Trends in Specialization of Interpreters using Offline Narrowing-Driven Partial Evaluation.

Gustavo Arroyo
Centro Interdisciplinario de Investigación y Docencia en Educación Técnica (CIIDET), Av. Universidad 282 Pte., Col. Centro, CP 76000, Querétaro, México
garroyo@ciidet.edu.mx

J. Guadalupe Ramos
Instituto Tecnológico de la Piedad, Av. Tecnolgico No. 2000, Col. Meseta de los laureles, CP 59300, La Piedad, Michoacán, México
guadalupe@dsic.upv.es

Abstract

The search of compilation by specialization of interpreters is a source to source program transformation which has inspired the work of scientists in partial evaluation from many years ago. Narrowing-driven Partial Evaluation (NPE) is a powerful technique for the specialization of functional logic programs. Recent advances in research of offline NPE schemes allow us to develop partial evaluators that process bigger programas. In this work we introduce the stages of a novel pure offline partial evaluator developed in the functional logic language Curry which is able to specialize FlatCurry (the intermediate representation of Curry) programs. In particular, we describe the first experiments in the specialization of interpreters. Our partial evaluator specializes more realistic programs than previous versions since it allows the processing of programs including built-ins and constraints.

Categories and Subject Descriptors D [*PROGRAMMING LANGUAGES, LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages, Processors

Keywords Offline partial evaluation, specialization, interpreters, compilation, narrowing

1. Introduction

It is well known that there are two methods to formally describe the semantics of a programming language. The first one is by describing the translation process from a source language into another target language whose semantics is already known, i.e, the description of a translator. The second one is by describing a procedure to evaluate statements that belong to the language to be defined, i.e., the description of an interpreter [12]. While interpreters are easier to write and maintain, they are inefficient. On the other hand the executable program of a compilation is more efficient, but is more expensive to implement.

One way to get the best of both approaches is to implement a specialization of an interpreter, to automatically generate an efficient implementation [33].

Partial evaluation of programs is a formal technique for specialization and optimization of programs based on semantics which has been investigated within different programming paradigms and applied to a wide variety of languages. Also known as a source-to-source computer programs transformation technique to specialize a program with respect to a part of their input (hence also called program specialization).

Writing interpreters and compilers performance can be connected by partial evaluation; the advantages of prototyping interpreters in general, and the efficiency of the compilers are gained. Because the net effect of the specialization of an interpreter (to a program) is the compilation [5].

A partial evaluator takes a program and part of its input data (known as static data) and try to perform (*reduce*) all the computations that are possible from such data. The partial evaluator returns a new program called *residual* program, which usually runs more efficiently than the original program, since the computations that depend on static data have been made during the partial evaluation itself once and for ever [20].

Partial evaluators can be broadly classified into online and offline. Online partial evaluators decide dynamically during specialization which operations to reduce. As offline partial evaluator processes an annotated program, in which the annotations determine whether an operator is to be applied or not.

Partial evaluation has been extensively applied in the area of functional programming [10, 20, 34] and logic programming [13, 23, 26, 28], where it is commonly known as partial deduction.

In [1] the foundations of *narrowing-driven partial evaluation* (NPE) are examined, previously *narrowing* was originally introduced by Slagle [32] as a mechanism for theorem proving which is a sound and complete method for solving equations with respect to a set of rules *confluent* and *terminating* [19]; this is an enough reason to use *narrowing* as a basic principle for defining the execution semantics of functional logic programs [14].

NPE [1] is a powerful technique of specialization for the first order component of many functional languages like Haskell [29] and functional logic as Curry [15]. At NPE, using a refinement of *narrowing* [32] to perform symbolic computation, being *needed narrowing* [6] the strategy that has better properties. In [4] state that NPE is formalized within the theoretical framework established in [26] and by Martens and Gallagher in [27] for deduction of logic programs, although several concepts have been generalized to deal with functional features such as user-defined functions, eager and lazy evaluation strategies, and deterministic reduction steps. In general, the narrowing space of terms may be infinite. However, even in this case, NPE may end when the original program is *quasi-terminating* with respect to the *narrowing* strategy consid-

ered, i.e., when only finitely many different terms—modulo variable renaming—are computed.

An offline approach to narrowing-driven partial evaluation has been introduced in [30]. In order to improve its accuracy, [7] adapts a *size-change analysis* [24] to the setting of narrowing, this analysis is then used to identify a particular form of quasi-termination[1]. The output of a standard binding-time analysis is also used in order to provide information on which function arguments are static (and thus ground) and which are dynamic, so the combined use of size-change graphs and binding-time analysis; let to infer if the program quasi-terminates as well which ones problematic fragments of code must be annotated to be generalized at partial evaluation time.

In this work, we present the specialization of interpreters written in the functional logic language Curry with first order functions definitions using the improved offline approach to narrowing-driven partial evaluation of [7], we include also an extension to consider several built-ins and constraints. The paper is organized as follows. Section 2 introduces our pure offline partial evaluator named **mixpo**. Then, Section 3 presents an explanation of the implementation of interpreters. Section 4 shows the specialization of interpreters with built-ins and constraints. In Section 5 a discussion of some related work is included and finally Section 6 concludes.

2. Pure offline partial evaluator (mixpo)

In this section, we introduce a description of the stages, data and processes of our offline partial evaluator. For this, we refer to some notions related to the setting of narrowing driven partial evaluation which were considered in our implementation.

Online vs offline. Online partial evaluators perform a single monolithic process [10] which combines symbolic evaluation, propagation of partial values and a dynamic analysis to ensure the termination of the process. In fact, the so called narrowing-driven approach [1] to partial evaluation was originally introduced as an online method.

On the other hand, offline partial evaluators have two clearly separate stages, i.e., a first stage commonly known as binding-time analysis (BTA) which aims to include some annotations to the program to guide the specialization process.

Binding Time Analysis. A BTA usually performs a static analysis to propagate the known (abstract) values throughout the entire program. In our case, the BTA computes a fixed point on the arguments of the program functions from the initial known values of a function call in order to specialize by considering only static (`S`) and dynamic (`D`) abstract values[2].

The result of our BTA is monovariant, i.e., a single sequence of binding-times is associated to the arguments of each program function—a monovariant *division*—(see, eg, [20]).

We call our offline partial evaluator as pure or 100% offline (**mixpo**) because all decisions about partial evaluation termination (well known like control issues) are taken before the proper specialization process. Thus, partial evaluation process is entirely directed by the annotations which were added during the BTA, i.e., the termination analysis is carried out at an early stage.

Additionally, our BTA considers a recent approach to analyze the behavior of functions arguments in successive function calls.

Size Change Analysis. The *size-change analysis* [24] adapted in [7] to the setting of narrowing, is then used to detect potential sources of non-termination, so that the arguments that may introduce infinite loops at partial evaluation time are annotated to be generalized, i.e., replaced by fresh variables, in the partial evaluation stage.

