A Transformational Approach to Polyvariant BTA of Higher-Order Functional Programs*

Gustavo Arroyo[1], J.Guadalupe Ramos[2], Salvador Tamarit[1], and Germán Vidal[1]

[1] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
{garroyo,stamarit,gvidal}@dsic.upv.es
[2] Instituto Tecnológico de la Piedad, La Piedad, Michoacan, México
guadalupe@dsic.upv.es

Abstract. We introduce a *transformational* approach to improve the first stage of offline partial evaluation of functional programs, the so called binding-time analysis (BTA). For this purpose, we first introduce an improved defunctionalization algorithm that transforms higher-order functions into first-order ones, so that existing techniques for termination analysis and propagation of binding-times of first-order programs can be applied. Then, we define another transformation (tailored to defunctionalized programs) that allows us to get the accuracy of a polyvariant BTA from a monovariant BTA over the transformed program. Finally, we show a summary of experimental results that demonstrate the usefulness of our approach.

1 Introduction

Partial evaluation [14] aims at specializing programs w.r.t. part of their input data (the *static* data). Partial evaluation may proceed either online or offline. *Online* techniques implement a single, monolithic procedure that specializes the program while dynamically checking that the termination of the process is kept. *Offline* techniques, on the other hand, have two clearly separated phases. The aim of the first phase, the so called *binding-time analysis* (BTA), is basically the propagation of the static information provided by the user. A BTA should also ensure that the specialization process terminates; for this purpose, it often includes a termination analysis of the program. The output of this phase is an *annotated* program so that the second phase—the proper specialization—only needs to follow these annotations (and, thus, it runs faster than in the online approach).

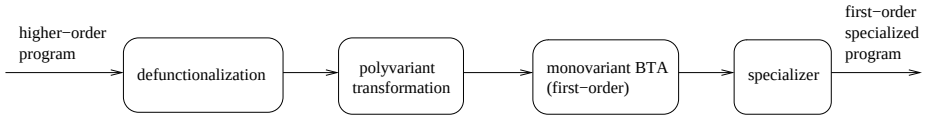
Narrowing-driven partial evaluation [2] is a powerful technique for the specialization of functional (logic) programs based on the *narrowing* principle [21], a conservative extension of rewriting to deal with logic variables (i.e., *unknown*

* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008, by SES-ANUIES and by DGEST (México).

information in our context). An offline approach to narrowing-driven partial evaluation has been introduced in [17]. In order to improve its accuracy, [6] adapts a *size-change analysis* [15] to the setting of narrowing. This analysis 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).

Unfortunately, the size-change analysis of [6] and the associated BTA suffer from several limitations. Firstly, the size-change analysis is only defined for *first-order* functional programs, thus limiting its applicability. And, secondly, the associated binding-time analysis is *monovariant*, i.e., a single sequence of binding-times[3] is associated to the arguments of a given function and, thus, all calls to the same function are treated in the same way, which implies a considerable loss of accuracy.

In this work, we present a transformational approach to overcome the above drawbacks. Basically, we first transform the original higher-order program by *defunctionalization* [18]. In particular, we introduce an extension of previous defunctionalization techniques (like [4, 12]) that is specially tailored to improve the accuracy of the size-change analysis. Then, we introduce a source-to-source transformation that aims at improving the accuracy of both the size-change analysis and the associated BTA by making explicit the binding-times of every argument of a function. In this way, every function call with different binding-times can be treated differently. Thanks to this transformation, one can get the same accuracy by using a monovariant BTA over the transformed program as by using a *polyvariant* BTA (where several binding-times can be associated to a given function) over the original program.



The paper is organized as follows. After some preliminaries, Sect. 3 introduces our defunctionalization technique in a stepwise manner. Then, Sect. 4 presents a polyvariant transformation that makes explicit the binding-times of functions. Section 5 shows a summary of the experimental results conducted to evaluate the usefulness of the approach. Finally, Sect. 6 discusses some related work and concludes.

2 Preliminaries

Term rewriting [7] offers an appropriate framework to model the first-order component of many functional (logic) programming languages. Therefore, in the remainder of this paper we follow the standard framework of term rewriting for developing our results.

[3] We consider the usual binding-times: static (definitely known at partial evaluation time) and dynamic (possibly unknown at partial evaluation time).

A *term rewriting system* (TRS for short) is a set of rewrite rules $l = r$ such that l is a nonvariable term and r is a term whose variables appear in l; terms l and r are called the left-hand side and the right-hand side of the rule, respectively. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. We restrict ourselves to finite signatures and TRSs. We denote the domain of terms and *constructor terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively, where $\mathcal{V}$ is a set of variables with $\mathcal{F} \cap \mathcal{V} = \emptyset$.

