# Fusing Generic Functions

Artem Alimarine and Sjaak Smetsers

Computing Science Institute
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
alimarin@cs.kun.nl, sjakie@cs.kun.nl

**Abstract.** Generic programming is accepted by the functional programming community as a valuable tool for program development. Several functional languages have adopted the generic scheme of type-indexed values. This scheme works by specialization of a generic function to a concrete type. However, the generated code is extremely inefficient compared to its hand-written counterpart. The performance penalty is so big that the practical usefulness of generic programming is compromised. In this paper we present an optimization algorithm that is able to completely eliminate the overhead introduced by the specialization scheme for a large class of generic functions. The presented technique is based on consumer–producer elimination as exploited by fusion, a standard general purpose optimization method. We show that our algorithm is able to optimize many practical examples of generic functions.
**AMS classification (2000):** 68N18, 68N19, 68Q55.
**CR classification (1998):** B.3.4, D.1.1, D.3.3, I.2.2.
**Keywords and phrases:** symbolic evaluation, fusion, generic functions, polytypic functions, functional programming languages, program transformation, typing, operational semantics.

## 1 Introduction

Generic programming is recognized as an important tool for minimizing boilerplate code that results from defining the same operation on different types. One of the most wide-spread generic programming techniques is the approach of type-indexed values [Hin00]. In this approach, a generic operation is defined once for all data types. For each concrete data type an instance of this operation is generated. This instance is an ordinary function that implements the operation on the data type. We say that the generic operation is *specialized* to the data type.

The generic specialization scheme uses a structural view on a data type. In essence, an algebraic type is represented as a sum of products of types. The structural representation uses *binary* sums and products. Generic operations are defined on these structural representations. Before applying a generic operation the arguments are converted to the structural representation, then the operation is applied to the converted arguments and then the result of the operation is converted back to its original form.

A programming language's feature is only useful in practice, if its performance is adequate. Directly following the generic scheme leads to very inefficient code, involving numerous conversions between values and their structural representations. The generated code additionally uses many higher-order functions (representing dictionaries corresponding to the type arguments). The inefficiency of generated code severely compromises the utility of generic programming.

In the previous work [AS04] we used a partial evaluation technique to eliminate generic overhead introduced by the generic specialization scheme. We proved that the described technique completely removes the generic overhead. However, the proposed optimization technique lacks termination analysis, and therefore works only for non-recursive functions. To make the technique work for instances on recursive types we abstracted the recursion with a Y-combinator and optimized the non-recursive part. This technique is limited to generic functions that do not contain recursion in their types, though the instance types can be recursive. Another disadvantage of the proposed technique is that it is tailored specifically to optimize generics, because it performs the recursion abstraction of generic instances.

The present paper describes a general purpose optimization technique that is able to optimize a significantly larger class of generic instances. In fact, the proposed technique eliminates the generic overhead in nearly all practical generic examples. When it is not able to remove the overhead completely, it still improves the code considerably. The presented optimization algorithm is based on fusion [AGS03,Chi94]. In its turn, fusion is based on the consumer-producer model: a producer produces data which are consumed by the consumer. Intermediate data are eliminated by combining (*fusing*) consumer-producer pairs.

The contributions of the present paper are:

- The original fusion algorithm is improved by refining both consumer and producer analyses. Our main goal is to achieve good fusion results for generics, but the improvements also appear to pay off for non-generic examples.
- We describe the class of generic programs for which the generic overhead is completely removed. This class includes nearly all practical generic programs.

In the next section we introduce the code generated by the generic specialization. This code is subject to the optimization described further in the paper. The generated code is represented in a simple functional language defined in section 3. Section 4 defines the semantics of fusion with no termination analysis. Basic properties of this fusion algorithm are discussed in section 5. Standard fusion with termination analysis [AGS03] is described in section 6. Sections 7 and 8 introduce our extensions to the consumer and the producer analyses. Fusion of generic programs is described in 9. The performance results for generic programs are presented in section 10. Section 11 discusses related work. Conclusions are presented and future work is discussed in section 12.