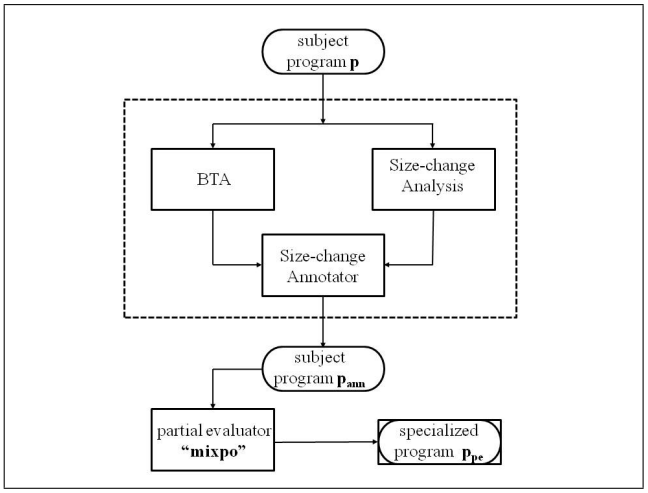

Figure 1. Outline offline partial evaluator (**mixpo**)

Size-change analysis lets to find out which function arguments into the function calls are increasing and hence they could cause infinite loops, for this, the analysis builds a set of size-change (multi) graphs that depict the arguments behavior. In this setting an *idempotent multigraph* describes the arguments in a cycle, i.e., how is the behavior (decreasing or increasing) of the arguments departing from a certain function and returning to the same function and hence completing a cycle (Observe Figure 3).

Roughly speaking, we use size-change graphs to approximate the changes in parameter sizes from one function call to another. In fact, we use the the size-change graphs information to identify a particular form of quasi-termination [11] called PE-termination in [7], which ensures that only finitely many different function calls can be produced in a computation.

We were inspired by the Theorem 1 and PE-termination Definition of [7] to implement the annotation process, this allows us to ensure quasi-terminating computations and as a consequence the finiteness of the partial evaluation process.

The source language. Now we present the source language of subject programs to be specialized. The syntax of *flat* programs [17] has been successfully applied for the representation of functional logic programs which makes explicit the pattern matching strategy by case expressions. This flat representation constitutes the core of modern functional logic languages like Curry. In the setting of Curry, *FlatCurry* is the flat representation of source programs, and hence, as for annotation as for specialization we compute FlatCurry programs. Indeed, one feature of Curry programs is that they are automatically translated and stored in flat representation [18].

We use a subset of the abstract syntax for programs in the flat representation as follows:

$\mathcal{R}$	::=	$\mathcal{D}_1 \ldots \mathcal{D}_m$	(program)
$\mathcal{D}$	::=	$f(\overline{x_n}) = e$	(function definition)
e	::=	x	(variable)
	|	$c(\overline{e_n})$	(constructor call)
	|	$f(\overline{e_n})$	(function call)
	|	$case\ e\ of\ \{\overline{p_n \rightarrow e_n}\}$	(rigid case)
	|	$fcase\ e\ of\ \{\overline{p_n \rightarrow e_n}\}$	(flexible case)
p	::=	$c(\overline{x_n})$	(pattern)

[1] A computation quasi-terminates when it only contains a finite number of different terms modulo variable renaming

[2] We consider the conventional values obtained from the binding-time analysis as *static* (absolutely known at partial evaluation time) and *dynamic* (probably known at partial evaluation time)

```
data Nat = Z | S Nat
main x = prod (x) (fib (S (S (S (S (S Z))))))

fib x = case x of
        Z -> Z
        (S Z) -> (S Z)
        (S (S n)) -> sum (fib (S n)) (fib n)

prod x y = case x of
           Z -> Z
           (S w) -> sum y (prod w y)

sum x y = case x of
          Z -> y
          (S w) -> S (sum w y)
```

Figure 2. `Fibonacci` example for natural numbers

Here, a term like $\overline{o_n}$ represents a sequence of objects $o_1, \ldots, o_n$. Thus, a program $\mathcal{R}$ consists of a sequence of function definitions $\mathcal{D}$ such that the left-hand side is linear and has only variable arguments The right-hand side of each function definition is an expression e composed by variables ($\mathcal{V}$), constructors ($\mathcal{C}$), function calls ($\mathcal{F}$), and case expressions for pattern matching, i.e., pattern matching is compiled into case expressions. Variables are denoted by $x, y, z \ldots$, constructors by $a, b, c \ldots$, and defined functions by $f, g, h \ldots$ The difference between $case$ and $fcase$ shows up when the argument e is a free variable: $case$ suspends (which corresponds to *residuation*) whereas $fcase$ nondeterministically binds this variable to the pattern in a branch of the case expression and proceeds with the appropriate branch (which corresponds to *narrowing*).

2.1 The structure of mixpo

Figure 1 shows the complete scheme of our implementation, we can see that the process contained in the dotted box includes both simple BTA and size-change analysis, both processes take as input the program (**p**) to be specialized and deliver their results to the annotation process itself, that produces the annotated program ($\mathbf{p_{ann}}$). Then, the partial evaluator (**mixpo**) receives $\mathbf{p_{ann}}$ and performs the proper partial evaluation process to produce a final specialized program $\mathbf{p_{pe}}$.

We present an annotation procedure that is based originally on the quasi-termination analysis of [7], and it was improved at [8] (It is available at `http://users.dsic.upv.es/~garroyo/`). Moreover, since this quasi-termination analysis was originally introduced for TRSs, it was adapted to the *flat* language in [8] also.

We use the same definitions of the quasi-terminations analysis of [8], just make a slight change in the program annotation procedure.

EXAMPLE 1. *Consider the* `fibonacci` *example for natural numbers shown in Fig. 2. In general we have calculated eight size-change graphs, but the size-change analysis yields as result three idempotent multigraphs shown in figure 3.*

In this case we base on the Theorem 6 of [8], but now we do not consider the right-linearity of the dynamic variables. So now the program is annotated by replacing every rule $f(\overline{x_n}) = e$ in $\mathcal{R}$ by the new rule $f(\overline{x_n}) = ann^g(ann^u(e))$, where ann stands for annotation, u for unfolding and g for generalization. We have even changed the annotation criteria for generalization, i.e., we do not add annotations for generalization anymore. Now we create a vector for generalization in the annotated program which indicates to the specialization procedure, which terms should be generalized (see the constructor `BtD` in Figure 4).