A TRS $\mathcal{R}$ is *constructor-based* if the left-hand sides of its rules have the form $f(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$. The set of variables appearing in a term t is denoted by $Var(t)$. A term t is *linear* if every variable of $\mathcal{V}$ occurs at most once in t. $\mathcal{R}$ is left-linear if l is linear for all rules $l = r \in \mathcal{R}$. The *definition* of f in $\mathcal{R}$ is the set of rules in $\mathcal{R}$ whose root symbol in the left-hand side is f.

The root symbol of a term t is denoted by $root(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). As it is common practice, a *position* p in a term t is represented by a sequence of natural numbers, where ϵ denotes the root position. Positions are used to address the nodes of a term viewed as a tree: $t|_p$ denotes the *subterm* of t at position p and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. A term t is *ground* if $Var(t) = \emptyset$. A term t is a *variant* of term t' if they are equal modulo variable renaming. A *substitution* σ is a mapping from variables to terms such that its domain $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is finite. The identity substitution is denoted by id. Term t' is an *instance* of term t if there is a substitution σ with $t' = \sigma(t)$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. In the following, we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

In the remainder of this paper, we consider *inductively sequential* TRSs [3] as *programs*, a subclass of left-linear constructor-based TRSs. Essentially, a TRS is inductively sequential when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction, i.e., typical functional programs.

3 Defunctionalization

In this section, we introduce a stepwise transformation that takes a higher-order program and returns a first-order program (a term rewrite system). In contrast to standard defunctionalization algorithms, we perform a restricted form of variable instantiation and unfolding (see step 2 below) so that more higher-order information is made explicit. Although it may increase code size, the next steps of the BTA—size-change analysis and propagation of binding-times—can exploit it for producing more accurate results; hopefully, this code size increase is then removed in the specialization phase (see the benchmarks in Sect. 5).

We present our improved defunctionalization technique in a stepwise manner:

1. The first step makes both applications and partial function calls explicit.

2. Then, the second step instantiates functional variables with all possible partial function calls.
3. Finally, if after reducing applications[4] as much as possible the program still contains some applications, the third step adds an appropriate definition for the application function.

3.1 Making Applications and Partial Calls Explicit

First, we make every application of the higher-order program explicit by using the fresh function apply. Also, we distinguish partial applications from total functions. In particular, partial applications are represented by means of the fresh constructor symbol partcall. Total calls are denoted in the usual way, e.g., $f(\overline{t_n})$ for some defined function symbol f/n. A partial call is denoted by $\mathsf{partcall}(f, k, \overline{t_m})$ where f/n is a defined function symbol, $0 < k \leq n$, and $m + k = n$, i.e., $\overline{t_m}$ are the first m arguments of f/n but there are still k missing arguments (if $k = n$, then the partial application has no arguments, i.e., we have $\mathsf{partcall}(f, n)$).[5] For simplicity, in the following we consider that partcall is a variadic function; nevertheless, one can formalize it using a function with three arguments so that the third argument is a (possibly empty) list with the already applied arguments.

Once all applications are made explicit with apply and partcall, we apply the following transformation to the right-hand sides as much as possible:

$$\mathsf{apply}(\mathsf{partcall}(f, k, \overline{t_m}), t_{m+1}) = \begin{cases} f(\overline{t_{m+1}}) & \text{if } k = 1 \\ \mathsf{partcall}(f, k-1, \overline{t_{m+1}}) & \text{if } k > 1 \end{cases} \qquad (*)$$

This is useful to avoid unnecessary applications in the defunctionalized program when enough information is available to reduce them statically.

In the following, we assume that every program contains an entry function, called main, which is defined by a single rule of the form $(\mathsf{main}\ x_1\ \ldots\ x_n\ =\ r)$, with $x_1, \ldots, x_n \in \mathcal{V}$ different variables, and that the program contains no call to this distinguished function. Furthermore, we consider that run time calls to main have only constructor terms (i.e., values) as arguments; this is required to avoid the introduction of higher-order expressions which are not in the program.

Example 1. Consider the following higher-order program $\mathcal{R}_1$ (as it is common practice, we use a curried notation for higher-order programs):

$$\begin{array}{ll} \mathsf{main}\ xs = map\ inc\ xs & map\ f\ [\,] \quad\ = [\,] \\ inc\ x \quad = Succ\ x & map\ f\ (x : xs) = f\ x : map\ f\ xs \end{array}$$

[4] Basically, every expression of the form $\mathsf{apply}(\mathsf{partcall}(\ldots), \ldots)$ can be reduced, where apply is the application function and partcall denotes a partial function call.

