An Offline Partial Evaluator for Curry Programs *

J. Guadalupe Ramos †
I.T. La piedad
Av. Tecnológico 2000
La Piedad, Mich., México
guadalupe@dsic.upv.es

Josep Silva Germán Vidal
DSIC, T.U. Valencia
Camino de Vera S/N
E-46022 Valencia, Spain
{jsilva,gvidal}@dsic.upv.es

Abstract

Narrowing-driven partial evaluation is a powerful technique for the specialization of functional logic programs. In this paper, we describe the implementation of a narrowing-driven partial evaluator for Curry programs which follows the offline approach to ensuring termination. Although the new partial evaluator is less precise than previous (online) partial evaluators for Curry, it is much faster and, thus, allows the specialization of larger programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—partial evaluation

General Terms algorithms, performance

Keywords narrowing, offline partial evaluation

1. Introduction

A partial evaluator is a source-to-source program transformer which takes a program and part of its input data—the so-called *static* data—and returns a residual, specialized program which is (hopefully) faster than the original program since those computations that depend only on the static data have been performed once and for all at partial evaluation time.

Narrowing-driven partial evaluation (NPE) is a powerful specialization technique for functional logic programs [3]. Although NPE can be seen as a traditional partial evaluation scheme for program specialization, it can also achieve more powerful optimizations like deforestation, elimination of higher-order functions (represented in a first-order setting by defunctionalization), etc. A narrowing-driven partial evaluator [1] is currently integrated into the PAKCS environment [8] for the declarative multi-paradigm language Curry [7].

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-00231 and HU 2003-0003, by the *Generalitat Valenciana* GRUPOS03/025, and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

† This research was done while visiting the Tech. University of Valencia (Spain), funded by grants SEIT-ANUIES 20020V and DGEST 'beca-comisión" (México).

WCFLP'05 September 29, 2005, Tallinn, Estonia.

Although NPE gives good results on small programs, it does not scale up well to realistic problems (e.g., interpreter specialization). In order to overcome this problem, an *offline* NPE scheme has recently been introduced [10].

Partial evaluators fall in two main categories, *online* and *offline*, according to the time when termination issues are addressed. Online partial evaluators are usually more expensive—i.e., they have higher run times—but also more precise—i.e., they produce faster residual programs—since they have more information available at partial evaluation time. On the other hand, offline partial evaluators usually proceed in two stages: the first stage returns a program which includes annotations to guide the partial computations (e.g., to identify those function calls that can be safely unfolded); then, the second stage—the proper partial evaluation—only needs to obey the annotations and, thus, it is generally much faster—but less precise, since termination is only statically analyzed—than online partial evaluators.

In this work, we describe the implementation of an offline narrowing-driven partial evaluator that follows the scheme proposed in [10].

2. Meta-Programming in Curry

The implementation of our partial evaluator 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
```

Consider, e.g., the following Curry program "`reverse.curry`" defining function reverse with an accumulating parameter:

```
rev xs = rr xs []

rr []     ys = ys
rr (x:xs) ys = rr xs (x:ys)
```

Here, function "`rr`" is represented in FlatCurry by means of the following data structure:

```
Func ("reverse","rr") 2 Public (FuncType ...)
(Rule [0,1]
  (Case Flex (Var 0)
    [Branch (Pattern ("prelude","[]") [])
            (Var 1),
     Branch (Pattern ("prelude",":") [2,3])
            (Comb FuncCall ("reverse","rr")
               [Var 3,
                Comb ConsCall ("prelude",":")
                     [Var 2, Var 1]
               ])]))
```

Source Curry programs can be read in and translated into FlatCurry by using the function `readFlatCurry`. Then, one can manipulate the data structure representing a FlatCurry program and, finally, write it down to a FlatCurry file by using the function `writeFCY`.

3. Program Annotation

In general, the narrowing space of a term may be infinite. However, the termination of partial evaluation can be ensured whenever the partial computations are *quasi-terminating* [6] (i.e., only finitely many different function calls—modulo variable renaming—are processed). In [10], we introduced a syntactic characterization for *inductively sequential* programs (roughly, source Curry programs), called *nonincreasing*, which guarantees the quasi-termination of narrowing computations. Intuitively speaking, nonincreasing programs fulfill the following conditions:

- the right-hand side of each rule in a function definition is linear (i.e., every variable occurs at most once) and
- if a function belongs to a (potential) cycle, then no nested function calls are allowed and, moreover, it should *consume* its parameters or leave them unchanged (i.e., the depth of the variables in the parameters must not be increased).

Since this characterization is quite restrictive, [10] introduced an automatic algorithm to annotate arbitrary source programs so that quasi-termination can still be ensured by using an extension of narrowing that generalizes annotated subterms and, consequently, ensures the termination of the partial evaluation process. This algorithm is very fast since it only performs a single pass[1] through the FlatCurry data structure annotating those expressions that violate the nonincreasing characterization.

E.g., given the program "`gauss.curry`", which computes de classical Gauss function for numbers expressed as Peano natural numbers:

```
data Nat = Z | S Nat

g Z     = Z
g (S n) = + (S n) (g n)

+ Z     y = y
+ (S x) y = S (+ x y)
```

the annotation algorithm of [10] returns the following annotated program:

```
data Nat = Z | S Nat

g Z     = Z
g (S n) = + (S n) (gen (g n))

+ Z     y = y
+ (S x) y = S (+ x y)
```

Here, the auxiliary function "`gen`", defined as the identity function, is only used to annotate those subterms that should be generalized at partial evaluation time.

Adapting the annotation algorithm of [10] for inductively sequential systems to the FlatCurry language is not difficult. Basically, rather than inspecting the right-hand sides of the rules, one should inspect the "branches" of each function definition. For example, for function "g" above, the implemented system returns the following FlatCurry expression:

```
Func ("gauss","g") 1 Public (FuncType ...)
 (Rule [0]
 (Case Flex (Var 0)
  [Branch (Pattern ("gauss","Z") [])
     (Comb ConsCall ("gauss","Z") []),
   Branch (Pattern ("gauss","S") [1])
     (Comb FuncCall  ("gauss","+")
       [(Comb ConsCall ("gauss","S") [(Var 1)]),
        (Comb FuncCall ("libope","gen")
          [(Comb FuncCall  ("gauss","g") [(Var 1)]
                                          )])])]))
```

where "`gen`" is defined in a partial evaluation library which is imported in the specialized version of the program[2].

4. the OFFLINE PARTIAL EVALUATOR

The offline narrowing-driven partial evaluator proceeds in three sequential phases: specialization, extraction of new rules, and post-unfolding.

Specialization. This phase is based on the *generalizing* extension of narrowing explained in section 3. Basically, It performs a meta-interpreter for FlatCurry expressions, extended as follows:

Generalization: if a FlatCurry expression contains annotations (i.e., occurrences of function "`gen`"), the expression is *flattened*

[1] Assuming that we already know which functions are cyclic (by using, e.g., a simple graph of functional dependencies).

[2] It is intended that "`gen`" could be part of the standard prelude of Curry as it happens with the "`peval`" function of the current online partial evaluator.

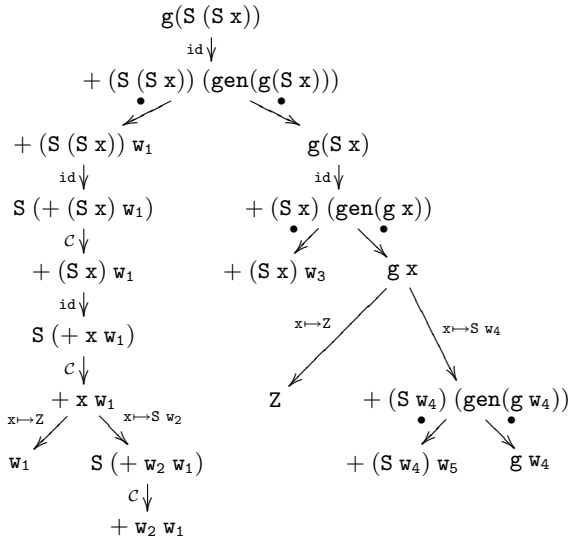
into several expressions by replacing annotated subexpressions with fresh variables.

Narrowing: if the FlatCurry expression is a function call and does not contain annotations, a standard narrowing step is performed.

Decomposition: if the FlatCurry expression is a constructor call, it is reduced to its arguments (if any).

These three operations are successively applied to the initial and following expressions until no more new expressions (modulo variable renaming) are generated. As mention before, the termination of this process is ensured thanks to the annotation stage that guarantees the nonincreasing property [10]. The result of this process can be depicted as a tree (that we call the *search tree* of generalizing needed narrowing) which contains all the reduced expressions that were produced by the three operations (generalization, narrowing and decomposition). This tree is posteriorly used for the generation of the residual rules.

For example, generalizing narrowing computes the following search tree for the initial call "`g (S (S x))`" w.r.t. the previously annotated program:



where w_1 to w_5 are fresh variables. Here, generalization steps are denoted by $\rightarrow_\bullet$, narrowing steps by $\rightarrow_\sigma$, where σ is the computed substitution (`id` is the identity substitution), and decomposition steps by $\rightarrow_c$.

Extraction of new rules. This second stage computes the generation of residual rules (the so-called *resultants*) from the search tree. For this purpose, we implemented a function that traverses the search space of generalizing narrowing and extracts a residual rule (without annotations) from each proper narrowing step. For instance, the residual program associated to the search space shown above is as follows:

```
g (S (S x))     = + (S (S x)) (g (S x))
g (S x)         = + (S x) (gx)
g x             = FCase x of
                        (Z    -> Z)
                        (S w4 -> + (S w4) (g w4))

+ (S (S x)) w1 = S (+ (S x) w1)
+ (S x)     w1 = S (+  x w1)
+  x        w1 = FCase x of
                        (Z    -> w1)
                        (S w2 -> S (+ w2 w1))
```

Furthermore, the rules of residual programs usually require a renaming post-process (see [4]). In the example, we get:

```
g_2 x    = +_2 x (g_1 x)
g_1 x    = +_1 x (g_0 x)
g_0 x    = FCase x of
                (Z    -> Z)
                (S w4 -> +_1 w4 (g_0 w4))

+_2 x w1 = S (+_1 x w1)
+_1 x w1 = S (+_0 x w1)
+_0 x w1 = FCase  x of
                 (Z    -> w1)
                 (S w2 -> S (+_0 w2 w1))
```

Post-unfolding. Finally, the third phase performs a simple post-unfolding *transition compression* [9] to eliminate intermediate functions. This stage was already included in the online partial evaluator and required no extension. In our example, the post-unfolding phase returns the following specialized functions:

```
g_2 x    = S (S (+ x (g_1 x)))
g_1 x    = S (+ x (g_0 x))
g_0 x    = FCase x of
                (Z    -> Z)
                (S w4 -> S (+ w4 (g_0 w4)))

+ x w1 = FCase  x of
               (Z    -> w1)
               (S w2 -> S (+ w2 w1))
```

5. Implementation

The source code of the system has been organized in nine independent modules that we describe here:

GNNTrees.curry Contains the definition of a data structure for generalizing needed narrowing (GNN) trees and their associated trees manipulation functions.

```
data GNNTree =  Leaf Expr
             | LeafM  Expr Expr
             | TreeN  Expr [GNNTree]
             | TreeNM Expr [GNNTree]
             | TreeD  Expr [GNNTree]
             | TreeG  Expr [GNNTree]
```

This data structure embodies the GNN tree represented by one constructor definition for each kind of node: `Leaf` for expressions in a terminal derivation (e.g., purely constructor calls), `LeafM` for expressions whose computation has been stopped because they have already been processed before. `TreeN` for a typical needed narrowing step (e.g., a function call). `TreeNM` for expressions that are computed by first time (they manage the memoization marks to ensure quasi-termination). `TreeD` for constructor-rooted expressions labeled for decomposition. And, `TreeG` for expressions labeled for generalization.

For instance, the narrowing tree of section 4, contains these `TreeNM` (memoized) nodes:

```
(TreeNM (+ (S x) w_1) ... )
(TreeNM (+ x     w_1) ... )
(TreeNM (g x)         ... )
```

We also find expressions in the tree that have not been processed because they are equal modulo variable renaming to a previous node (i.e., `LeafM` nodes):

```
(LeafM (+  w_2     w_1) ... )
```

```
(LeafM (+  (S x)   w_3) ... )
(LeafM (+  (S w_4) w_5) ... )
(LeafM (g  w_4)         ... )
```

In addition, the following nodes are labeled for generalization (i.e., `LeafM` nodes):

```
(TreeG (+ (S (S x)) (gen (g (S x)))) ... )
(TreeG (+ (S x)     (gen(g x)))      ... )
(TreeG (+ (S w_4)   (gen (gw_4)))    ... )
```

OffPeval.curry implements the algorithm used in the specialization phase of the offline narrowing-driven partial evaluator. As mention in section 4, it constructs a generalizing needed narrowing tree as follows:

```
gnn  m e
 case e of
   isCons       -> (Leaf e, m)
   isConsRooted -> gnnD m (decompose e)
   hasGEN and isOpRooted -> gnnG m (generalize e)
   existsRen    -> (LeafM e,m)
   otherwise    -> gnnRLNT m r
```

Roughly speaking, the generalizing needed narrowing algorithm (`gnn`) receives two arguments: `m` is the list of already processed expressions; and `e` is the expression to be analyzed.

`gnn` proceeds by inspecting the expression `e`. If `e` is only formed by constructor calls, then `gnn` returns a `Leaf` node, thus ending the derivation of this branch. If `e` is a constructor-rooted expression, then a decomposition phase is applied by the `gnnD` function. If the expression `e` contains annotations for generalization and it is operation-rooted, then a generalization step is performed by the `gnnG` function. If the expression belongs to `m` then the computation is stopped and a new `LeafM` is returned. Otherwise a needed narrowing step [2] is applied by the `gnnRLNT` function.

Since `gnn` is recursively called by functions `gnnD`, `gnnG` and `gnnRLNT`, the process continues until no more new expressions are produced. Thus, a finite `GNNTree` data structure is constructed in a single pass through the original FlatCurry expression.

Let us note that, contrarily to the online partial evaluator [3, 1], the concepts of *global control*—to stop the partial evaluation process—and *local control*—to stop a derivation step—do not apply to the offline approach. In the offline partial evaluator, control termination issues are overcame by a convenient manipulation (e.g. generalization) of the annotated expressions.

Generalization.curry implements required functions for generalization: searching of marks in expressions, flattening expressions with `gen` marks, removing annotations, etc.

RLNT.curry is an implementation of the RLNT [2] calculus for flatCurry programs. Its core is the function `rlnt e`, that inputs an expression `e` to be evaluated and outputs a meta-interpreted expression `e'`. `e` can present the following cases:

`(Comb FuncCall f es)` The function call is unfolded, and `rlnt` returns the right hand side of the corresponding rule with a variable renaming.

`(Case ctype (Var n) ces)` The *Case* expression is returned with the variable bindings propagated to all the branches.

`(Case ctype (Lit l) ces)` The expression associated to the corresponding pattern is returned.

`(Case ctype (Comb ConsCall fname es) ces)` It is the classical *case select* expression. The computed answer is the corresponding expression in the branch that matches the argument of the case.

`(Case ctype (Comb FuncCall fname es) ces)` The function call is unfolded, and the evaluation continues with the case expression.

`(Case ctype (Case ctype2 (Var n) ces2) ces)` This case corresponds to the "*case of case*" expression of Wadler's deforestation [11].

`(Case ctype ( Case ctype2 ce2 ces2) ces)` The case expression is returned with a recursive call to `rlnt`. `e'` is: `Case ctype (rlnt (Case ctype2 ce2 ces2))`

Let us note that the `rlnt` function applies the specialization action corresponding to each kind of expression, but it does not consider any termination issue.

Resultants.curry performs the extraction of the final resultants `(Expr,Expr)`, by visiting each node of the GNN tree and constructing rules depending on the kind of the node.

PostUnfolding.curry is in charge of the post-unfolding simplification phase.

OffPretty.curry includes functions for the pretty-printing of expressions, trees and rules.

Prog2Flat.curry builds a FlatCurry program from a list of resultants `(Expr,Expr)`.

Util.curry contains a collection of useful utilities.

6. The offline partial evaluator in practice

The offline partial evaluator can be executed by using the Curry compiler PAKCS [8]. First, the main module must be loaded:

```
prelude> :l OffPeval
```

and then, the partial evaluation process can be started with a call to function `mix` with the (annotated) program to be partially evaluated:

```
OffPeval> mix "examples/power"
```

The specializer assumes that the first function in the program (usually `main`) is the initial call for specialization. The original program should include `gen` annotations in order to ensure the termination of the process.

There are several intermediate steps that can be displayed if the user is interested in:

1. the GNN tree constructed.
2. the GNN tree including memoized nodes, i.e., nodes participating in quasi-termination.
3. the set of resultants, before the post-processing stage.
4. the computed renaming of the left-hand sides of rules.
5. the renamed resultants (the completely renamed rules).
6. the program obtained after the post-unfolding simplification.

These additional options can be managed with a set of `Boolean` flags in the `OffPeval` module. By default, all values are `False` (i.e., no additional information is displayed):

```
gnnFlag     = False
gnnMemoFlag = False
resulFlag   = False
renFlag     = False
renresFlag  = False
postFlag    = False
```

Also, there is a flag that determines whether the specialized code is added to the original program (`True`, e.g., for preserving functions while benchmarking) or replaces it (`False`, e.g., for measuring code size reduction). By default this flag is True:

```
mixFlag = True
```

The residual program is finally stored in a FlatCurry file (`.fcy`) with the suffix "`_pe`" (i.e., "`OriginalName_pe.fcy`").

7. Online vs. Offline

We have conducted a number of experiments to compare the new offline partial evaluator with the previous online system [1]. The sources of the partial evaluator and a detailed explanation of the benchmarks considered below are publicly available from

`http://www.dsic.upv.es/users/elp/german/offpeval/`

For each benchmark, we computed the time for executing the previous *online* NPE tool (`onlineNPE`), the time for executing the new *offline* NPE tool described so far (`offlineNPE`), including both the time for annotating the original program (`ann`) and for performing partial computations and extracting the residual program (`mix`), as well as the speedups achieved by the programs specialized with each technique (`speedup1` and `speedup2`); speedups are given by *orig*/*spec*, where *orig* and *spec* are the absolute run times of the original and specialized programs, respectively.

The average results are as follows (more details about the experimental evaluation can be found in [10]):

`onlineNPE:`	9642 ms.	`speedup1:`	1.706
`offlineNPE (ann):`	119 ms.	`speedup2:`	1.630
`offlineNPE (mix):`	1726 ms.		

The experiments point out that the offline scheme reduced significantly the partial evaluation times (namely the average, including both annotation and proper partial evaluation, is less than 20% of the average for the original NPE tool), which means that our main goal has been achieved.

As for the speedups, we note that most of the benchmarks were *specialization* problems (rather than *optimization* problems), which explains the good results achieved by our offline NPE tool. However, the new method is not able to pass the so-called "KMP-test". To overcome this drawback, the information of a traditional binding-time analysis [5] (i.e., specifying which componentes of the initial call are dynamic or static) should be integrated into the partial evaluator (currently, only termination is taken into account with no assumption on the static data).

References

[1] 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.

[2] 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.

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

[4] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *Theory and Practice of Logic Programming*, 5(3):273–303, 2005.

[5] N. H. Christensen, R. Glück, and S. Laursen. Binding-time analysis in partial evaluation: One size does not fit all. *Lecture Notes in Computer Science*, 1755:80+, 2000.

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

[7] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.

[8] 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, C.A.U. Kiel, Germany, 2004.

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

[10] J. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Programs. In *Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP 2005)*, 2005. To appear.

[11] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.