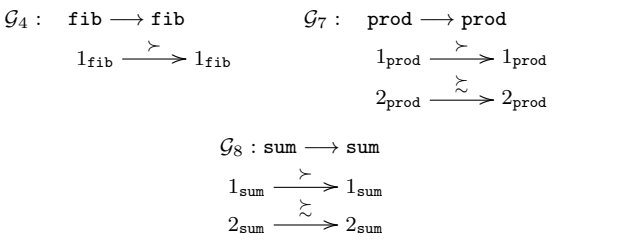

Figure 3. Idempotent multigraphs for `fibonacci`

```
data Nat = Z | S Nat

GEN2 = BtD ("prod", 2, [D, S])

main v1 = MEM (prod v1 (UNF (fib (S (S (S (S (S Z))))))))

fib eval rigid
fib Z = Z
fib (S Z) = (S Z)
fib (S (S v3)) = UNF (sum (UNF (fib (S v3))) (UNF (fib v3)))

prod eval rigid
prod Z v2 = Z
prod (S v3) v2 = UNF (sum v2 (MEM (prod v3 v2)))

sum eval rigid
sum Z v2 = v2
sum (S v3) v2 = S (UNF (sum v3 v2))

UNF v1 = v1
MEM v1 = v1
```

Figure 4. Annotated `fibonacci` example

EXAMPLE 2. *Consider again the* `fibonacci` *example for natural numbers shown in Fig. 2. Let us consider a call to the annotation stage with this program and the abstract initial value* [D] *for the argument of* `main` *definition. Our BTA returns the following division:*

$$\{\texttt{main} \mapsto (\texttt{D}),\quad \texttt{fib} \mapsto (\texttt{S}),\quad \texttt{prod} \mapsto (\texttt{D},\texttt{S}),\quad \texttt{sum} \mapsto (\texttt{S},\texttt{D})\}$$

where S *denotes that an argument is static (ground) and* D *that it is dynamic. Here, our annotation procedure returns the annotated program shown in figure 4. According to the function ann^u of [8], we can annotate* `fib` *and* `sum` *with* `UNF`*; because both have at least one edge $i_f \overset{\succ}{\longrightarrow} i_f$ in their respective idempotent multigraphs such that i_f is static.* `prod` *is annotated with* `MEM` *following the function ann^u also, i.e., does not meet the first three alternatives of ann^u. About the function ann^g; at least the first argument of* `prod` *must be annotated with* `GEN`*, however to meet with this requirement we add a tuple to the vector for generalization. This tuple should contain: function name, arity and the values* [S] *or* [D] *for the arguments, where* [D] *indicates that the corresponding term must be generalized in the specialization stage.*

So far, we have described just the annotation stage. Taking the specialization stage, we first recall the generic procedure for NPE [1], shown in figure 5, which is similar to the Gallager's procedure for the partial evaluation of logic programs [13]. So given a set of operation-rooted[3] terms, i.e., function calls, the procedure applies an unfolding operator *unfold* such that a residual rule set is

[3] An expression is *operation-rooted* if it is rooted by a defined function symbol

Input: a program $\mathcal{R}$ and a set of calls T
Output: a set of calls S
Initialization: $i := 0;\ T_0 := T$
Repeat
$\mathcal{R}' := unfold(T_i, \mathcal{R})$;
$T_{i+1} := abstract(T_i, \mathcal{R}'_{calls})$;
$i := i + 1$;
Until $T_i = T_{i-1}$ (modulo variable renaming)
Return: $S := T_i$

Figure 5. Generic procedure for NPE

returned. This *unfold* operator is driven by the `UNF` annotations, so that all functions annotated with `UNF` should be unfolded because according to the idempotent multigraphs originating the annotation; there is at least one argument with a strict order of reduction ($\succ$) and also static, meaning that its value is known at the time of specialization, it certainly ensures that the (partial) evaluation of the function ends, while functions annotated with `MEM` should suspend the evaluation of the call and return to the global control.

The derivations that are computed in the process of specialization in the local control are based on a slight extension of the RLNT calculus [3]. RLNT calculus is a non-standard semantics used to partially evaluate the function calls and describes basic computing operations of flat programs, such that RLNT specialized programs were originally processed just in *online* way. In the offline setting, we require to add annotations to functions and their arguments (*offline*); Indeed, RLNT calculus was extended to support these annotations, this extended semantics can be found at [8]. Each computation performed with the RLNT calculus; generates *residual rules* which will compose the residual program.

On other hand the abstraction operator *abstract* takes a finite set of operation-rooted terms T_i and then properly adds the set of operation-rooted subterms in the right-hand sides of the unfolded calls, which is denoted by $\mathcal{R}'_{calls}$. The new set T_{i+1} may need further evaluation and, thus, the process is iteratively repeated while new terms are introduced.

About vector for generalization, when the global control finds a term operation-rooted with subterms marked with `D` into this vector; the term should be flattened before adding the terms to the set of calls to be evaluated [30]. By instance, given the term $f\ (g\ (x),\ h(y))$, if the generalization vector contains a tuple with ("`f`", 2, [`D`, `D`]); function calls $f(w1, w2)$, $g(x)$ and $h(y)$ are added to the current set of function calls (to be) partially evaluated, i.e., T_{i+1}, where $w1, w2$ are fresh variables corresponding to the generalization of the terms $g(x)$ and $h(y)$.

Intuitively speaking, the source program of Figure 2 shows that function `fib` can be fully evaluated, i.e., the program can be specialized just evaluating the function `fib`. If we use the annotated program of Figure 4, we may see that you can fully unfold the both function as `sum` as `fib` because both are annotated with `UNF`. If you specialize the annotated program with our partial evaluator (**mixpo**) gets the program shown in Figure 6, (**mixpo**) leaves the specialized program on "FlatCurry" language, however PAKCS [18] is able to show this program in Curry. We can see, in fact, have been fully evaluated `sum` and `fib`. Likewise, the function `prod` has been partially evaluated. In short, the result indicates that for each unit of `main` variable should add five units to the right hand side of `prod_pe1 Z`, i.e., to zero.

3. Implementation of interpreters

In this Section we introduce details of the implementation of interpreters, but first we present information about the meta-programming facilities of Curry [16].

```
module fib2_ann (Nat(Z,S), main, prod_pe1) where

data Nat = Z | S Nat

main :: b-> a
main v0 = prod_pe1 v0

prod_pe1 :: b-> a
prod_pe1 eval rigid
prod_pe1 Z = Z
prod_pe1 (S v6) = S (S (S (S (S (prod_pe1 v6)))))
```

Figure 6. Specialized `fibonacci` example

Meta-programming in Curry The implementation of our partial evaluator and our Curry interpreter relies on the meta-programming facilities of the language Curry. In particular, we consider the intermediate language FlatCurry for representing functional logic programs (available by using the Curry libraries FlatCurry and FlatCurryTools). In FlatCurry, all functions are defined at the top level (i.e., local function declarations in source programs are globalized by lambda lifting) and the pattern matching strategy is made explicit by the use of case expressions. In this setting, a FlatCurry program is represented by means of the following data type:

```
data Prog = Prog String    --name of module
                 [String]  --imported modules
                 [TypeDecl] --type declarations
                 [FuncDecl] --function declarations
                 [OpDecl]   --operator declarations
```