[5] A similar representation is used in FlatCurry, the intermediate representation of the functional logic programming language Curry [11].

where natural numbers are built from Z and $Succ$ and lists are built from [] and ":". First, we make all applications and partial calls explicit:

$$\begin{array}{ll} \mathsf{main}(xs) & = \mathsf{apply}(\mathsf{apply}(\mathsf{partcall}(map, 2), \mathsf{partcall}(inc, 1)), xs) \\ map(f, [\,]) & = [\,] \\ map(f, x : xs) & = \mathsf{apply}(f, x) : \mathsf{apply}(\mathsf{apply}(\mathsf{partcall}(map, 2), f), xs) \\ inc(x) & = Succ(x) \end{array}$$

Then, we reduce all calls to apply with a partial call as a first argument using the transformation $(*)$ above so that we get the transformed program $\mathcal{R}_2$:

$$\begin{array}{ll} \mathsf{main}(xs) = map(\mathsf{partcall}(inc, 1), xs) & map(f, [\,]) = [\,] \\ inc(x) = Succ(x) & map(f, x : xs) = \mathsf{apply}(f, x) : map(f, xs) \end{array}$$

3.2 Instantiation of Functional Variables.

In the next step, we instantiate the functional variables of the program so that some applications can hopefully be reduced. This is the main difference w.r.t. previous defunctionalization algorithms. In the following, we say that a variable is a *functional* variable if it can be bound (at run time) to a partial call. Now, we replace every functional variable by all possible partial applications.

Let $\textsc{pcalls}_{\mathcal{R}}$ be the set of function symbols that appear in the partcall's of $\mathcal{R}$, i.e.,

$$\textsc{pcalls}_{\mathcal{R}} = \{f/n \mid \mathsf{partcall}(f, k, t_1, \ldots, t_m) \text{ occurs in } \mathcal{R}, \text{ with } k + m = n\}$$

Then, for each function $f/n \in \textsc{pcalls}_{\mathcal{R}}$ with type[6]

$$\tau_1 \mapsto \ldots \mapsto \tau_n \mapsto \ldots \mapsto \tau_k \mapsto \tau_{k+1} \qquad (n \leq k)$$

we replace each rule $l[x]_p = r$ where x is a functional variable of type

$$\tau'_j \mapsto \ldots \mapsto \tau'_k \mapsto \tau'_{k+1}$$

and $1 \leq j \leq n$ (so that some argument is still missing), by the instances

$$\sigma(l[x]_p = r) \qquad \text{where } \sigma = \{x \mapsto \mathsf{partcall}(f, n - j + 1, x_1, \ldots, x_{j-1})\}$$

with $x_1, \ldots, x_{j-1}$ different fresh variables (if $j = 1$, no argument is added to the partial call). Clearly, we refer above to the types inferred in the original higher-order program. Nevertheless, if no type information is available, one could just instantiate the rules with all possible partial calls; this might introduce some useless rules but would be safe. For instance, consider the rule

$$f(x, xs) \;=\; map(x, xs)$$

[6] Observe that $n < k$ implies that function f returns a functional value.

where x is a functional variable of type $\mathbb{N} \mapsto \mathbb{N}$. Given the function $sum/2 \in \text{PCALLS}_{\mathcal{R}}$ with type $\mathbb{N} \mapsto \mathbb{N} \mapsto \mathbb{N}$, we replace the rule above by the following instance:

$$f(\mathsf{partcall}(sum, 1, n), xs) \;=\; map(\mathsf{partcall}(sum, 1, n), xs)$$

The instantiation of rules is applied repeatedly until no rule with a functional variable appears in the program.[7] Then, as in the previous step, we apply the transformation $(*)$ above as much as possible to the right-hand sides of the program. The following example illustrates this instantiation process.

Example 2. Consider again the transformed program $\mathcal{R}_2$ of Example 1. We have $\text{PCALLS}_{\mathcal{R}_2} = \{inc/1\}$. There is only one functional variable f (with type $\tau_1 \mapsto \tau_2$) in the rules defining map, hence we produce the following instantiated rules:

$$\begin{array}{ll} map(\mathsf{partcall}(inc, 1), [\,]) & = [\,] \\ map(\mathsf{partcall}(inc, 1), x : xs) & = \mathsf{apply}(\mathsf{partcall}(inc, 1), x) : map(\mathsf{partcall}(inc, 1), xs) \end{array}$$

Now, by reducing all calls to apply with a partcall as a first argument, we get the transformed program $\mathcal{R}_3$:

$$\begin{array}{ll} \mathsf{main}(xs) & = map(\mathsf{partcall}(inc, 1), xs) \\ map(\mathsf{partcall}(inc, 1), [\,]) & = [\,] \\ map(\mathsf{partcall}(inc, 1), x : xs) & = inc(x) : map(\mathsf{partcall}(inc, 1), xs) \\ inc(x) & = Succ(x) \end{array}$$

Note that $map(\mathsf{partcall}(inc, 1), \ldots)$ should be understood as a fresh function, e.g., we could rewrite the program as follows:

$$\begin{array}{llll} \mathsf{main}(xs) & = map_{inc}(xs) & map_{inc}([\,]) & = [\,] \\ inc(x) & = Succ(x) & map_{inc}(x : xs) & = inc(x) : map_{inc}(xs) \end{array}$$

Observe that no call to apply occurs in the final program and, thus, there is no need to add a definition for apply (i.e., the next step would not be necessary for this example). Determining the class of programs for which we can guarantee that no occurrence of apply appears in the transformed programs is an interesting subject for future work.