## 2 Generics

In this section we give a brief overview of the generic specialization scheme which is based on the approach by Hinze [Hin00]. Generic functions exploit the fact that any data type can be represented in terms of sums, pairs and unit, called the *base types*. These base types can be specified by the following Haskell-like data type definitions.

$$\begin{aligned}&\textbf{data}\ \mathbb{1} &&= \mathsf{Unit}\\ &\textbf{data}\ a\ \times\ b &&= \mathsf{Pair}\ a\ b\\ &\textbf{data}\ a\ +\ b &&= \mathsf{Inl}\ a \mid \mathsf{Inr}\ b\end{aligned}$$

A generic (*type-indexed*) function $g$ is specified by means of instances for these base types. The structural representation of a concrete data type, say $T$, is used to generate an instance of $g$ for $T$. The idea is to convert an object of type $T$ first to its structural representation, apply the generic operation $g$ to it, and convert the resulting object back from its structural to its original representation.

Suppose that the generic function $g$ has generic (*kind-indexed*) type $G$. Then the instance $g_T$ of $g$ for the concrete type $T$ has the following form.

$$g_T\,\vec{f} = adapt_{\langle G,T\rangle}(g_{T^\circ}\,\vec{f})$$

where $T^\circ$ denotes the structural representation of $T$, $g_{T^\circ}$ represents the instance of $g$ on $T^\circ$, and the adapter $adapt_{\langle G,T\rangle}$ takes care of conversion between $T$ and $T^\circ$. We will illustrate this generic specialization scheme with a few examples, starting with the structural representations of some familiar data types:

$$\begin{aligned}&\textbf{data}\ \mathsf{List}\ a &&= \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)\\ &\textbf{data}\ \mathsf{Tree}\ a &&= \mathsf{Leaf}\ a \mid \mathsf{Branch}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)\\ &\textbf{data}\ \mathsf{Rose}\ a &&= \mathsf{Rose}\ a\ (\mathsf{List}\ (\mathsf{Rose}\ a))\end{aligned}$$

These types are represented as

$$\begin{aligned}&\textbf{type}\ \mathsf{List}^\circ\ a &&= \mathbb{1} + a \times \mathsf{List}\ a\\ &\textbf{type}\ \mathsf{Tree}^\circ\ a &&= a + \mathsf{Tree}\ a \times \mathsf{Tree}\ a\\ &\textbf{type}\ \mathsf{Rose}^\circ\ a &&= a \times \mathsf{List}\ (\mathsf{Rose}\ a)\end{aligned}$$

Observe that only the top-level of the data definitions is converted to the structural form.

A type and its structural representation are isomorphic. The isomorphism is witnessed by a pair of conversion functions. For instance, for lists these functions are

$$\begin{aligned}&\mathsf{convTo}_{\mathsf{List}} :: \mathsf{List}\ a \to \mathsf{List}^\circ\ a\\ &\mathsf{convTo}_{\mathsf{List}}\ l \quad = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Nil} &&\to \mathsf{Inl}\ \mathsf{Unit}\\ &\qquad\qquad\qquad\quad\ \ \mathsf{Cons}\ x\ xs &&\to \mathsf{Inr}\ (\mathsf{Pair}\ x\ xs)\\ &\mathsf{convFrom}_{\mathsf{List}} :: \mathsf{List}^\circ\ a \to \mathsf{List}\ a\\ &\mathsf{convFrom}_{\mathsf{List}}\ l = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Inl}\ \mathsf{Unit} &&\to \mathsf{Nil}\\ &\qquad\qquad\qquad\quad\ \ \mathsf{Inr}\ (\mathsf{Pair}\ x\ xs) &&\to \mathsf{Cons}\ x\ xs\end{aligned}$$

