. One may see this problem as the well-known problem of call-by-reference versus call-by-value.

Another kind of transformations solves partially this problem without raising the number of arguments of f. In our example, since we know that we have a list of three arguments, in the body of $f_{[x;y;z]}$, the code, which checks that its argument is a list, can be omited. Thus, a good idea consists in transforming the list by a product. Then, the call to $f_{[x;y;z]}$ becomes $f_{[x;y;z]}(x,y,z)$. A sum type is transformed into a product type and the function $f_{[x;y;z]}$ does not take a list but a product, eliminating by this way the dynamic checking on lists. However, there is a problem if this list is needed by the result of f. In this case, in the body of $f_{[x;y;z]}$, this list needs to be recreated and it would be better to give directly the list rather than a product. In fact, this transformation is conditioned by the use of the list. If it is always destroyed inside f then this transformation is correct. Otherwise, it is questionable. Here again, the two phases we propose, detect these situations. The specialization first replaces $f([x;y;z])$ by $f_{[x;y;z]}([x;y;z])$. Then use-analysis detects that this list is always used and transform this call by $f^{*}_{[x;y;z]}(x,y,z)$.

An important example of partially static structures where the program is an interpreter which uses association lists for implementing environments is well treated by our mechanisms. The first component of each pair of those lists carries a variable name and the second its current value. In general, at each point of the interpreted program, the list length and the variable names are known whereas their actual values are unknown. The first phase specializes the interpreter functions with the current expression and the current environment (partially known). The second phase discovers that expressions are never used and environments are always destroyed. Thus, some expressions disappear and environments are transformed into products (the names of variables also disappears).

This approach which splits partial evaluation in two mechanisms also makes the problem of function unfolding completely independent. Unfolding can be done at the end of the specialization process, in a third phase.

Our system Limix implements those ideas. It is split in several general parts. The first one specializes functions. The second and the third ones have the responsibility of erasing unused expressions and of specializing partially static structures. A last phase performs unfoldings.

The rest of this article is divided as follows. Section 2 introduces LiML, the language treated by Limix. Then an overview of Limix is given. Section 4 describes the specialization phase. Section 5 and 6 present the erasing of expressions and the constructors transformations. Limix split this phase into two sub-phases. The first one describes in section 5 performs a use analysis and erases unused expression whereas the second one presented in section 6 simplifies products constructors and transforms values belonging to a sum type into products. Section 7 gives an example of specialization with Limix. Sections 8 and 9 compare Limix with other related works and conclude.

2 LiML: A small first order language

This section describes the language treated by Limix. LiML is a subset of CAML-light[Ler93], a dialect of the functional language ML.

LiML is a first order language with type declaration (tagged sum type), products, pattern-matching and conditional, local variables (`let`) and global function and variable declarations.

Functions are uncurried (they always take only one argument, eventually a product). Lists have a particular syntax. hd`::`tl builds a list whose first element is hd and its tail tl. The empty list is `[]` and `[`e_1`;`...`;`e_n`]` builds a list whose elements are $e_1 ... e_n$.

Its syntax is as follows:

```
Prog    =  Instr ... Instr
Instr   =  TypeDecls || Decls
           include FileName
Decls   =  let [rec] Decl and ... ;;
Decl    =  PPat = Expr
           Fun PPat = Expr
           Fun = function Cases
PPat    =  Var || _ || (PPat ) || PPat,...,PPat
Cases   =  Pat -> Expr | ... | Pat -> Expr
Pat     =  PPat || ( Pat ) || Cons [ Pat ]
           Pat ,..., Pat
Expr    =  Pat || Fun Expr
           ( Expr ) || Expr,...,Expr
           let PPat = Expr and ... in Expr
           if Expr then Expr else Expr
           match Expr with Cases
```

Section 7 shows an example of a LiML program which is an interpreter for the toy language "MP" (introduced in [Ses85]). MP is a small imperative language with Lisp structures, assignments, conditionals and while-loops.

The concrete syntax of MP-programs is:

```
Prog    =  Head Block
Head    =  program Name Var ... Var =
Block   =  Com
           { Com ;...; Com }
Com     =  let Var = Expr
           if Expr then Block else Block
           while Expr do Block
Expr    =  Constant || Var || Op(Expr,...,Expr)
           (Expr) || Expr::Expr || [e_1,...,e_n]
Op      =  car || cdr || cons || equal? || atom?
```

The program head defines the list of variables used inside it. The first symbol is the name of the program. It generally serves as input argument. At the beginning, all variables are bound to an initial value given to the evaluator which corresponds to the input. The value `N` is identified to *false*. `Eval` returns the final environment. `EvalB` evaluates a block, `EvalC` a command and `EvalE` an expression.