Let us note that this step of the transformation is safe since main can only be called with constructor terms as arguments. Otherwise, functional variables should be instantiated with all possible partial applications and not only those in $\text{PCALLS}_{\mathcal{R}}$. On the other hand, not all instantiations of functional variables will be *reachable* from main. The use of a *closure analysis* may improve the accuracy of the transformed program (but it will also add a significant time overhead in the defunctionalization process).

[7] Note that a function may have several functional arguments and, thus, we could apply the instantiation process to the instantiations of a rule previously considered.

3.3 Adding an Explicit Definition of apply.

In contrast to standard defunctionalization techniques (like [4, 12]), the transformation process so far may produce a first-order program with no occurrences of function apply in many common cases (as in the previous example).

In other cases, however, some occurrences of apply remain in the transformed program and a proper definition of apply should be added. This is the case, e.g., when there is a call to apply with a function call as a first argument. In this case, the value of the partial call will not be known until run time and, thus, we add the following sequence of rules:

$$\begin{array}{l}\mathsf{apply}(\mathsf{partcall}(f,n),x_1) = \mathsf{partcall}(f,n-1,x_1)\\ \mathsf{apply}(\mathsf{partcall}(f,n-1,x_1),x_2) = \mathsf{partcall}(f,n-2,x_1,x_2)\\ \ldots\\ \mathsf{apply}(\mathsf{partcall}(f,1,x_1,\ldots,x_{n-1}),x_n) = f(x_1,\ldots,x_n)\end{array}$$

for each function symbol $f/n \in \text{PCALLS}_{\mathcal{R}}$.

Our defunctionalization process can be effectively applied not only to programs using simple constructs such as $(map\ f\ \ldots)$ but also to programs that make essential use of higher-order features, as the following example illustrates.

Example 3. Consider the following higher-order program from [20]:

$$\begin{array}{ll}\mathsf{main}\ x\ y = f\ x\ y & \\ g\ r\ a = r\ (r\ a) & f\ Z = inc\\ inc\ n = Succ\ n & f\ (Succ\ n) = g\ (f\ n)\end{array}$$

where natural numbers are built from Z and $Succ$. The first step of the defunctionalization process returns

$$\begin{array}{ll}\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & \\ g(r,a) = \mathsf{apply}(r,\mathsf{apply}(r,a)) & f(Z) = \mathsf{partcall}(inc,1)\\ inc(n) = Succ(n) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n))\end{array}$$

Here, $\text{PCALLS}_{\mathcal{R}} = \{inc/1, g/2\}$. We only have a functional variable r in the rule defining function g (with associated type $\mathbb{N} \mapsto \mathbb{N}$) and, therefore, the following instances of the rules defining function g are added:

$$\begin{array}{l}g(\mathsf{partcall}(inc,1),a) = \mathsf{apply}(\mathsf{partcall}(inc,1),\mathsf{apply}(\mathsf{partcall}(inc,1),a))\\ g(\mathsf{partcall}(g,1,x),a) = \mathsf{apply}(\mathsf{partcall}(g,1,x),\mathsf{apply}(\mathsf{partcall}(g,1,x),a))\end{array}$$

By reducing all calls to apply with a partcall as a first argument, we get

$$\begin{array}{ll}\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & f(Z) = \mathsf{partcall}(inc,1)\\ g(\mathsf{partcall}(inc,1),a) = inc(inc(a)) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n))\\ g(\mathsf{partcall}(g,1,x),a) = g(x,g(x,a)) & inc(n) = Succ(n)\end{array}$$

Finally, since an occurrence of function apply remains in the program, we add the following rules:

$$\begin{array}{l}\mathsf{apply}(\mathsf{partcall}(inc,1),x) = inc(x)\\ \mathsf{apply}(\mathsf{partcall}(g,2),x) = \mathsf{partcall}(g,1,x)\\ \mathsf{apply}(\mathsf{partcall}(g,1,x),y) = g(x,y)\end{array}$$

The correctness of our defunctionalization transformation is an easy extension of that in [4, 12] by considering that function apply is strict in its first argument and, thus, our main extension, the instantiation of functional variables, is safe.

Note also that our approach is also safe at partial evaluation time where missing information (in the form of logical variables) might appear since the evaluation of higher-order calls containing free variables as functions is not allowed in current implementations of narrowing (i.e., such calls are *suspended* to avoid the use of higher-order unification [13]).

Regarding the code size increase due to our defunctionalization algorithm, the fact that it makes more higher-order information explicit comes at a cost: in the worst case, the source program can grow exponentially in the number of functions (e.g., when the program contains partial calls to all defined functions). Nevertheless, this case will happen only rarely and thus the code size increase is generally reasonable. Furthermore, the subsequent specialization phase is usually able to reduce the code (see Sect. 5).