For simplicity, here we only show the data types for representing function declarations:

```
data FuncDecl = Func QName      --qualified name
                     Int        --arity
                     Visibility --public/private
                     TypeExpr   --type
                     Rule       --rule
data Rule = Rule [VarIndex] Expr
```

Therefore, each function is represented by a single rule whose left-hand side contains different variables (`[VarIndex]`) and whose right-hand side is an expression containing variables, literals, function and constructor calls, disjunctions, and case expressions:

```
data Expr = Var VarIndex
          | Lit Literal
          | Comb CombType QName [Expr]
          | Or Expr Expr
          | Case CaseType Expr [BranchExpr]
data CombType   = FuncCall | ConsCall
data CaseType   = Rigid | Flex
data BranchExpr = Branch Pattern Expr
data Pattern    = Pattern QName [VarIndex]
                | LPattern Literal
```

Description of interpreters implementation Reasoning that an interpreter is a kind of operational semantics of low-level, and therefore may serve as a definition of a programming language [20], so in Figure 7 we show the simple operational semantics which should follow our interpreters on an informal basis. However, it is really based on the basic RLNT calculus of [2] and on the syntax subset of flat language shown in Section 2. We believe this is enough to implement our interpreters to be specialized. It is necessary to say that we do not consider some features of Curry like: higher-order neither concurrent programming for these implementations but built-ins and constraints are.

Let's briefly describe our operational semantics: The symbols "$[\![$" and "$]\!]$" in an expression like $[\![e]\!]$ do not denote a semantic function but are only used to identify which part of an expression

HNF

$$[\![e]\!] \Rightarrow \text{e} \in \mathcal{V} \text{ or } e = c() \text{ with } c \in \mathcal{C}$$
$$[\![c(e_1,\ldots,e_n)]\!] \Rightarrow c([\![e_1]\!],\ldots,[\![e_n]\!])$$

Function Eval

$$[\![g(\overline{e_n})]\!] \Rightarrow [\![\sigma(r)]\!] \text{ if } g(\overline{x_n}) = r \in \mathcal{R} \text{ is a function definition with fresh variables and } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case Eval

$$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow [\![(f)case\ e'\ of\ \{\overline{p_k \to e_k}\}]\!] \text{ if } [\![e]\!] \Rightarrow [\![e']\!],\ e \notin \mathcal{V},\ root(e) \notin \mathcal{C}, \text{ and } e \neq (f)case\ (\_)\ of\ \{\ldots\}$$

Figure 7. Simple operational semantics for interpreters

should be still evaluated. The two rules classified into HNF can be applied when the considered term is in head normal form, i.e., it is a variable or a constructor-rooted term. Function Eval, in the original RLNT calculus performs the unfolding of a function call, this is a purely functional unfolding since all arguments in the left-hand sides of the rules are variables. However, since we are considering some built-ins and constraints like "==", "+" , "-" , "*"; where they are functions, the expressions with these symbols could be evaluated at once. So we make directly some actions to apply these built-ins into the interpreter with the aim of having an interpreter simple and easy, for the time being. **mixpo** is able also to process some built-ins and constraints. We will see some examples of interpreters in Section 4.

Case Eval first evaluates the case argument by creating a call for this subterm, as you can see this rule excludes the evaluation of a case expression whose argument is either a variable, a constructor-rooted term, or any another case expression. By the moment we just have implemented some cases of e', i.e., some cases as resulting of the case expression. Keeping in mind getting a relatively simple implementation of interpreters.

As it can seen in the `fibonacci` example, it has a `main` function definition, **mixpo** looks for the default function `main` to start the specialization, so if we want to specialize an interpreter w.r.t. a *program* it must be included into the same interpreter; we put in the right hand side of `main` the function call to the interpreter with the following arguments: a boolean flag, the *program* and the expression to be evaluated. Previously we must translate the *program* to FlatCurry to include just its list of functions in *flat* format. Let us explain this, in the appendix A, an example of a Curry simple interpreter can be seen; there, we have the function:

```
int p e = do (Prog _ _ _ funs _ ) <- readFlatCurry p
             print (ieval True funs e)
```

which is enclosed by {- -}. It is only used to test the interpreter, that is the reason why it is commented. `int` does two actions; reads the *program* and prints the evaluation of expression `e`. So we give just this structure into the interpreter as the *program* to be interpreted, `ieval` is the main function of the interpreter. If you see the right hand side of `main` function, one of `ieval` arguments is:

```
 [Func ("rev2","main") 0 Public (TCons ("Prelude","Int") [ ])
(Rule [ ] (Comb FuncCall ("Prelude","+") [Comb FuncCall
("Prelude","-") [Comb FuncCall ("Prelude","*") [Lit (Intc 3),
Lit (Intc 4)],Lit (Intc 1)],Lit (Intc 2)]))]
```

which represents the functions list in FlatCurry of the following Curry program: `main = 3*4-1+2` commented also by {- -} in the example. You may specialize this simple program with **mixpo**, able to specialize built-ins, as we hope you get the following program:

```
-- Program file: metaintf/examples/rev2_ann_pe
module rev2_ann(main) where
main :: a
main = 13
-- end of module metaintf/examples/rev2_ann_pe
```

Now, if you specialize the simple interpreter of the appendix, i.e., first to annotate and then specialize, you get the following program:

```
module int_arith_ann(main,ieval_pe1) where
import FlatCurry

main :: a
main = ieval_pe1

ieval_pe1 :: a  ieval_pe1 =
    FlatCurry.Comb FlatCurry.FuncCall ("Prelude","+")
      [FlatCurry.Comb FlatCurry.FuncCall ("Prelude","-")
         [FlatCurry.Lit (FlatCurry.Intc 12),FlatCurry.Lit
           (FlatCurry.Intc 1)],
                FlatCurry.Lit (FlatCurry.Intc 2)]
```

Striking that all the code of the interpreter has been removed, but we have as a result of this partial evaluation just two function definitions: the `main` function and the `ieval` function partially evaluated.

Continuing with the implementation description of interpreters structure (which refers to the example of Appendix A), we have two rules to evaluate a constructor-rooted terms, only one of them unfolds, the rule with `True` argument, and stops until HNF is reached. `ievalList` evaluates the function/constructor expression list. See the following rules:

```
ieval True v2  (Comb ConsCall v8 v9) =
                        (Comb ConsCall v8 (ievalList True v2 v9) )
ieval False _   (Comb ConsCall c es) = Comb ConsCall c es
```

While `ieval False _ (Comb ConsCall c es)` is used to have some control over the evaluation of the `case` expression argument, i.e., to restrict the unfolding of the possible `ConsCall` expressions. Let us check the following segment of Curry code:

```
ieval top funs (Case ctype e ces) =
    case (ieval False funs e) of
     Comb ConsCall f es -> ieval top funs (matchBranch ces f es)
     Lit l -> ieval top funs (matchBranchLit ces l)
```

For the sake of simplicity, we take into account only two possible cases of the resulting `Case` expression, both results make a call to the evaluation of the respective branch of the `Case`, depending on the resulting `Case` expression the branch is selected by the functions `matchBranch` or `matchBranchLit`.