3 Overview of Limix

Limix is divided in several phases. The front-end reads a program and transforms it into an internal representation. The second phase specializes function bodies with the values (partially or completely known) of its argument and computes its return value. This phase is named *destructors specialization* because it transforms structures like pattern-matching and conditionals. The third part called *constructor specialization* erases unused expressions and annotates constructors which are always used. It is based on a backwards analysis. The next phase called *mutual specialization of constructors and destructors* implements transformations of tagged values to products, eliminates unnecessary products and transforms their corresponding destructors. An unfolding strategy follows. Finally, a code generator produces the residual program.

The entire system works with two levels. The higher level written in CAML drives the general algorithms. The lower level more automatic has to propagate information through a function body and has the task of performing transformations inside it. It uses a graph rewriting system based on *interaction nets* [Laf90, Bec92]. This choice is motivated by several reasons. First, since the FUSE experiment, graphs is known to be interesting for representing a function body. The sharing of expressions is easily realized. A second reason is based on the kind of algorithms that Limix applies. Since each phase looks like a data propagation joined with local transformations inside each function body, a rewriting system works perfectly well. Moreover, the sense of information propagation during the different phases are different (forwards or backwards) and because interaction nets are completely symmetric (no difference between constructors and destructors), they give an elegant implementation and very similar mechanisms for all phases of Limix. However, this characteristic is not central here and we do not have enough space to describe this feature in details. An interaction net is a net composed of labeled node called *agents* connected two by two using *link*. A net can be *reduced* using rewriting rules named *interaction rules*.

The front-end of Limix reads the program which needs to be specialized from a file and transforms function bodies into nets and rules. The last function definition gives the function to specialize. Its argument are supposed to be unknown. For instance, if we want to specialize the function `append` with its second argument set to `[1;2;3]`, one can write the program:

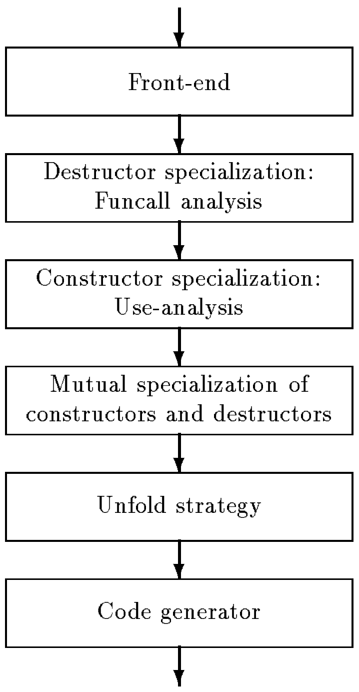
```
let rec append(x,y) =
   match x with [] -> y
              | h::t -> h::append(t,y) ;;
let append123 x = append(x,[1;2;3]) ;;
```

The function that will be partially evaluated is `append123`. This phase also checks syntax and binding of variables. However, type checking is not performed and one needs first to evaluate the program with CAML (a program written in LiML is a CAML program and has the same semantics). The front-end returns a list of function identifiers with the interaction net corresponding to their body.

Then, the destructor specialization phase computes the different function calls (a couple function identifier/value) and the return value associated to each specialized function. This analysis is polyvariant (it creates different versions of the same function). A second sub-phase transforms the bodies of all functions. The result is a new list of specialized function defined by the original function identifier, the value of the argument and the transformed net corresponding to the specialized body.

The next phase performs a use-analysis, erases unused expression and annotates constructors always used. The use-analysis is monovariant (one use-value for each specialized function): it computes the used-value of the argument of a function knowing the use-value of its return value. Then, a second sub-phase erases expressions and annotates constructors. The result is the same list of specialized function of the previous phase but with a new net for their body.

The "mutual specialization of constructors and destructors" looks like, for its analysis part, the "destructor specialization" and it is also polyvariant. However, this analysis always terminates. Here, only annotated constructors and the dummy value ∞ (which means that, during the previous phase, Limix has discovered that this expression is never used) build values. A second sub-phase transforms constructors and destructors using those values. The result is the same function list of the previous phase with a new body.