4 Polyvariant Transformation

In this section, we introduce a source-to-source transformation that, given a program $\mathcal{R}$, returns a new program $\mathcal{R}'$ that is semantically equivalent to $\mathcal{R}$ but can be more accurately analyzed. Basically, our aim is to get the same information from a monovariant BTA over the transformed program $\mathcal{R}'$ as from a polyvariant BTA over the original program $\mathcal{R}$.

Intuitively speaking, we make a copy of each function definition for every call with different binding-times for its arguments. For simplicity, we only consider the basic *binding-times* S (static, known value) and D (dynamic, possibly unknown value). The least upper bound over binding-times is defined as follows:

$$\text{S} \sqcup \text{S} = \text{S} \qquad \text{S} \sqcup \text{D} = \text{D} \qquad \text{D} \sqcup \text{S} = \text{D} \qquad \text{D} \sqcup \text{D} = \text{D}$$

The least upper bound operation can be extended to sequences of binding-times in the natural way, e.g.,

$$\text{SDS} \sqcup \text{SSD} = \text{SDD} \qquad \text{SDS} \sqcup \text{DSD} = \text{DDD} \qquad \text{SDS} \sqcup \text{DSS} = \text{DDS}$$

A binding-time *environment* is a substitution mapping variables to binding-times. We will use the following auxiliary function B_e (adapted from [14]) for computing the binding-time of an expression:

$$\begin{array}{lll} B_e[[x]]\rho & = \rho(x) & (\text{if } x \in \mathcal{V}) \\ B_e[[\mathsf{h}(t_1, \ldots, t_n)]]\rho & = B_e[[t_1]]\rho \sqcup \ldots \sqcup B_e[[t_n]]\rho & (\text{if } \mathsf{h} \in \mathcal{C} \cup \mathcal{D}) \end{array}$$

where ρ denotes a binding-time environment. Roughly speaking, an expression $(B_e[[t]]\rho)$ returns S if t contains no variable which is bound to D in ρ, and D otherwise.

$$
\begin{array}{l}
poly\_trans(\{\,\}) = \{\,\} \\
poly\_trans(\{R\} \cup \mathcal{R}) = poly\_trans(\mathcal{R}) \\
\qquad \cup \begin{cases}
\{f_{\overline{b_n}}(\overline{t_n}) = pt(r, bte(f(\overline{t_n}), \overline{b_n})) \mid \overline{b_n} \in BT^n\} & \text{if } R = (f(\overline{t_n}) = r) \\
\{\mathsf{apply}_{b_n}(\mathsf{partcall}(f_{\overline{b_{n-1}}}, k, \overline{x_{n-1}}), x_n) = \mathsf{partcall}(f_{\overline{b_n}}, k-1, \overline{x_n}) \mid \overline{b_n} \in BT^n\} & \\
\qquad\qquad \text{if } R = (\mathsf{apply}(\mathsf{partcall}(f, k, \overline{x_{n-1}}), x_n) = \mathsf{partcall}(f, k-1, \overline{x_n})) & \\
\{\mathsf{apply}_{b_n}(\mathsf{partcall}(f_{\overline{b_{n-1}}}, k, \overline{x_{n-1}}), x_n) = f_{\overline{b_n}}(\overline{x_n}) \mid \overline{b_n} \in BT^n\} & \\
\qquad\qquad \text{if } R = (\mathsf{apply}(\mathsf{partcall}(f, k, \overline{x_{n-1}}), x_n) = f(\overline{x_n})) &
\end{cases}
\end{array}
$$

$$
\begin{array}{l}
pc(\{\,\}) = \{\,\} \\
pc(\{R\} \cup \mathcal{R}) = \begin{cases}
pc(\{f(t_1, \ldots, \mathsf{partcall}(g_{\overline{b_m}}, k, \overline{t_m}), \ldots, t_n) = r \mid \overline{b_m} \in BT^m\} \cup \mathcal{R}) & \\
\qquad \text{if } R = (f(\overline{t_n}) = r),\ t_i = \mathsf{partcall}(g, k, \overline{t_m}),\ i \in \{1, \ldots, n\} & \\
\{R\} \cup pc(\mathcal{R}) & \text{otherwise}
\end{cases}
\end{array}
$$

$$
pt(t, \rho) = \begin{cases}
t & \text{if } t \in \mathcal{V} \\
c(\overline{pt(t_n, \rho)}) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C} \\
f_{\overline{b_n}}(\overline{pt(t_n, \rho)}) & \text{if } t = f(\overline{t_n}),\ f \in \mathcal{D},\ B_e[\![t_i]\!]\rho = b_i,\ i = 1, \ldots, n \\
\mathsf{partcall}(f_{\overline{b_n}}, k, \overline{pt(t_n, \rho)}) & \text{if } t = \mathsf{partcall}(f, k, \overline{t_n}),\ B_e[\![t_i]\!]\rho = b_i,\ i = 1, \ldots, n \\
\mathsf{apply}_{b_2}(pt(t_1, \rho), pt(t_2, \rho)) & \text{if } t = \mathsf{apply}(t_1, t_2),\ B_e[\![t_i]\!]\rho = b_i,\ i = 1, 2
\end{cases}
$$

