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

J. Guadalupe Ramos
Instituto Tecnológico de la Piedad,
Av. Tecnológica 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 programs. 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 function 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. The behavior (decreasing or increasing) of the arguments in a cycle defines the arguments in a cycle, i.e., (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:

\[
\begin{align*}
R & ::= D_1, \ldots, D_m & \text{(program)} \\
D & ::= f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
& | c(e_1) & \text{(variable)} \\
& | f(x_1, \ldots, x_n) & \text{(constructor call)} \\
& | \text{case } e \text{ of } \{ p_1 \Rightarrow e_1 \ldots, p_m \Rightarrow e_m \} & \text{(rigid case)} \\
& | \text{case } e \text{ of } \{ p_1 \Rightarrow e_1 \ldots, p_m \Rightarrow e_m \} & \text{(flexible case)} \\
p & ::= c(e_1) & \text{(pattern)}
\end{align*}
\]

Figure 1. Outline offline partial evaluator (mixpo)
data Nat = Z | S Nat
main x = prod (x) (fib (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 -> (S (S 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 \( s_n \) represents a sequence of objects \( o_1, \ldots, o_n \).
Thus, a program \( R \) consists of a sequence of function definitions \( D \) such that the left-hand side of each function definition is an expression \( e \) composed by variables \( (V) \), constructors \( (C) \), function calls \( (F) \), 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 annotated program \( \langle p_{ann} \rangle \) and performs the proper evaluation procedure to produce a final specialized program \( p_{ape} \).

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 \( f(x) = e \) in \( R \) by the new rule \( f(x) = ann^3(ann^2(e)) \), 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 the specializes procedure, which terms should be generalized (see the constructor \( BtD \) in Figure 4).

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

Figure 5. Generic procedure for NPE

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 toplevel (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:

\[
\text{data}\ \text{Prog} = \text{Prog}\ 
\begin{array}{ll}
\text{String} & \text{name of module} \\
\text{[String]} & \text{imported modules} \\
\text{[TypeDecl]} & \text{type declarations} \\
\text{[FuncDecl]} & \text{function declarations} \\
\text{[String]} & \text{operator declarations}
\end{array}
\]

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

\[
\begin{array}{ll}
data\ \text{FuncDecl} = \text{Func}\ 
\begin{array}{l}
\text{QName} \ [\text{VarIndex}] \\
\text{--qualified name}
\end{array}
\text{TypeExpr} & \text{arity} \\
\text{Visibility} & \text{public/private} \\
\text{Rule} & \text{--rule}
\end{array}
\]

data\ \text{Rule} = \text{Rule}\ 
\begin{array}{l}
\text{[VarIndex]}\ \text{Expr}
\end{array}
\]

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

\[
\begin{array}{l}
data\ \text{Expr} = \text{Var}\ [\text{VarIndex}] \\
\text{Lit}\ \text{Literal} \\
\text{Comb}\ \text{CombType}\ \text{QName}\ \text{Expr} \\
\text{Or}\ \text{Expr}\ \text{Expr} \\
\text{Case}\ \text{CaseType}\ \text{Expr}\ \text{BranchExpr}
\end{array}
\]

\[
\begin{array}{l}
data\ \text{CombType} = \text{FuncCall} \ | \ \text{ConsCall}
\end{array}
\]

\[
\begin{array}{l}
data\ \text{CaseType} = \text{Rigid} \ | \ \text{Flex}
\end{array}
\]

\[
\begin{array}{l}
data\ \text{BranchExpr} = \text{Branch}\ \text{Pattern}\ \text{Expr}
\end{array}
\]

\[
\begin{array}{l}
data\ \text{Pattern} = \text{Pattern}\ \text{QName}\ \text{[VarIndex]} \\
\text{[\text{LPattern}\ \text{LLiteral}]}
\end{array}
\]

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 \(\epsilon\) and \(\lambda\) in an expression like \([\epsilon]\) do not denote a semantic function but are only used to identify which part of an expression
Now, if you specialize the simple interpreter of the appendix, i.e., first to annotate and then specialize, you get the following program:

```haskell
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.FuncCall ("Prelude","-") [Lit (FlatCurry.Intc 2),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 `ieval` is reached. `ieval` evaluates the function/constructor expression list. See the following rules:

```haskell
ieval True v2 (ConsConsCall v8 (ievallist True v2 v9))
ieval False_ (ConsConsCall c e) = ConsConsCall c e
```

While `ieval False_ (ConsConsCall c e)` 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:

```haskell
ieval top funs (Case cype e funs) = 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 (matchRHS funs (mn,f) es)`.