Front-end
Destructor specialization: Funcall analysis
Constructor specialization: Use-analysis
Mutual specialization of constructors and destructors
Unfold strategy
Code generator

The next phase unfolds function calls. There are two strategies in Limix. The first one simply unfolds specialized function calls when this function is called only once. The second strategy unfold the maximum of function calls if this function is not recursive. Both strategies always terminate. The second one gives a faster program but generally bigger than the first one. Moreover, unfolding does not produce code duplication.

The last phase of Limix generates code from the list of specialized functions and the net corresponding to their body. The only difficulty here is to rebuild a hierarchy of mutually recursive function definitions. The translation from interaction nets to LiML terms is easily done.

4 Destructor specialization

The first part of Limix consists in propagating values from where they are created (*constructors*) to where they are used (*destructors*) and to specialize those destructors. This mechanism can be seen as a restricted specializer where a special value $\top$ is added to handle partially static values and which uses only function calls specialization (no unfolding, no arity raising and no constant simplification).

Destructor specialization is split into two sub-phases. The first one, called *function call analysis* records function calls with the value of their argument and computes an approximation of their return value. This mechanism is not very different to polyvariant static analysis. However, since the domain used by this phase is potentially infinite, there is a problem of termination which is similar to the problem of infinite specialization. The second phase specializes function bodies by transforming and annotating destructors.

This mechanism returns a set of specialized functions constituted by a function name, the value of its argument and the specialized body. For each function, different specialized versions may exist depending on the different values of its argument.

The abstract domain of values $\mathcal{V}$ and the least upper bound operator $\sqcup$ are defined below.

$$\begin{array}{rcl} \mathcal{V} & = & \bot \\ & & \top \\ & & \Pi(\mathcal{V},\mathcal{V}) \\ & & C_i^T \\ & & C_i^T(\mathcal{V}) \end{array}$$

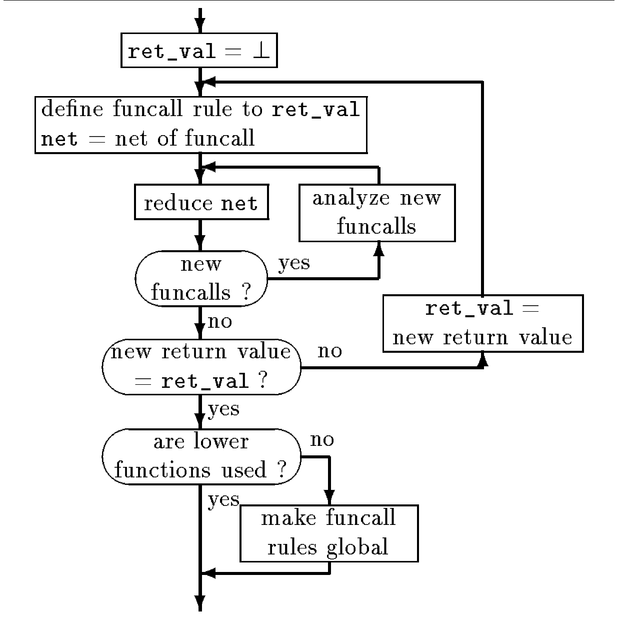
$$\begin{array}{rcl} \bot \sqcup y & = & y \\ \top \sqcup y & = & \top \\ \Pi(x_1,y_1) \sqcup \Pi(x_2,y_2) & = & \Pi(x_1 \sqcup x_2, y_1 \sqcup y_2) \\ C_i^T(x_1) \sqcup C_i^T(x_2) & = & C_i^T(x_1 \sqcup x_2) \\ C_i^T(x_1) \sqcup C_j^T(x_2) & = & \top \text{ if } i \neq j \end{array}$$

$\bot$ denotes a not yet known value and it appears as the result value of a not yet analyzed function. Normally, it disappears at the end of this phase. A remaining $\bot$ means that one of the specialized function never returns a value (it loops forever). The value $\top$ means a definitively unknown value, that results either from an initial unknown value or from a generalization of the result of a conditional (or from a generalization needed to insure termination). Finally, $\Pi(x,y)$, C_i^T and $C_i^T(x)$ denote a pair and a tagged value (constant or not).