Fig. 1. Polyvariant transformation: functions *poly_trans* and *pt*

Given a linear term $f(\overline{t_n})$ (usually the left-hand side of a rule), and a sequence of binding-times $\overline{b_n}$ for f, the associated binding-time environment, $bte(f(\overline{t_n}), \overline{b_n})$, is defined as follows:

$$bte(f(\overline{t_n}), \overline{b_n}) = \{x \mapsto b_1 \mid x \in \mathcal{V}ar(t_1)\} \cup \ldots \cup \{x \mapsto b_n \mid x \in \mathcal{V}ar(t_n)\}$$

Definition 1 (polyvariant transformation). *Let $\mathcal{R}$ be a program and $\overline{b_n}$ be a sequence of binding-times for* main$/n$*. The polyvariant transformation of $\mathcal{R}$ w.r.t. $\overline{b_n}$, denoted by $\mathcal{R}^{\overline{b_n}}_{poly}$, is computed as follows:*

$$
\begin{array}{l}
\mathcal{R}^{\overline{b_n}}_{poly} = \{\ \mathsf{main}(\overline{x_n}) = pt(r, bte(\mathsf{main}(\overline{x_n}), \overline{b_n})) \mid \mathsf{main}(\overline{x_n}) = r \in \mathcal{R}\ \} \\
\qquad\qquad \cup\ poly\_trans(pc(\mathcal{R} \setminus \{\mathsf{main}(\overline{x_n}) = r\}))
\end{array}
$$

where the auxiliary functions poly_trans, pc and pt are defined in Fig. 1. Here, we denote by BT^n the set of all possible sequences of n binding-times.

Intuitively, the polyvariant transformation proceeds as follows:

– First, the left-hand side of function main is not labeled since there is no call to main in the program. The right-hand side is transformed as any other user-defined function using *pt* (see below).

– For the program rules (i.e., rules defining functions different from apply), we first label the occurrences of partcall in the left-hand sides using auxiliary function pc.[8] Observe that the first case of the definition of pc includes the transformed rule in the next recursive call since the left-hand side may contain several partcall arguments; in the second case, when no occurrence of partcall remains, the rule is deleted from the recursive call.
  Then, we replace the resulting rules by a number of copies labeled with all possible sequences of binding-times, whose right-hand sides are then transformed using function pt. Here, we could restrict the possible binding-times to those binding-times that are safe approximations of the arguments $\overline{t_m}$ of the partial call. However, we keep the current formulation for simplicity.
– Rules defining apply are transformed so that the binding-times of the partial function and the new argument are made explicit. Observe that we label the function symbol inside a partial call but not the partial call itself. Also, apply is just labeled with the binding-time of their second argument; the binding-time of the first argument is not needed since the binding-times labeling the function inside the corresponding partial call already contains more accurate information.
– Function pt takes a term and a binding-time environment and proceeds as follows:
  - Variables and constructor symbols are left untouched.
  - Function calls are labeled with the binding-times of their arguments according to the current binding-time environment. Function symbols in partial calls are also labeled in the same way.
  - Applications and partial calls are labeled as in function $poly\_trans$.

Observe that labeling functions with all possible sequences of binding-times produces usually a significant increase of code size. Clearly, one could perform a pre-processing analysis to determine the *call patterns* $f(\overline{b'_m})$ that may occur from the initial call to $\mathsf{main}(\overline{b_n})$. This approach, however, will involve a similar complexity as constructing the higher-order call graph of [20]. Here, we preferred to trade time complexity for space complexity. Furthermore, many of these copies are dead code and will be easily removed in the partial evaluation stage (see Sect. 5).

Example 4. Consider the defunctionalized program $\mathcal{R}$ of Example 3:

$$\begin{array}{ll}
\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & f(Z) = \mathsf{partcall}(inc,1) \\
g(\mathsf{partcall}(inc,1),a) = inc(inc(a)) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n)) \\
g(\mathsf{partcall}(g,1,x),a) = g(x,g(x,a)) & inc(n) = Succ(n) \\
\mathsf{apply}(\mathsf{partcall}(g,2),x) = \mathsf{partcall}(g,1,x) & \mathsf{apply}(\mathsf{partcall}(inc,1),x) = inc(x) \\
\mathsf{apply}(\mathsf{partcall}(g,1,x),y) = g(x,y) &
\end{array}$$

[8] For clarity, we assumed that all occurrences of partcall appear at the top level of arguments, i.e., either the argument t_i has the form $\mathsf{partcall}(\ldots)$ or it contains no occurrences of partcall.