Regarding the evaluation of functions, after many tests, we decided to check built-ins assessment prior to unfold the function. If the top of the function call refers to the symbols: "==", "+" , "-", "*" or "failed"; the interpreter could make a call to: `ieval_EQ`, `ieval_ARITH` or returns a ''`failed`'' expression, otherwise the interpreter makes a call to unfold, i.e., calls `ieval top funs (matchiRHS funs (mn,f) es)`. You may verify this in the following rule:

```
ieval top funs (Comb FuncCall (mn,f) es) =
    if mn == "Prelude"
     then if f == "failed"
       then Comb FuncCall (mn,f) es
       else if f == "=="
             then (ieval_EQ top funs (mn,f) es)
             else case f of
                      "+" -> ieval_ARITH top funs (mn,f)  es
                      "-" -> ieval_ARITH top funs (mn,f)  es
                      "*" -> ieval_ARITH top funs (mn,f)  es
     else ieval top funs (matchiRHS  funs (mn,f) es)
```

`ieval_EQ` evaluates a simple case of strict equality, if the arguments are not integer literal, it tries to evaluate the list of arguments. `ieval_ARITH` evaluates the simple arithmetic functions, if the arguments are not integer literal, as done `ieval_EQ`, `ieval_ARITH` aims to evaluate the list of arguments.

`matchiRHS`, first looks for the function to unfold into set of functions structure, i.e. into the *program*. When the interpreter finds the function rule, returns the expression on the right hand side of that rule with the substitution of all occurrences of variables on the

left hand side of the same rule by the expressions that apply. See the following rules:

```
matchiRHS [ ] (_,_) _ = Comb FuncCall ("Prelude","failed") [ ]
matchiRHS (Func (_,fname) _ _ _ funrule : fds) (mn,name) es =
   if fname==name then matchRHS_aux funrule es
                  else matchiRHS fds (mn,name) es

matchRHS_aux (Rule vars rhs) es = substitute vars es rhs
```

To understand a little what happens with the substitution of variables by the corresponding expressions see example 3.

EXAMPLE 3. *In this example we are going to describe how* **mixpo** *unfolds. We know our interpreters unfold the same way that* **mixpo**. *So consider the annotated program of Figure 4. To start the specialization of any annotated program,* **mixpo**, *looks for the function* `main` *by default, in this case* **mixpo** *builds the following call:*

```
[(Comb FuncCall ("fib_ann","main") [(Var 0)])]
```

that is in FlatCurry, which corresponds to (`main v0`) *in Curry source. We have put a trace into* **(mixpo)** *to know the calls to unfold, the first three are:*

```
unfoldpo: (main v0)
unfoldpo: (MEM (prod v0 (UNF (fib (S (S (S (S (S (Z ))))))))))
unfoldpo: (MEM (prod v0 (Case (S (S (S (S (S (Z ))))))) of
   (Z  -> (Z ))
   (S v105 -> (Case v105 of
     (Z  -> (S (Z )))
     (S v106 -> (UNF (sum (UNF (fib (S v106)))
                          (UNF (fib v106)))))
  ))
)))
```

Given the call (`main v0`), `matchiRHS`, *first looks for this function to unfold. You can see that the right hand side of* (`main v0`) *is the expression of second call to unfold and, the third call to unfold corresponds the same expression of the second call to unfold but with the* ((`fib` (`S` (`S` (`S` (`S` `Z`)))))) *expression unfolded. Note that the latter call to unfold there is a* `Case` *expression which corresponds to the right hand side of the function* `fib`, *in flat format, remember that annotation and specialization is done in this format. But what about substitutions? The first substitution happens when* `matchiRHS` *finds the structure of the function* `main`, *then the variables on the right hand side indicated at the left of* `main`, *in this case* [1] *are replaced by expressions of the call to unfold, in this case the expression list of the call is* [(`Var 0`)], *see the call* (`main v0`) *in flat format. Thus* `v1` *is replaced by* `v0` *and is shown in the result of the first unfold. The second substitution happens when* ***mixpo*** *requires the second unfold:*

```
(MEM (prod v0 (UNF (fib (S (S (S (S (S (Z ))))))))))
```

first `matchiRHS` *tries to unfold the function* `fib`, *so finds the structure of this function, some code like:*

```
fib v1 =  case v1 of
   (Z  -> (Z ))
   (S v2 -> (case v2 of
     (Z  -> (S (Z )))
     (S v3 -> (UNF (sum (UNF (fib (S v3))) (UNF (fib v3)))))))
```

then the variables on the right hand side indicated at the left of `fib`, *in this case* [1] *are replaced by expression of the call to unfold, in this case the expression list of the call is* (`S` (`S` (`S` (`S` `Z`))))), *thus* `v1` *is replaced by* (`S` (`S` (`S` (`S` (`S` `Z`))))) *resulting a code very similar to that shown in the third trace of call to unfold in this example.*

In general terms this is the implementation description of interpreters structure.

4. Specialization of interpreters with built-ins and constraints

You may compile through the specialization of an interpreter that runs just one fixed-source *program*, producing a target program in the partial evaluator's output language, besides compiling by partial evaluation always generates correct object code [20]. So if we have a FlatCurry *program* $\mathtt{p}_{\mathtt{fcy}}$ that accepts a function call with static and dynamic data arguments included into the same *program*, and a program $\mathtt{int}_{\mathtt{Cy}}$ written in Curry language to interpret *programs* in the intermediate FlatCurry language that produces target programs $\mathtt{tgt}_{\mathtt{fcy}}$ in FlatCurry also. Thus we may represent a particular interpretation process as:

$$\mathtt{tgt}_{\mathtt{fcy}} \;= [\![\ \mathtt{int}_{\mathtt{Cy}}\ ]\!]\ [\mathtt{p}_{\mathtt{fcy}}]$$

Our offline partial evaluator **mixpo** accepts annotated programs in FlatCurry, thus the *first Futamura projection* (very similar to the definition of [20]) may be represented like:

$$\mathtt{tgt}_{\mathtt{fcy}} \;= [\![\ \mathbf{mixpo}]\!]\ [\mathtt{int}_{\mathtt{Cy}}\ \mathtt{p}_{\mathtt{fcy}}]_{\mathtt{fcy}}$$

We include the *program* to be interpreted and a call to this *program* into the same interpreter as arguments of the interpreter's call, you can see in the appendix the right hand side of `main` function definition, i.e., the `ieval` function call. In [5] also use different data (types) structures for programs and data input in order to compile by partial evaluation.