You may verify this in the following rule:

```haskell
ieval top funs (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 (matchRHS 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.

`matchRHS` 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:

\[
\text{matchiRHS} [ \{ \ldots \} _n = \text{Comb FuncCall} (\text{"Prelude"}, \text{"failed"}) ] _n
\]

\[
\text{matchiRHS} (\text{Func} (\ldots), \text{name}) _n = \text{funrule : fds} (\text{sm}, \text{name}) _n
\]

if \text{fname} == \text{name} then matchiRHS_aux funrule es

else matchiRHS fds (\text{sm}, \text{name}) _n

matchiRHS_aux (Rule \text{vars} rhs) es = \text{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 \texttt{mixpo}
unfolds. We know our interpreters unfold the same way that \texttt{mixpo}.
So consider the annotated program of Figure 4. To start the special-
ization of any annotated program, \texttt{mixpo}, looks for the function main
by default, in this case \texttt{mixpo} builds the following call:

\[
[\text{Comb FuncCall} (\text{"fib\_ann"}, \text{"main"}) [\text{[Var 0]]}]\]

that is in FlatCurry, which corresponds to (\texttt{main v0}) in Curry
source. We have put into (\texttt{mixpo}) to know the calls to
unfold, the first three are:

unfoldpo: (\texttt{main v0})

unfldpo: (\texttt{\text{H\_prod v0 \text{UNF (fib (S (S (S (S (S (Z)))))})}})

unfldpo: (\texttt{H\_prod v0 \text{Case} (S (S (S (S (S (Z))))) of (Z -> (Z))})

\[
(S \text{v105} -> \text{Case} \text{v105 of} (Z -> (S (Z))) (S \text{v106} -> \text{UNF (simp (UNF (fib (S \text{v106})) (UNF (fib \text{v106})))))})\]

\]

Given the call (\texttt{main v0}), \texttt{matchiRHS}, first looks for this func-
tion to unfold. You can see that the right hand side of (\texttt{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 (\texttt{(fib (S (S (S (S (S (Z)))))))}) expression unfolded. Note
that the latter call to unfold there is a \texttt{Case} expression which cor-
responds to the right hand side of the function \texttt{fib}, in flat format,
remember that annotation and specialization is done in this for-
mat. But what about substitutions? The first substitution happens
when \texttt{matchiRHS} finds the structure of the function \texttt{main} then
the variables on the right hand side indicated at the left of \texttt{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
(\texttt{main v0}) in flat format. Thus \texttt{v1} is replaced by \texttt{v0} and is shown
in the result of the first unfold. The second substitution happens when
\texttt{mixpo} requires the second unfold:

\[
\text{H\_prod v0 \text{UNF (fib (S (S (S (S (S (Z)))))})})\]

first \texttt{matchiRHS} tries to unfold the function \texttt{fib}, so finds the struc-
ture of this function, some code like:

\[
\text{fib} \text{v1} = \text{case} \text{v1} \text{of} (Z \rightarrow (Z)) (S \text{v2} \rightarrow \text{case} \text{v2} \text{of} (Z \rightarrow (S (Z))) (S \text{v3} \rightarrow \text{UNF (simp (UNF (fib (S v3))) (UNF (fib v3)))))})\]

then the variables on the right hand side indicated at the left of
\texttt{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 (S (Z)))))
thus \texttt{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 inter-
preters 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 \texttt{pcty} that accepts a function call with static
dynamic data arguments included into the same program, and a
program \texttt{intcty} written in Curry language to interpret programs
in the intermediate FlatCurry language that produces target programs
\texttt{tgcty} in FlatCurry also. Thus we may represent a particular inter-
pretation process as:

\[
\text{tgcty} = [\text{intcty}] [\text{pcty}]\]

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

\[
\text{tgcty} = [\text{mixpo}] [\text{intcty}] [\text{pcty}]\]

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 \texttt{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:

\[
\{-\}

\text{main} = (3 = 3) \&\& (1 = 1)

\]

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

(Comb FuncCall ("Prelude","&&") [Lit (Intc 1),Lit (Intc 1)])]) (Comb FuncCall ("andexam","main") []) \}

If we require to PKACS 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 (\texttt{mn}, f) es)

if \texttt{mn} == “Prelude”

then if \texttt{f} == “failed”

then Comb FuncCall (\texttt{mn}, f) es

else if \texttt{f} == “==”

then (ieval_EQ top funs (\texttt{mn}, f) es)

else case \texttt{f} of

“&&” -> ieval_ARITH top funs (\texttt{mn}, f) es

“*” -> ieval_ARITH top funs (\texttt{mn}, f) es

“+” -> ieval_ARITH top funs (\texttt{mn}, f) es

“&&” -> ieval_SAND top funs (\texttt{mn}, f) es

else

ieval top funs (matchiRHS funs (\texttt{mn}, f) es)

Notice that we added the call \texttt{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 funs (\texttt{mn}, fn) [es1,es2] =

case (ieval True funs es1) of

(Comb ConsCall (Pre,Tru) []) -> (ieval top funs es2)

(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:
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 v0 = ieval_pe1 (Comb FuncCall ("fib7","main") [v0])
```