Given the initial binding-times SD, our polyvariant transformation produces the following program $\mathcal{R}^{\text{SD}}_{poly}$:[9]

$$\mathsf{main}(x, y) = \mathsf{apply}_{\text{D}}(f_{\text{S}}(x), y)$$

$$f_{\text{S}}(Z) = \mathsf{partcall}(inc, 1)$$
$$f_{\text{S}}(Succ(n)) = \mathsf{partcall}(g_{\text{S}}, 1, f_{\text{S}}(n))$$

$$inc_{\text{S}}(n) = Succ(n)$$
$$inc_{\text{D}}(n) = Succ(n)$$

$$g_{\text{SD}}(\mathsf{partcall}(inc, 1), a) = inc_{\text{D}}(inc_{\text{D}}(a))$$
$$g_{\text{SD}}(\mathsf{partcall}(g_{\text{S}}, 1, x), a) = g_{\text{SD}}(x, g_{\text{SD}}(x, a))$$

$$\mathsf{apply}_{\text{D}}(\mathsf{partcall}(inc, 1), x) = inc_{\text{D}}(x)$$
$$\mathsf{apply}_{\text{D}}(\mathsf{partcall}(g_{\text{S}}, 1, x), y) = g_{\text{SD}}(x, y)$$

The next section presents a summary of an experimental evaluation conducted with a prototype implementation of the partial evaluation.

5 The Transformation in Practice

In this section, we present a summary of our progress on the development of a partial evaluator that follows the ideas presented so far. The undertaken implementation follows these directions:

- The system accepts higher-order programs which are first transformed using the techniques of Sect. 3 (defunctionalization) and Sect. 4 (polyvariant transformation).
- Then, the standard size-change analysis of [6] (for first-order programs) is applied to the transformed program.
- Finally, we annotate the program using the output of the size-change analysis and apply the specialization phase of the existing offline partial evaluator [17, 6]. We note that no propagation of binding-times is required here[10] since this information is already explicit in every function call thanks to the polyvariant transformation.

Table 1 shows the effectiveness of our transformation over the following examples: `ack`, the well known Ackermann's function, which is specialized for a given first argument; `bulyonkov`, a slight extension of the running example in [9]; `combinatorial`, a simple program including the computation of combinatorials; `changeargs`, another variation of the running example in [9]; `dfib`, a higher-order example that uses the well-known Fibonacci's function; `dmap`, a

[9] Actually, the transformation produces some more (useless) rules that we do not show for clarity. Note also that, according to our technique, the occurrence of *inc* in the expression $\mathsf{partcall}(inc, 1)$ should be labeled with an empty sequence of binding-times. However, for simplicity, we write just *inc*.

[10] In the original scheme, the binding-time of every function argument is required in order to identify *static* loops that can be safely unfolded.

Table 1. Benchmark results (run times, milliseconds)

benchmark	original	specialized		poly specialized	
	run time	run time	speedup	run time	speedup
ack	1526	507	3.01	522	2.92
bulyonkov	559	727	0.77	402	1.39
combinatorial	991	887	1.12	612	1.62
changeargs	772	1157	0.67	478	1.62
dfib (HO)	326	294	1.11	95	3.43
dmap (HO)	905	427	2.12	885	1.02
Average	**760**	**602**	**1.26**	**416**	**1.83**

Table 2. Benchmark results (code size, bytes)

benchmark	original	specialized		poly specialized	
	size	size	variation	size	variation
ack	951	3052	3.21	4168	4.38
bulyonkov	2250	3670	1.63	2440	1.08
combinatorial	2486	3546	1.43	6340	2.55
changeargs	3908	5335	1.37	5599	1.43
dfib (HO)	2911	4585	1.58	6204	2.12
dmap (HO)	2588	5236	2.02	3279	1.27
Average	**2321**	**4147**	**1.79**	**4408**	**1.90**

higher-order example with a function to map two functions to every element of a list.

For the first-order examples, we considered the previous offline partial evaluator of [17, 6], the only difference being that in the last two columns the considered program is first transformed with the polyvariant transformation. As it can be seen, the polyvariant transformation improves the speedups in three out of four examples.

For the higher-order examples, since the previous offline partial evaluator did not accept higher-order programs, we compare the new offline partial evaluator with an online partial evaluator for Curry that accepts higher-order functions [1]. In this case, we get an improvement in one of the examples (`dfib`) and a slowdown in the other one (`dmap`). This result is not surprising since an online partial evaluator is usually able to propagate much more information than an offline partial evaluator. Nevertheless, the important remark here is that we are able to deal with programs that could not be dealt with the old version of the offline partial evaluator.

Averages are obtained from the geometric mean of the speedups.

A critical issue of our transformation is that it might produce a significant increase of code size. This is explored in Table 2. Here, although the size of residual programs produced with our approach is slightly bigger than the size of residual programs obtained with previous approaches, it is still reasonable. Actually, our benchmarks confirm that most of the (dead) code added in the

polyvariant transformation has been removed at specialization time. Nevertheless, producing intermediate programs with are too large might be problematic if memory is exhausted. Thus we are currently considering the definition of some program analysis that can be useful for avoiding the introduction of (potentially) dead code during the transformation process.