Let us analyze the specialization of an interpreter with the use of the conjunction (sequential); the built-in type `&&`. The interpreter function `main` looks like:

```
{-
 main = (3 == 3) && (1 == 1)
-}
 main = ieval True [Func ("andexam","main") 0 Public (TCons
("Prelude","Bool") [ ]) (Rule [ ] (Comb FuncCall ("Prelude","&&")
[Comb FuncCall ("Prelude","==") [Lit (Intc 3),Lit (Intc 3)],
Comb FuncCall ("Prelude","==") [Lit (Intc 1),Lit (Intc 1)]]))]
(Comb FuncCall ("andexam","main") [ ])
```

If we require to PAKCS the evaluation of the expression `(3 == 3) && (1 == 1)`, "True" is the result. In order to interpret this expression we need to make some changes, first the evaluation of functions must be able to process the built-in type `&&`, so we change this part of interpreter code:

```
ieval top funs (Comb FuncCall (mn,f) es) =
    if mn == "Prelude"
     then if f == "failed"
       then Comb FuncCall (mn,f) es
       else if f == "=="
             then (ieval_EQ top funs (mn,f) es)
             else case f of
                     "+" -> ieval_ARITH top funs (mn,f)  es
                     "-" -> ieval_ARITH top funs (mn,f)  es
                     "*" -> ieval_ARITH top funs (mn,f)  es
                     "&&" -> ieval_SAND top funs (mn,f)  es
     else ieval top funs (matchiRHS  funs (mn,f) es)
```

Notice that we added the call `ieval_SAND` precisely to assess the referenced built-in. This last function definition is shown below:

```
ieval_SAND :: Bool -> [FuncDecl] -> QName -> [Expr] -> Expr
ieval_SAND    top       fns          (mn,fn)  [e1,e2] =
   case (ieval True fns e1) of
        (Comb ConsCall (Pre,Tru) [])-> (ieval top fns e2)
        (Comb ConsCall (Pre,Fal) []) ->
                          (Comb ConsCall (Pre,Fal) [])
```

From experience we have seen a string specializes more easily if we define a simple declaration. So we have defined the following declarations:

```
Pre = "Prelude"
Fal = "False"
Tru = "True"
```

After specializing the interpreter with the above changes we obtain the following program:

```
module int_sand_ann(main,ieval_pe1) where
import FlatCurry
main :: a
main = ieval_pe1
ieval_pe1 :: a
ieval_pe1 eval rigid
ieval_pe1 = case (Comb ConsCall ("Prelude","True") []) of
       Comb v141 v142 v143 -> case v141 of
         ConsCall -> case v142 of
           (v144,v145) -> case v143 of
             [] -> Comb ConsCall ("Prelude","True") []
```

You may verify the result loading this FlatCurry program to PAKCS then requiring the evaluation of `main` function that:

```
Comb ConsCall ("Prelude","True") []
```

is the result of the requirement.

So far we have seen the partial evaluation of simple programs without dynamic arguments, now see the specialization of the interpreter containing the fibonacci program for natural numbers (shown at Figure 2). As in the latter interpreter partial evaluation we must change the program to interpret. So the interpreter function `main` now looks like:

```
main x = ieval True  [Func ("fib7","main") 1 Public (FuncType
(TCons ("fib7","Nat") []) (TCons ("fib7","Nat") [])) (Rule [1]
(Comb FuncCall ("fib7","prod") [Comb FuncCall ("fib7","fib")
[Var 1],Comb FuncCall ("fib7","fib") [Comb ConsCall ("fib7","S")
[Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","S") [Comb
ConsCall ("fib7","S") [Comb ConsCall ("fib7","S") [Comb ConsCall
("fib7","Z") [][][][][][]]),  Func ("fib7","fib") 1 Public (FuncType
(TCons ("fib7","Nat") []) (TCons ("fib7","Nat") [])) (Rule [1]
(Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") []) (Comb
ConsCall ("fib7","Z") []),Branch (Pattern ("fib7","S") [2]) (Case
Rigid (Var 2) [Branch (Pattern ("fib7","Z") []) (Comb ConsCall
("fib7","S") [Comb ConsCall ("fib7","Z") []]),Branch (Pattern
("fib7","S") [3]) (Comb FuncCall ("fib7","sum") [Comb FuncCall
("fib7","fib") [Comb ConsCall ("fib7","S") [Var 3]],Comb FuncCall
("fib7","fib") [Var 3]])])])),  Func ("fib7","prod") 2 Public
(FuncType (TCons ("fib7","Nat") []) (FuncType (TCons ("fib7",
"Nat") []) (TCons ("fib7","Nat") []))) (Rule [1,2] (Case Rigid
(Var 1) [Branch (Pattern ("fib7","Z") []) (Comb ConsCall ("fib7",
"Z") []),Branch (Pattern ("fib7","S") [3]) (Comb FuncCall ("fib7"
,"sum") [Var 2,Comb FuncCall ("fib7","prod") [Var 3,Var 2]])])),
Func ("fib7","sum") 2 Public (FuncType (TCons ("fib7","Nat") [])
(FuncType (TCons ("fib7","Nat") []) (TCons ("fib7","Nat") [])))
(Rule [1,2] (Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") [])
(Var 2),Branch (Pattern ("fib7","S") [3]) (Comb ConsCall ("fib7",
"S") [Comb FuncCall ("fib7","sum") [Var 3,Var 2]])]))]
(Comb FuncCall ("fib7","main") [x])
```

You can see the `x` of function `main` this variable represents a dynamic value of the interpreter, i.e, a value probably known at partial evaluation time. This dynamic value spreads through the interpreter when you run the BTA, so our BTA returns the following division:

```
[("main",1,[D]),("ieval",3,[S,S,D]),("ieval_ARITH",2,[D,D]),
("ieval_EQ",1,[D]),("ieval_args",2,[S,D]),("matchBranch",3,[D,D,D]),
("matchBranchLit",2,[D,D]),("matchiRHS",3,[S,D,D]),
("matchRHS_aux",2,[S,D]),("substituteAll",3,[D,D,D]),
("replaceVar",3,[D,D,D]),("substituteAllArgs",3,[D,D,D]),
("substituteAllCases",3,[D,D,D]),("substituteAllCase",3,[D,D,D])]
```

The size-change analysis (SCA) yields as result thirteen idempotent multigraphs, the following functions: `main`, `ieval_EQ`, `ieval_ARITH` and `matchRHS_aux` are not involved in the result of SCA, i.e., do not have idempotent multigraphs, so the calls to these functions are annotated with `UNF` (during the annotation process) because they do not introduce introduce infinite loops. `matchiRHS` has an idempotent multigraph but it's first edge, i.e., that corresponds to its first argument, has a label with strict order of reduction ($\succ$) and is static. Therefore the `matchiRHS` function calls are also labeled with `UNF`. The rest of the function calls are annotated with `MEM`. Therefore, even without taking into account the vector of generalization, we could deduce that the specialization of this interpreter may not be very good because it has too many functions that can not be unfolded at the time of the partial evaluation. However, generally the result of the partial evaluation of our interpreters that accept a call with dynamic arguments; is a mixture of the interpreter and the source program to interpret, containing parts derived from both [20]. By instance, after the specialization of the interpreter, the rule of the definition `main` becomes the following:

```
main :: b -> a
main v0 = ieval_pe1 (Comb FuncCall ("fib7","main") [v0])
```

Very nice specialization, but `matchiRHS` now has four substitutions identified in the right hand side; the branches to unfold: `main`, `fib`, `prod` and `sum`. We have also obtained two cases for `ieval` definition, i.e., that has grown a bit the size of the code obtained. We would expect to be executed faster than the original interpretation at least, but we have reached almost the same runtime average. The interpretation of the program has certainly improved because the unfold process has been transferred directly to the function that is responsible for do it (`matchiRHS`), i.e., some steps of interpretation have been saved. Although in this case, specialization has not sufficiently reduced the interpreter code because most of the functions can not be unfold to a value.