4.1 Function call analysis

This phase determines the set of triples function/argument value/ return value. This information is gradually saved as interaction rules. The main goal is to compute the interaction rules between a particular function and a value which gives the corresponding return value.

The upper level implements the following algorithm:



ret_val = ⊥
define funcall rule to ret_val
net = net of funcall
reduce net
analyze new funcalls
new funcalls ?
yes
no
ret_val = new return value
new return value = ret_val ?
no
yes
are lower functions used ?
no
yes
make funcall rules global

This analysis takes a function call which is composed of a function and the value of its argument. First, it defines the return value associated to it as $\perp$. Then it reduces the interaction net corresponding to the body of this function where the value of its argument is put on argument link. It analyzes recursively new function calls. Thus the return value is compared to the old one. If they are different, the loop is executed one more time. At the end, the rule which defines the return value for this particular function call is made global if it (and new function calls) does not call a function which is currently analyzed.

At the level of interaction nets, values are computed as follows:

- A product $\Pi(x,y)$ returns the value $\Pi(v_x,v_y)$ where v_x and v_y are the values associated to x and y.
- The projection $fst(x)$ returns $\perp$ ($\top$) if v_x is $\perp$ ($\top$) and v_1 if v_x is $\Pi(v_1,v_2)$.
- Local binding `let` *pat* `=` e_1 `in` e_2 looks like a mixing of duplication nodes and projections.
- A constructor $C_i^T(x)$ of a sum type T returns $C_i^T(v_x)$.
- A selector $\overline{C}_i^T(x)$ returns $\perp$ ($\top$) if v_x is $\perp$ ($\top$); v if v_x is $C_i^T(v)$ and $\perp$ if the constructor does not match (if this program is used, this code will produce an error).
- A conditional `if` c `then` e_1 `else` e_2 returns v_{e_1} (v_{e_2}) if v_c is $true^{bool}$ ($false^{bool}$); otherwise $v_{e_1} \sqcup v_{e_2}$. In fact, the value returns by e_1 (e_2) is computed after the value v_c. Thus, the unused branch is never evaluated.
- Pattern-matching is a mixing of selectors and conditionals.

4.2 Destructor transformations and annotations

This second phase specializes destructors of each body of the functions discovered during the previous phase. For each function with a particular value of its argument, the interaction net, which represents its body, is transformed into an other net where calls are annotated with the values of their arguments, where destructors like conditionals, pattern-matching, selectors, projections and static binding are specialized.

If $\perp$ appears inside a value, it means that this value will never be computed at run time. So, this phase erases expressions which receive such a value. The other transformations are as follows:

- A projection $fst(x)$ is left if v_x is $\top$. Otherwise, it is annotated with a star * to denote that this destructor receives a pair: $fst^\star(x)$.
- A selector $\overline{C}_i^T(x)$ is also left if v_x is $\top$. If v_x is $C_i^T(v)$ the selector is annotated like projection with a star *: $\overline{C}^{\star T}_{\ i}(x)$. Otherwise, the annotation $\overline{C}^{\infty T}_{\ i}(x)$ is produced. When the code for this expression will be generated, this constructor will be replaced by a call to an error function.
- A conditional `if` c `then` e_1 `else` e_2 erases e_2 (e_1) if v_c is $true^{bool}$ ($false^{bool}$) and the condition is linked to an erase node. Otherwise it is left.
- Pattern-matching is a mixing of selectors and conditionals.
- Function calls are replaced by a call to the specialized function.

The result of this phase is a set of specialized functions defined by the name of their initial function, the values of their argument and the net which defines their body.

5 Constructor specialization

This second part of Limix is intented to analyze which constructors are never used or always destroyed. This mechanism is a backwards analysis which starts with destructors and ends at constructor places. The similarity between the previous phase and this one is obvious when we look at the duality between a pair which is a constructor and the pattern-matching on a product type : a line like `(x,y)` has the same behavior as `let (x,y) in ...` if we turn expressions upside down. This *duality* is the main reason of our choice of implementing terms as interaction nets and evaluation as rewriting rules. Since interaction nets are symmetrical, this phase becomes very similar to destructor specialization. The main difference between this part and the previous one is that it always terminates which is not the case for function calls analysis due to a possibility of infinite specialization of a recursive function.