6 Related Work and Conclusions

Let us first review some related works. Defunctionalization was first introduced by Reynolds [18] (see also [10], where a number of applications are presented). Defunctionalization has already been used in the context of partial evaluation (see, e.g., [8]) as well as in the online approach to narrowing-driven partial evaluation [1]. The main novelty w.r.t. these approaches is that we introduced a more aggressive defunctionalization by instantiating functional variables with all possible partial calls. Although it may increase code size, the transformed program has more information explicit and both size-change analysis and specialization may often produce better results.

Size-change analysis has been recently extended to higher-order functional programs in [20]. In contrast to our approach, Sereni proposes a direct approach over higher-order programs that requires the construction of a complex call graph which might produce less efficient binding-time analyses. We have applied our technique to the example in [20] and we got the same accuracy (despite the use of defunctionalization). A deeper comparison is the subject of ongoing work.

Regarding the definition of transformational approaches to polyvariant BTA, we only found the work of [9]. In contrast to our approach, Bulyonkov duplicates the function arguments so that, for every argument of the original function, there is another argument with its binding-time. Furthermore, some additional code to compute the binding-times of the calls in the right-hand sides of the functions is added. Then, a first stage of partial evaluation is run with some concrete values for the binding-time arguments of some function. As a result, the specialized program may include different versions of the same function (for different combinations of binding-times). Then, partial evaluation is applied again using the actual values of the static arguments. Our approach replaces the first stage of transformation and partial evaluation by a simpler transformation based on duplicating code and labeling function symbols. No experimental comparison can be made since we are not aware of any implementation of Bulyonkov's approach.

Other approaches to polyvariant BTA of higher-order programs include Mogensen's work [16] for functional programs and Vanhoof's modular approach [22] for Mercury programs. In contrast to our approach, Mogensen presents a *direct* (i.e., not based on defunctionalization) approach for polyvariant BTA of higher-order functional programs.[11] Vanhoof's approach is also a direct approach to polyvariant BTA of higher-order Mercury programs. A nice aspect of [22] is that no closure analysis is required, since closures are encapsulated in the notion

[11] Actually, Mogensen's approach includes some steps that resemble a defunctionalization process but never adds a definition for an explicit application function.

of binding-time. The integration of some ideas from [22] in our setting could improve the accuracy of the method and reduce the increase of code size.

Other related approaches to improve the accuracy of termination analysis by labeling functions can be found in [19], which is based on a standard technique from logic programming [5]. Here, some program clauses are duplicated and labeled with different *modes*—the mode of an argument can be *input*, if it is known at call time, or *output*, if it is unknown—in order to have a well-moded program where every call to the same predicate has the same modes. This technique can be seen as a simpler version of our polyvariant transformation.

To summarize, in this work we have introduced a transformational approach to polyvariant BTA of higher-order functional programs. Our approach is based on two different transformations: an improved defunctionalization algorithm that makes as much higher-order information explicit as possible, together with a polyvariant transformation that improves the accuracy of the binding-time propagation. We have developed a prototype implementation of the complete partial evaluator, the first offline narrowing-driven partial evaluator that deals with higher-order programs and produces polyvariant specializations. Our experimental results are encouraging and point out that the new BTA is efficient and still sufficiently accurate.

As for future work, there are a number of interesting issues that we plan to investigate further. As mentioned above, [22] presents some ideas that could be adapted to our setting in order to improve the accuracy of the BTA and to avoid the code explosion due to the absence of a separate closure analysis in our transformation. Also, the use of more refined binding-time domains (including partially static information as in, e.g., [16, 22]) may improve the accuracy of the specialization at a reasonable cost.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2008 for many useful comments and suggestions.

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 and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.
3. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming, FLOPS'99*, pages 335–352. Springer LNCS 1722, 1999.
5. K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

6. G. Arroyo, J.G. 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.
7. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
8. A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.
9. M.A. Bulyonkov. Extracting Polyvariant Binding Time Analysis from Polyvariant Specializer. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 59–65. ACM, New York, 1993.
10. O. Danvy and L.R. Nielsen. Defunctionalization at Work. In *PPDP*, pages 162–174. ACM, 2001.
11. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
12. J.C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. of 8th Italian Conf. on Logic Programming, GULP'93*, pages 569–585, 1993.
13. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
14. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
15. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.
16. T.Æ. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT, Vol.2*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 1989.
17. J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'05)*, pages 228–239. ACM Press, 2005.
18. J.C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–297, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).
19. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 2008. To appear.
20. 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.
21. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.
22. W. Vanhoof. Binding-Time Analysis by Constraint Solving. A Modular and Higher-Order Approach for Mercury. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 2000.