5. Related Work

Considering only offline partial evaluation that have experimented with the specialization of interpreters are the following: In [21], Jørgensen generates compilers from interpreters by partial evaluation and obtain interpreters (in strict functional languages) from formal languages, he uses Similix; a self-interpreter for large (higher order included [22]) subset of Scheme, so the target language is Scheme but translates from BAWL. The BTA of Similix is monovariant like ours. In [5], Andersen has developed a self-applicable partial evaluator for a significant subset of C language without a BTA, he transforms the program into a intermediate language core C, analogously we translate to FlatCurry, In [5] also was reported the first implementation of self-applicable partial evaluator for an imperative language. Tempo is an offline specializer for C programs [33] able to specialize both in bytecode interpreters as structured language and yields excellent speedups. In [25], Leuschel et. al. present LOGEN as a self-interpreter for logic programs they have achieved the Jones Optimality in a systematic way, as our work they present the partial evaluation[4] process into two phases— BTA and specialization phase, and we use a very similar algorithm for the proper specialization.

6. Conclusion and future work

We have presented the first experiments in the specialization of interpreters using offline narrowing-driven partial evaluation. Our partial evaluation process has an automatic monovariant BTA, a first order termination analysis called size-change analysis and the proper pure offline partial evaluator process (**mixpo**). The binding time analysis processes (BTA, SCA, the annotator) and the specializer are written in Curry. The interpreters accepts FlatCurry programs and the target language is FlatCurry also. The interpreters that runs a simple source programs without dynamic arguments specialize very well. Interpreters running programs with dynamic arguments are also specialized and so far we get a mixture of the interpreter and the source program to interpret but the speedups keeps almost the same average.

There is a lot of work to do to improve this work, because our partial evaluator has higher order functions is not self-applicable, so first we need a higher-order (HO) polivariant BTA and also to implement a HO termination analysis. We have participated in a recent work to make a transformation to polivariant BTA of

[4] The logic programming paradigm introduced strictly the term "partial deduction" to replace the term "partial evaluation"

HO functions by defunctionalization [9], we have to analyze the feasibility of using this transformation with our current termination analysis, i.e. the first order SCA, or may use the extended SCA to HO functional programs [31].

References

[1] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20 (1):3–26, 2002.

[2] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[3] E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters*, 85(1):19–25, 2003.

[4] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[5] L. O. Andersen. Partial evaluation of C and automatic compiler generation (extended abstract). In *Compiler Construction. 4th International Conference. (Paderborn, Germany). Lecture Notes in Computer Science*, pages 251–257. Springer-Verlag, 1992.

[6] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[7] G. Arroyo, J. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Logic-Based Program Synthesis and Transformation. Revised and Selected Papers from LOPSTR'06*, pages 60–76. Springer LNCS 4407, 2007.

[8] G. Arroyo, J. Ramos, S. Tamarit, and G. Vidal. Offline Narrowing-Driven Specialization in Practice. In *ACTAS de las VII Jornadas sobre Programación y Lenguajes (PROLE 07)*, pages 137–146, 2007. II Congreso Español de Infórática (CEDI 2007).

[9] G. Arroyo, J. G. Ramos, S. Tamarit, and G. Vidal. A transformational approach to polyvariant bta of higher-order functional programs. In *LOPSTR*, pages 40–54, 2008.

[10] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[11] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[12] Y. Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.

[13] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.

[14] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20: 583–628, 1994.

[15] M. Hanus. Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~mh/curry/`, 2006.

[16] M. Hanus. Flatcurry: An intermediate representation for Curry programs. `http://www.informatik.uni-kiel.de/~curry/flat/`, May 2011.

[17] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[18] M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.6.0: The Portland Aachen Kiel Curry System—User Manual. Technical report, University of Kiel, Germany, 2004.

[19] J. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[20] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[21] J. Jørgensen. Compiler generation by partial evaluation. Technical report, Masters thesis, DIKU, 1991.

[22] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *POPL*, pages 258–268, 1992.

[23] H. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symp. on Principles of Programming Languages*, pages 255–267, 1982.

[24] C. Lee, N. Jones, and A. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28: 81–92, 2001.

[25] M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising Interpreters Using Offline Partial Deduction. In *Program Development in Computational Logic*, pages 340–375. Springer LNCS 3049, 2004.

[26] J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[27] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of the 12th Int'l Conf. on Logic Programmin (ICLP'95)*, pages 597–611. MIT Press, 1995.

[28] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20: 261–320, 1994.

[29] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[30] J. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 228–239. ACM Press, 2005.

[31] D. Sereni. Termination Analysis and Call Graph Construction for Higher-Order Functional Programs. In *Proc. of the 12th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'07)*, pages 71–84. ACM, 2007.

[32] J. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[33] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in program compilation by interpreter specialization. Technical Report 3588, INRIA, December 1998.

[34] V. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects, 1985*, pages 257–281. Springer LNCS 217, 1986.