Like destructor specialization, this phase is divided in two sub-phases. The first phase called *use analysis* determines the argument usage of a function when we know how its return value is used. This is similar to function calls analysis except that the direction of propagation is opposite. Each function is "specialized" with the different uses of its return value and may produce different versions. The second sub-phase transforms and annotates constructors like the second sub-phase of destructors specialization. Then, a function can be seen as an object which is both a constructor and a destructor. So, both parts of Limix give specialized versions of the same function.

5.1 Use analysis

Use analysis follows the same algorithm than function calls analysis. The domain of values is the dual of the previous analysis. Below the definitions of this domain and the least upper bound are presented:

$$\begin{array}{rcl} \overline{\mathcal{V}} & = & \overline{\bot} \\ & & \overline{\top} \\ & & \overline{\Pi}(\overline{\mathcal{V}},\overline{\mathcal{V}}) \\ & & \overline{C}_i^T(\overline{\mathcal{V}}) \end{array}$$

$$\begin{array}{rcl} \overline{\top} \sqcup y & = & y \\ \overline{\bot} \sqcup y & = & \overline{\bot} \\ \overline{\Pi}(x_1,y_1) \sqcup \overline{\Pi}(x_2,y_2) & = & \overline{\Pi}(x_1 \sqcup x_2, y_1 \sqcup y_2) \\ \overline{C}_i^T(x_1) \sqcup \overline{C}_i^T(x_2) & = & \overline{C}_i^T(x_1 \sqcup x_2) \\ \overline{C}_i^T(x_1) \sqcup \overline{C}_j^T(x_2) & = & \overline{\bot}\ i \neq j \end{array}$$

At the end, $\overline{\bot}$ means an unknown use, $\overline{\top}$ means that this value is never used, and the dual of a product or a tagged value means that this value always reaches a projection or a selector annotated with a star *.

- A product $\Pi(x,y)$ with $\overline{\top}$ ($\overline{\bot}$) as use-value puts $\overline{\top}$ ($\overline{\bot}$) on x and y. With $\overline{\Pi}(v_1,v_2)$, it puts v_1 on x and v_2 on y.
- The projection $\underline{fst^\star(x)}$ with v, puts $\overline{\Pi}(v,\overline{\bot})$ on x. $fst(x)$ always puts $\overline{\top}$ on x.
- A local binding `let (x,y)=` e_1 `in` e_2 annotated with a star $\star$ puts the use-value at the top of e_2, gets back the use-values v_x and v_y of the variables x and y and puts the value $\overline{\Pi}(v_x,v_y)$ at the top of e_1. This mechanism seems complex. However, with the translation of ML to interaction nets, it appears simple and is similar to a pair for function calls analysis.
- A duplicate node gets the two use-values and puts their least upper bound.
- A constructor $C_i^{\star T}(x)$ of a sum type T puts v on x if its use-value is $\overline{C}_i^T(v)$. Otherwise, it puts $\overline{\top}$ or $\overline{\bot}$.
- A selector $\overline{C}_i^{\star T}(x)$ puts $\overline{C}_i^T(v)$.
- A residual conditional `if` c `then` e_1 `else` e_2 simply copies its use-value to both e_1 and e_2 and puts $\overline{\top}$ on c.
- Pattern-matching is a mixing of selectors and conditionals.
- A function call not yet analyzed will be analyzed later (after reduction of this net) and an already analyzed function call puts the use-values corresponding to this call on its arguments.

5.2 Constructor erasing and annotations

This phase is the dual of the destructors specializations and annotations phase. It uses the result of use-analysis by reducing interaction nets corresponding to a function body for a particular use-value and defining a new definition. Constructors with a use-value equal to $\overline{\top}$ simply disappear. The dummy value ∞ is put on their return link. This value is needed by the next phase of Limix for constructors simplifications. Those which receive a value $\overline{\bot}$ are left residual. The last case occurs when a constructor receives a value which is composed by the dual value (for instance $\overline{\Pi}(x,y)$ for a pair). It means that the value computed by this constructor is always used. Those constructors are annotated with a star *. Destructors are transformed if one of their returns links has a value $\overline{\top}$. For instance, a local binding like `let (x,y) =` e_1 `in` e_2 where the variable y is never used is transformed to a projection $fst^\star(e_1)$.

6 Mutual specialization of constructors and destructors

This phase uses the result of the previous phase and specializes both constructors and their corresponding destructors. It is split like destructor specialization and constructor specialization into two sub-phases. The first sub-phase computes for a particular function call, which is the part of its argument and of its result which comes from a constructor annotated with a star *. This sub-phase is similar to function calls analysis. However, it always terminates. The second mechanism specialized function bodies using this information.

