

# A Transformational Approach to Polyvariant BTA of Higher-Order Functional Programs<sup>\*</sup>

Gustavo Arroyo<sup>1</sup>, J.Guadalupe Ramos<sup>2</sup>, Salvador Tamarit<sup>1</sup>, and Germán Vidal<sup>1</sup>

<sup>1</sup> DSIC, Technical University of Valencia, E-46022, Valencia, Spain  
{garroyo, stamarit, gvidal}@dsic.upv.es

<sup>2</sup> Instituto Tecnológico de la Piedad, La Piedad, Michoacan, México  
guadalupe@dsic.upv.es

## 1 Introduction

Partial evaluation [10] 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 (and 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 [16], a conservative extension of rewriting to deal with logic variables (i.e., *unknown* information in our context). An offline approach to narrowing-driven partial evaluation has been introduced in [12]. In order to improve its accuracy, [5] adapts a *size-change analysis* [11] 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 away (i.e., replaced by fresh variables).

Unfortunately, the size-change analysis of [5] 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<sup>3</sup> 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.

---

<sup>\*</sup> 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).

<sup>3</sup> We consider the usual binding-times: static (definitely known at partial evaluation time) and dynamic (possibly unknown at partial evaluation time).

In this work, we present a transformational approach to overcome the above drawbacks. Basically, we first transform the original higher-order program by *defunctionalization* [13]. In particular, we introduce an extension of previous defunctionalization techniques (like [3, 8]) 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.

## 2 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). Our transformation extends previous approaches (e.g., [8]) in order to make as much higher-order information explicit as possible, so that the next steps of the BTA (size-change analysis and propagation of binding-times) can exploit it.

In the following, we consider constructor-based rewrite systems as (first-order) *programs*. Formally, a *term rewriting system* (TRS for short) is a set of rewrite rules  $l \rightarrow r$  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 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  $\mathcal{F} \cap \mathcal{V} = \emptyset$ . The set of variables appearing in a term  $t$  is denoted by  $Var(t)$ . A TRS  $\mathcal{R}$  is *constructor-based* if the left-hand sides of its rules have the form  $f(s_1, \dots, s_n)$  where  $s_i$  are constructor terms, i.e.,  $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ , for all  $i = 1, \dots, n$ .

A *position*  $p$  in a term  $t$  is represented by a sequence of natural numbers, where  $\epsilon$  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  $s$ .

In the following, we write  $\overline{o_n}$  for the *sequence of objects*  $o_1, \dots, o_n$ .

### 2.1 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 `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  $\text{partcall}(f, n)$ ).<sup>4</sup> For simplicity, we consider that  $\text{partcall}$  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  $\text{apply}$  and  $\text{partcall}$ , we apply the following transformation to the right-hand sides as much as possible:

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

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  $\text{main}$ , which is defined by a single rule of the form  $(\text{main } x_1 \dots x_n = r)$ , with  $x_1, \dots, x_n \in \mathcal{V}$  different variables, and that the program contains no call to this distinguished function.

*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):

$$\begin{array}{lll} \text{main } xs = \text{map } \text{inc } xs & \text{map } f [] & = [] \\ \text{inc } x & = \text{Succ } x & \text{map } f (x : xs) = f x : \text{map } f xs \end{array}$$

where natural numbers are built from  $Z$  and  $\text{Succ}$  and lists are built from  $[]$  and  $“:”$ . First, we make all applications and partial calls explicit:

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

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

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

## 2.2 Instantiation of Functional Variables.

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

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

<sup>4</sup> A similar representation is used in FlatCurry, the intermediate representation of the functional logic programming language Curry [7].

Then, for each function  $f/n \in \text{PCALLS}_{\mathcal{R}}$  with type<sup>5</sup>

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

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

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

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

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

with  $x_1, \dots, x_{j-1}$  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.

The instantiation of rules is applied repeatedly until no rule with a functional variable appears in the program.<sup>6</sup> 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:

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

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

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

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).

### 2.3 Adding an Explicit Definition of **apply**.

In contrast to standard defunctionalization techniques (like [3, 8]), 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).

<sup>5</sup> Observe that  $n < k$  implies that function  $f$  returns a functional value.

<sup>6</sup> 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.