A. Example of a Curry simple interpreter

```
   module int_arith where
   import FlatCurry --metaprogramming facilities (e.g., data structure for flat progs)
   {-
   main = 3*4-1+2
   -}
   main = ieval True [Func ("rev2","main") 0 Public (TCons ("Prelude","Int") [ ]) (Rule [ ] (Comb FuncCall
("Prelude","+") [Comb FuncCall ("Prelude","-") [Comb FuncCall ("Prelude","*") [Lit (Intc 3),Lit (Intc 4)],
Lit (Intc 1)],Lit (Intc 2)]))] (Comb FuncCall ("rev2","main") [ ])
   --these functiond are only used by the partial evaluator to annotate some expressions:
   UNF x = x
   MEM x = x
   --function int is only used to test the interpreter (the partial evaluator calls
   --directly to ieval to avoid having I/O function calls
   {-
   int p e = do (Prog _ _ _ funs _ ) <- readFlatCurry p
                print (ieval True funs e)
   -}
   --ieval: this is the main function of the interpreter
   --arguments: a boolean flag, true for the top expression and false otherwise
   --           the program
   --           the expression to be evaluated
   --vars (we use no environment!)
   ieval _ _ (Var v) = Var v
   --literals
   ieval _ _ (Lit l) = Lit l

   --constructor calls (observe the use of the Boolean flag)
   ieval True v2  (Comb ConsCall v8 v9) = (Comb ConsCall v8 (ievalList True v2 v9) )
   --OJO: stops in HNF!
   --ievalList is only used to evaluate sequentially the arguments of
   --a constructor call or an arithmetic operation
   ievalList _  _ [ ] = [ ]
   ievalList top funs (e:es) = ievalListaux top funs (ieval top funs e) es
   ievalListaux top funs ne es = ne : (ievalList top funs es)

   ieval False _   (Comb ConsCall c es) = Comb ConsCall c es

   ieval top funs (Comb FuncCall (mn,f) es) =
     if mn == "Prelude"
      then if f == "failed"
        then Comb FuncCall (mn,f) es
        else if f == "=="
             then (ieval_EQ top funs (mn,f) es)
             else case f of
                      "+" -> ieval_ARITH top funs (mn,f)  es
                      "-" -> ieval_ARITH top funs (mn,f)  es
                      "*" -> ieval_ARITH top funs (mn,f)  es
      else ieval top funs (matchiRHS  funs (mn,f) es)

   ieval top funs (Case ctype e ces) =
     case (ieval False funs e) of
       Comb ConsCall f es -> ieval top funs (matchBranch ces f es)
       Lit l -> ieval top funs (matchBranchLit ces l)
   --problema con el renaming!
   ieval_ARITH :: Bool -> [FuncDecl] -> QName -> [Expr] -> Expr
   ieval_ARITH    top       fns         (mn,fn)  [e1,e2] =
     if (isLitInt e1) && (isLitInt e2)
        then (ieval_ARITH_aux (mn,fn) [e1,e2])
        else
           Comb FuncCall (mn,fn) (ievalList top fns [e1,e2])
   --evaluation of simple arithmetic functions
   ieval_ARITH_aux (_,f) [(Lit (Intc e1)),(Lit (Intc e2))] =
                case f of
                     ("*") -> Lit (Intc (e1*e2))
                     ("+") -> Lit (Intc (e1+e2))
                     ("-") -> Lit (Intc (e1-e2))
   isLitInt :: Expr -> Bool
   isLitInt e = case e of
                    (Lit (Intc _)) -> True
                    _              -> False
   ieval_EQ :: Bool -> [FuncDecl] -> QName -> [Expr] -> Expr
   ieval_EQ    top       fns         (mn,fn)  [e1,e2] =
     if (isLitInt e1) && (isLitInt e2)
        then (ieval_EQ_aux [e1,e2])
        else
           if (isLitInt e1) && (isVar e2)
              then Comb ConsCall ("Prelude","False") [ ]
              else ieval top fns (Comb FuncCall (mn,fn) (ievalList top fns [e1,e2]))
   ieval_EQ_aux [(Lit (Intc e1)),(Lit (Intc e2))] =
```

```
            case (e1==e2) of
                 True -> Comb ConsCall ("Prelude","True") [ ]
                 _    -> Comb ConsCall ("Prelude","False") [ ]
--matchBranch and matchBranchLit are used to select the matching branch
--of a case expression:
matchBranch cbranches c es =
     case cbranches of
         [ ]     -> (Comb FuncCall ("Prelude","failed") [ ])
         (Branch (Pattern p vars) e):ces ->
                if p==c then substitute vars es e
                        else matchBranch ces c es
matchBranchLit cbranches c =
     case cbranches of
         [ ]     -> (Comb FuncCall ("Prelude","failed") [ ])
         (Branch (LPattern p) e):ces ->
                if p==c then e
                        else matchBranchLit ces c
--------------------------------------------------------------------------
-- CALL UNFOLDING:
-- match a right-hand side of a given function:
matchiRHS [ ] (_,_) _ = Comb FuncCall ("Prelude","failed") [ ]
matchiRHS (Func (_,fname) _ _ _ funrule : fds) (mn,name) es =
   if fname==name then matchRHS_aux funrule es
                  else matchiRHS fds (mn,name) es
matchRHS_aux (Rule vars rhs) es = substitute vars es rhs
substitute :: [Int] -> [Expr] -> Expr -> Expr
substitute vars exps expr = substituteAll vars exps expr
-- substitute all occurrences of variables by corresponding expressions:
-- * substitute all occurrences of var_i by exp_i in expr
--   (if vars=[var_1,...,var_n] and exps=[exp_1,...,exp_n])
-- * leave all other variables unchanged (i.e., variables in case patterns)
--
substituteAll :: [Int] -> [Expr] -> Expr -> Expr
substituteAll vs es x =
    case x of
        (Var i)  -> replaceVar vs es i
        (Lit (Intc l))  -> Lit (Intc l)
        (Lit (Charc l))  -> Lit (Charc l)
        (Comb ConsCall c exps) -> Comb ConsCall c (mapsAll vs es exps)
        (Comb FuncCall c exps) -> Comb FuncCall c (mapsAll vs es exps)
--        (Comb (FuncPartCall ma) c exps) ->
--                      Comb (FuncPartCall ma) c (mapsAll vs es exps)
        (Case ctype e cases) -> Case ctype (substituteAll vs es e)
                                           (substituteAllCases vs es cases)
        (Or e1 e2) ->  (Or (substituteAll vs es e1) (substituteAll vs es e2))
        (Let [(lhs,rhs)] e) ->
             Let (mapsAllLet vs es [(lhs,rhs)]) (substituteAll vs es e)
        (Free vars e) -> Free vars (substituteAll vs es e)

replaceVar [ ] [ ] var = Var var
replaceVar (v:vs) (e:es) var  = if v==var then e
                                          else replaceVar vs es var
substituteAllCases _  _ [ ]              = [ ]
substituteAllCases vs es (tbranch:cases) =
   (substituteAllCase vs es tbranch) : (substituteAllCases vs es cases)
substituteAllCase vs es x =
             case x of
                  (Branch (Pattern (l,o) pvs) e) ->
                          Branch (Pattern (l,o) pvs) (substituteAll vs es e)
                  (Branch (LPattern l) e) ->
                          Branch (LPattern l) (substituteAll vs es e)
mapsAll vs es [ ]        = [ ]
mapsAll vs es (exp:exps) = (substituteAll vs es exp): (mapsAll vs es exps)
mapsAllLet vs es [ ] = [ ]
mapsAllLet vs es ((lhs,rhs):bindings) =
              (substituteAllLet vs es (lhs,rhs)):(mapsAllLet vs es bindings)
substituteAllLet :: [Int] -> [Expr] -> (VarIndex,Expr) -> (VarIndex,Expr)
substituteAllLet vs es (var,e)= (var,(substituteAll vs es e))

isVar :: Expr -> Bool
isVar e = case e of
               (Var _) -> True
               _       -> False
```