This mechanism is based on the following transformations:

- $\Pi^\star(x, \infty)$ is replaced by x. A pair, whose second argument is never used, simply disappears. The corresponding destructor $fst^\star(x)$ also disappears and becomes x.
- $C^{\star T}_i(x)$ becomes x. A tagged value forgets its tag. The corresponding selector $\overline{C}^{\star T}_i(x)$ also disappears. This transformation means that since we know which kind of constructor will reached those destructors, we can eliminate the tag This mechanism transforms a value belonging to a sum type to a value belonging to a product type and then eliminates the dynamic type checking.
- A constructor (product or sum types) whose arguments are all equal to ∞ disappears.

7 A simple example of specialization

This section describes the result of specializing the small interpreter shown below for a particular MP-program:

```
(* An interpreter for MP *)
(* Types for Values, Expressions and Commands *)
type V = N | P of V*V | Q of int ;;
type E = Quote of V | Var of string
       | Car of E | Cdr of E | Cons of E*E
       | Atom of E | Equal of E*E ;;
type C = Let of string*E
       | If of E*C list*C list
       | While of E*C list  ;;

(* Definition of EvalE, lookup, ... *)
include "MPaux" ;;

(* EvalC Command * Env -> Env *)
(* EvalB Command list * Env -> Env *)
let rec EvalC(c,env) =
   match c with
        Let(v,e)   -> store(v,(EvalE(e,env)),env)
      | If(e,b1,b2)-> (match EvalE(e,env) with
                 N -> EvalB(b2,env)
               | _ -> EvalB(b1,env))
      | While(e,b) -> (match EvalE(e,env) with
                 N -> env
               | _ -> EvalC(c,(EvalB(b,env))))
and EvalB(b,env) =
   match b with
       []   -> env
     | c::b -> EvalB(b,(EvalC(c,env))) ;;

(* Eval Program * Value -> Environment *)
let Eval(p,v) = EvalB(snd p,
                      init_env(fst p,v));;
```

The MP-program computes x times y. The two arguments x and y come from the first and the second component of an input pair. Numbers are represented as a list of 1 using unary arithmetic. There are two nested loops, one calculating an addition and the other a multiplication.

```
program mult i out x y =
  { let x = car(mult) ;
    let y = cdr(mult) ;
    let out = [] ;
    while y do
      { let y = cdr(y) ;
        let i = x ;
        while i do
          { let i = cdr(i) ;
            let out = 1::out}}}
```

We want to specialize the MP-interpreter with this program and its first argument x sets to 2. So the residual program will compute $2y$.

A first task consists in writing a LiML function `twice` which calls the MP-interpreter with the proper arguments:

```
(* A twice function *)
include "MPintr" ;;
(* The MP program mult *)
let mult = (["mult";"i";"out";"x";"y"],
            [ Let("x",(Car(Var"mult"))) ; ...
(* twice(y) = 2y *)
let twice y =
  lookup("out",Eval(mult,
                    P(P(Q 1,P(Q 1,N)),y))) ;;
```

Then, Limix starts to specialize `twice`. Function call analysis needs to generalize the value associated to the variable `out` for avoiding infinite specialization. Here this is not automatique. This problem fixed, Limix gives the following program (after variable renaming):

```
(* residual program *)
(* "while y" loop : (out,y) -> (out,y) *)
let rec EvalC_1 v =
   match snd v with (* y *)
```

```
      N -> v
    | _ ->  EvalC_1(P(Q 1,P(Q 1,fst v)),
                    tail(snd v)) ;;
let twice y =
  let e = ([],y) in
    match snd e with (* y *)
       N -> snd e
     | _ -> snd  EvalC_1(P(Q 1,P(Q 1,fst e)),
                         tail(snd e)) ;;
```

The MP-program, variables x, i and *mult* have completely disappeared. Environments have been replaced by a product (variables *out* and y), erasing the name of variables. This program computes a loop duplicating y. One may notice that the residual loop has a preamble because the first while-loop command unfold in **twice** is specialized with i equal to the input argument and the second one **EvalC_1** with i equal to 0. If i is also generalized (like *out*) no preamble will remain.