You can see the x of function main this variable represents a dynamic value of the interpreter, i.e. a value probably known at paper for GPCE'11 to these functions are annotated with UNF of SCA, i.e., do not have idempotent multigraphs, so the calls to these functions identified in the right hand side; the branches to unfold: main, fibb, 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.

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.

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 specialization for C programs [33] able to specialize both in bytecode interpreters as structured language and yields excellent speedups. In [25], Leuschel el. 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 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 specialization are written in Curry. The interpreters accept 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 implement a HO termination analysis. We have participated in a recent work to make a transformation to polivariant BTA of

---

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


A. Example of a Curry simple interpreter

```haskell
module int_arith where
import FlatCurry -- metaprogramming facilities (e.g., data structure for flat progs)
{-
main = 3*4-1+2
-}
main = ieval True [] 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 function is 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!)
i eval _ _ (Var v) = Var v
--literals
ieval _ _ (Lit l) = Lit l
--constructor calls (observe the use of the Boolean flag)
i eval True v2 (Comb ConsCall v8 v9) = (Comb ConsCall v8 (ievalList True v2 v9))
--QJO: stops in HNF!
--ievalList is only used to evaluate sequentially the arguments of
--a constructor call or an arithmetic operation
ievalList _ _ [] = []
ievalList v (es:v) = ievalList aux v (ieval True funs es) es
ievalListaux top funs ne es = ne : (ievalList top funs es)
i eval False _ (Comb ConsCall c es) = (Comb ConsCall c es)
i eval 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 (matchRHS funs (mn,f) es)
i eval top funs (Case ctype e ces) =
  if (isLitInt e) && (isLitInt e)
  then Comb ConsCall f es -> ieval top funs (matchBranch ces f es)
  Lit l -> ieval top funs (matchBranchLit ces l)
--problems con el renaming!
i eval ARITH :: Bool -> [FuncDecl] -> QName -> Expr -> Expr
ieval_ARITH top funs (mn,fn) [e1,e2] =
  if (isLitInt e1) && (isLitInt e2)
  then (ieval_ARITH_aux (mn,fn) [e1,e2])
  else Comb FuncCall (mn,fn) (ievalList top funs [e1,e2])
i eval_ARITH_aux (_,f) [(Lit (Intc e1)),(Lit (Intc e2))] =
  case f of
    (“*”) -> Lit (Intc (e1*e2))
    (“+”) -> Lit (Intc (e1+e2))
    (“-”) -> Lit (Intc (e1-e2))
isclosed :: Expr -> Bool
isclosed e = case e of
  (Lit (Intc _)) -> True
  _ -> False
ieval_EQ :: Bool -> [FuncDecl] -> QName -> Expr -> Expr
ieval_EQ top funs (mn,fn) [e1,e2] =
  if (isLitInt e1) && (isclosed e2)
  then (ieval_EQ_aux [e1,e2])
  else Comb ConsCall (“Prelude”,“False”) []
i eval top funs (Comb FuncCall (mn,fn) (ievalList top funs [e1,e2]))
i eval_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
--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:
matchRHS [ ] (_,_ ,_ = Comb FuncCall ("Prelude","failed") [ ]
matchRHS (Func (_,fname) _ _ funrule : fds) (sm,name) es =
    if fname==name then matchRHS_aux funrule es
    else matchRHS fds (sm,name) es
matchRHS_aux (Rule var 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 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:expes) = (substituteAll vs es exp) : (mapsAll vs es expes)
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