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:

$$\begin{aligned} \text{apply}(\text{partcall}(f, n), x_1) &= \text{partcall}(f, n - 1, x_1) \\ \text{apply}(\text{partcall}(f, n - 1, x_1), x_2) &= \text{partcall}(f, n - 2, x_1, x_2) \\ \dots & \\ \text{apply}(\text{partcall}(f, 1, x_1, \dots, x_{n-1}), x_n) &= f(x_1, \dots, x_n) \end{aligned}$$

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  $(\text{map } f \dots)$  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 [15]:

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

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

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

Here,  $\text{PCALLS}_{\mathcal{R}} = \{\text{inc}/1, g/2\}$ . 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:

$$\begin{aligned} g(\text{partcall}(\text{inc}, 1), a) &= \text{apply}(\text{partcall}(\text{inc}, 1), \text{apply}(\text{partcall}(\text{inc}, 1), a)) \\ g(\text{partcall}(g, 1, x), a) &= \text{apply}(\text{partcall}(g, 1, x), \text{apply}(\text{partcall}(g, 1, x), a)) \end{aligned}$$

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

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

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

$$\begin{aligned} \text{apply}(\text{partcall}(\text{inc}, 1), x) &= \text{inc}(x) \\ \text{apply}(\text{partcall}(g, 2), x) &= \text{partcall}(g, 1, x) \\ \text{apply}(\text{partcall}(g, 1, x), y) &= g(x, y) \end{aligned}$$

The correctness of our defunctionalization transformation is an easy extension of that in [3, 8] 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 [9]).

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.

### 3 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:

$$S \sqcup S = S \quad S \sqcup D = D \quad D \sqcup S = D \quad D \sqcup D = D$$

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

$$SDS \sqcup SSD = SDD \quad SDS \sqcup DSD = DDD \quad SDS \sqcup DSS = DDS$$

A binding-time *environment* is a substitution mapping variables to binding-times. We will use the following auxiliary function  $B_e$  (adapted from [10]) for computing the binding-time of an expression:

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

where  $\rho$  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  $\rho$ , and D otherwise.

Given a 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:

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

$$\begin{aligned}
poly\_trans(\{ \}) &= \{ \} \\
poly\_trans(\{R\} \cup \mathcal{R}) &= poly\_trans(\mathcal{R}) \\
&\cup \begin{cases} \{f_{\bar{b}_n}(\bar{t}_n) = pt(r, bte(f(\bar{t}_n), \bar{b}_n)) \mid \bar{b}_n \in BT^n\} & \text{if } R = (f(\bar{t}_n) = r) \\ \{\text{apply}_{\bar{b}_n}(\text{partcall}(f_{\bar{b}_{n-1}}, k, \bar{x}_{n-1}), x_n) = \text{partcall}(f_{\bar{b}_n}, k-1, \bar{x}_n) \mid \bar{b}_n \in BT^n\} \\ \quad \text{if } R = (\text{apply}(\text{partcall}(f, k, \bar{x}_{n-1}), x_n) = \text{partcall}(f, k-1, \bar{x}_n)) \\ \{\text{apply}_{\bar{b}_n}(\text{partcall}(f_{\bar{b}_{n-1}}, k, \bar{x}_{n-1}), x_n) = f_{\bar{b}_n}(\bar{x}_n) \mid \bar{b}_n \in BT^n\} \\ \quad \text{if } R = (\text{apply}(\text{partcall}(f, k, \bar{x}_{n-1}), x_n) = f(\bar{x}_n)) \end{cases} \\
pt(t, \rho) &= \begin{cases} t & \text{if } t \in \mathcal{V} \\ c(\overline{pt(t_n, \rho)}) & \text{if } t = c(\bar{t}_n), c \in \mathcal{C} \\ f_{\bar{b}_n}(\overline{pt(t_n, \rho)}) & \text{if } t = f(\bar{t}_n), f \in \mathcal{D}, B_e[[t_i]]\rho = b_i, i = 1, \dots, n \\ \text{partcall}(f_{\bar{b}_n}, k, \overline{pt(t_n, \rho)}) & \text{if } t = \text{partcall}(f, k, \bar{t}_n), B_e[[t_i]]\rho = b_i, i = 1, \dots, n \\ \text{apply}_{\bar{b}_2}(\overline{pt(t_1, \rho)}, \overline{pt(t_2, \rho)}) & \text{if } t = \text{apply}(t_1, t_2), B_e[[t_i]]\rho = b_i, i = 1, 2 \end{cases}
\end{aligned}$$

**Fig. 1.** Polyvariant transformation: functions *poly\_trans* and *pt*

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

$$\begin{aligned}
\mathcal{R}_{poly}^{\bar{b}_n} &= \{ \text{main}(\bar{t}_n) = pt(r, bte(\text{main}(\bar{t}_n), \bar{b}_n)) \mid \text{main}(\bar{t}_n) = r \in \mathcal{R} \} \\
&\cup poly\_trans(\mathcal{R} \setminus \{ \text{main}(\bar{t}_n) = r \})
\end{aligned}$$

where the auxiliary functions *poly\_trans* 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*.
- 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.
- For the remaining rules, we replace them by a number of copies labeled with all possible sequences of binding-times,<sup>7</sup> whose right-hand sides are then transformed using function *pt*.