8 Related Works

The only attempt to divide partial evaluator into two or more phases like Limix is mentioned in [KW92]. This system is divided into a polyvariant analysis and a code generation. The first phase looks like our destructor specialization plus an unfolding strategy. However, the motivation for this work is to generate polymorphic and highly reusable specialized function bodies. A use-analysis is performed for avoiding (or reducing) the infinite specialization problem. This mechanism is different from our use-analysis and is included in the specialize phase.

Some partial evaluator handle partially static structures. Similix [Bon91b, BJ93b] treats partially static structures only for user defined datatypes. In the residual program, constructors creating a partially known value disappear and their sub-expressions are given directly as arguments of specialized functions. This may lead to inefficiency as mentioned in the introduction. Moreover, unused sub-expressions are never erased and no data specialization is done.

In FUSE [WCRS91] and Schism [Con88, Con93], partially static structures only serves to carry other values (like a `cons` of which the first component is a constant). Partially static structures are not specialized. Moreover, in FUSE, they are generalized when the current environment changes (A non-unfold call to a function).

The most interesting work on data specialization is [Mog93]. This paper describes a mechanism which introduces specialized types of an initial type. This mechanism looks like the "mutual specialization of constructors and destructors" phase of Limix. However, Limix transforms a value belonging to a sum type into a product, not into a specialized sum type. Thus, the dynamic tests on those values disappear. Moreover, Mogensen does not eliminate unused expressions or the unused branches of conditionals creating unnecessary calculus and producing dead code which does not occur with Limix.

9 Conclusion and Future Work

We demonstrate, in this article, that partial evaluation can be split into three phases. The first one specializes function bodies when some of their arguments are known (eventually partially known). The second phase tries to know which parts of expressions are never used and which are always used. Then it erases unnecessary code and specializes (or simplifies) both constructors and destructors, function heads and function calls. A last phase produces unfolding. This fact which was not very obvious when partial evaluation only deals with completely known values, becomes important with partially static structures.

A lot of work still remains to have an independent and powerful partial evaluator. The first important point is to solve the problem of infinite specialization. This subject has not been tackled in this paper. In fact, because Limix, during specialization phase, propagates not only completely static data but also partially static structures, this problem is more serious than with partial evaluators using only completely known or unknown values. A generalization mechanism is needed to avoid (or reduce) infinite specialization.

LiML is a first order language with no side-effect. A possible extension to Limix is to enable lambda-abstractions and mutable types like in CAML. However, it is not clear to see if partially static closures can participate to a mechanism like the one presented in this article about partially static structures. Side-effects are also difficult to handle.

Another way of investigation concerns the problem of arity raising. This mechanism transform an uncurried function to a curried form. For instance, a call $f(x, y)$ becomes $f\ x\ y$.

A needed-analysis may be also add to Limix for the purpose of polyvariance and reusability of specialized function bodies like the mechanism in [KW92].

References

[Bec92] D. Bechet. Partial evaluation of interaction nets. In M. Billaud et al., editors, *WSA '92, Static Analysis, Bordeaux, France, September 1992. Bigre* *vols 81–82, 1992*, pages 331–338. Rennes: IRISA, 1992.

[BJ93a] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[BJ93b] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.

[Bon91a] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[Bon91b] A. Bondorf. Similix manual, system version 4.0. Technical report, DIKU, University of Copenhagen, Denmark, 1991.

[Bon93] A. Bondorf. Similix manual, system version 5.0. Technical report, DIKU, University of Copenhagen, Denmark, 1993.

[Con88] C. Consel. New insights into partial evaluation: The Schism experiment. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 236–246. Berlin: Springer-Verlag, 1988.

[Con93] C. Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 145–154. New York: ACM, 1993.

[KW92] M. Katz and D. Weise. Towards a new perspective on partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 29–36. New Haven, CT: Yale University, 1992.

[Laf90] Y. Lafont. Interaction nets. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 95–108. ACM, January 1990.

[Ler93] X. Leroy. *The CAML-light Reference Manual.* INRIA-ENS, March 1993. Version 0.6.

[Mog93] T. Mogensen. Constructor specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 22–32. New York: ACM, 1993.

[Ses85] P. Sestoft. The structure of a self-applicable partial evaluator. Technical Report 85/11, DIKU, University of Copenhagen, Denmark, 1985.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 165–191. Berlin: Springer-Verlag, 1991.