<sup>7</sup> Therefore, we introduce  $2^m$  copies of each rule defining a function  $f/m$ . As an alternative, one can perform a pre-processing analysis to determine the *call patterns*  $f(\bar{b}'_m)$  that may occur from the initial call to `main`( $\bar{b}_n$ ). This approach, however, will involve a similar complexity as constructing the higher-order call graph of [15]. Here,

- 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  $poly.trans$ .

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

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

Given the initial binding-times  $SD$ , our polyvariant transformation produces the following program  $\mathcal{R}_{poly}^{SD}$ .<sup>8</sup>

$$\begin{array}{ll}
 \text{main}(x, y) = \text{apply}_D(f_s(x), y) & \\
 f_s(Z) = \text{partcall}(inc, 1) & inc_s(n) = \text{Succ}(n) \\
 f_s(\text{Succ}(n)) = \text{partcall}(g_s, 1, f_s(n)) & inc_D(n) = \text{Succ}(n) \\
 \text{apply}_D(\text{partcall}(inc, 1), x) = inc_D(x) & g_{SD}(\text{partcall}(inc, 1), a) = inc_D(inc_D(a)) \\
 \text{apply}_D(\text{partcall}(g_s, 1, x), y) = g_{SD}(x, y) & g_{SD}(\text{partcall}(g_s, 1, x), a) = g_{SD}(x, g_{SD}(x, a))
 \end{array}$$

## 4 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. 2 (defunctionalization) and Sect. 3 (polyvariant transformation).
- Then, the standard size-change analysis of [5] (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 [12, 5]. We note that no propagation of binding-times is required here<sup>9</sup> since this information is already explicit in every function call thanks to the polyvariant transformation.

---

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.

<sup>8</sup> Actually, the transformation produces some more (useless) rules that we do not show for clarity.

<sup>9</sup> In the original scheme, the binding-time of every function argument is required to identify *static* loops that can be safely unfolded.

**Table 1.** Benchmark results

benchmark	original	specialized		poly specialized		benchmark	original	online		offline	
	run time	run time	speedup	run time	speedup		run time	run time	speedup	run time	speedup
ack	1954	490	3.99	483	4.05	doublefib	453	409	1.11	420	1.08
buly	689	989	0.7	481	1.43	mapH0	1199	508	2.36	1126	1.06
buly2	691	994	0.7	462	1.5	repfirst	2476	2524	0.98	2043	1.21
fib	1964	1846	1.06	1837	1.07	<b>Average</b>	<b>4128</b>	<b>3441</b>	<b>1.37</b>	<b>3589</b>	<b>1.12</b>
<b>Average</b>	<b>5298</b>	<b>4319</b>	<b>1.20</b>	<b>3263</b>	<b>1.75</b>						

(a) first-order programs

(b) higher-order programs

Table 1 (a) shows the effectiveness of the polyvariant transformation over some first-order programs (Ackermann’s function, Fibonacci’s function, and a couple of examples from [6]). Here, we consider the previous offline partial evaluator of [12, 5], 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 all examples. Table 1 (b) then considers some programs including higher-order functions. Since the previous offline partial evaluator did not accept higher-order programs, we now compare the new offline partial evaluator with an online partial evaluator for Curry that accepts higher-order functions [1]. As expected, the results in this case are worse compared with the online partial evaluator. Nevertheless, run times are still reasonable and the offline partial evaluator is much faster than the online one. Averages are obtained from the geometric mean of the speedups.

## 5 Related Work and Conclusions

Size-change analysis has been recently extended to higher-order functional programs in [15]. In contrast to our approach, Sereni proposes a direct approach over higher-order program 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 [15] 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 [6]. 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 related approaches to improve the accuracy of termination analysis by labeling functions can be found in [14], which is based on a standard technique

from logic programming [4]. 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. Our preliminary results with a prototype implementation are encouraging and point out that the new BTA is efficient and still sufficiently accurate.

## 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 and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *Proc. of FLOPS'99*, pages 335–352. Springer LNCS 1722, 1999.
4. K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
5. G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Proc. of LOPSTR'06*, pages 60–76. Springer LNCS 4407, 2007.
6. M.A. Bulyonkov. Extracting Polyvariant Binding Time Analysis from Polyvariant Specializer. In *Proc. of PEPM'93*, pages 59–65. ACM, New York, 1993.
7. M. Hanus (ed.). *Curry: An Integrated Functional Logic Language*. Available at: <http://www.informatik.uni-kiel.de/~mh/curry/>.
8. J.C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. of Italian Conf. on Logic Programming, GULP'93*, pages 569–585, 1993.
9. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
10. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
11. 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.
12. J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. ICFP'05*, pages 228–239. ACM, 2005.
13. 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).
14. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting, 2008. *Submitted*.
15. D. Sereni. Termination Analysis and Call Graph Construction for Higher-Order Functional Programs. In *Proc. of ICFP'07*, pages 71–84. ACM, 2007.
16. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.