To define a generic function $g$ the programmer has to provide the generic type $G$, and the instances on the base types. For example, the generic mapping is given by the type

$$\textbf{type}\ \mathsf{Map}\ a\ b = a \to b$$

and the base cases

$$\begin{array}{lll}
\mathsf{map}_{\mathbb{1}} & = \mathsf{case}\ u\ \mathsf{of} & \mathsf{Unit} \to \mathsf{Unit} \\
\mathsf{map}_{\times}\ l\ r\ p & = \mathsf{case}\ p\ \mathsf{of} & \mathsf{Pair}\ x\ y \to \mathsf{Pair}\ (l\ x)\ (r\ y) \\
\mathsf{map}_{+}\ l\ r\ e & = \mathsf{case}\ e\ \mathsf{of} & \mathsf{Inl}\ x \to \mathsf{Inl}\ (l\ x) \\
 & & \mathsf{Inr}\ y \to \mathsf{Inr}\ (r\ y)
\end{array}$$

This is all that is needed for the generic specializer to build an instance of map for any concrete data type $T$. As said before, such an instance is generated by interpreting the structural representation $T^{\circ}$ of $T$, and by creating an appropriate adapter. For instance, the generated mapping for $\mathsf{List}^{\circ}$ is

$$\begin{array}{l}
\mathsf{map}_{\mathsf{List}^{\circ}} :: \mathsf{Map}\ a\ b \to \mathsf{Map}\ (\mathsf{List}^{\circ}\ a)\ (\mathsf{List}^{\circ}\ b) \\
\mathsf{map}_{\mathsf{List}^{\circ}}\ f = \mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{\mathsf{List}}\ f))
\end{array}$$

Note how the structure of $\mathsf{map}_{\mathsf{List}^{\circ}}$ directly reflects the structure of $\mathsf{List}^{\circ}$. The adaptor converts the instance on the structural representation into an instance on the concrete type itself. E.g., the adapter converting $\mathsf{map}_{\mathsf{List}^{\circ}}$ into $\mathsf{map}_{\mathsf{List}}$ (i.e. the mapping function for List), has type

$$adapt_{\langle \mathsf{Map},\mathsf{List}\rangle} :: \mathsf{Map}\ (\mathsf{List}^{\circ}\ a)\ (\mathsf{List}^{\circ}\ b) \to \mathsf{Map}\ (\mathsf{List}\ a)\ (\mathsf{List}\ b)$$

The code for this adapter function is described below. We can now easily combine $adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}$ with $\mathsf{map}_{\mathsf{List}^{\circ}}$ to obtain a mapping function for the original List type.

$$\begin{array}{l}
\mathsf{map}_{\mathsf{List}} :: \mathsf{Map}\ a\ b \to \mathsf{Map}\ (\mathsf{List}\ a)\ (\mathsf{List}\ b) \\
\mathsf{map}_{\mathsf{List}}\ f = adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}\ (\mathsf{map}_{\mathsf{List}^{\circ}}\ f)
\end{array}$$

The way the adaptor works depends on the type of the generic function as well as on the concrete data type for which an instance is created. So called *embedding projections* are used to devise the automatic conversion. In essence such an embedding projection distributes the original conversion functions (the isomorphism between the type and its structural representation) over the type of the generic function. In general, the type of a generic function can contain arbitrary type constructors, including arrows. These arrows may also appear in the definition of the type for which an instance is derived. To handle such types in a uniform way, conversion functions are packed into *embedding-projection pairs*, EPs (e.g. see [HP01]), which are defined as follows.

$$\textbf{data}\ a \rightleftarrows b = \mathsf{EP}\ (a \to b)\ (b \to a)$$

For instance, packing the List conversion functions into an EP leads to:

$$\begin{array}{l}
\mathsf{conv}_{\mathsf{List}} :: \mathsf{List}\ a \rightleftarrows \mathsf{List}^{\circ}\ a \\
\mathsf{conv}_{\mathsf{List}} = \mathsf{EP}\ \mathsf{convTo}_{\mathsf{List}}\ \mathsf{convFrom}_{\mathsf{List}}
\end{array}$$

Now the adapter for $G$ and $T$ can be specified in terms of embedding projections using the EP that corresponds to the isomorphism between $T$ and $T^\circ$ as a basis. Actually, embedding projections are represented as a generic function themselves. This has the advantage that we can use the same specialization scheme for embedding projections that is used for other generic functions. More concretely, an embedding projection is a generic function ep with the generic type $a \rightleftarrows b$, and the base cases:

$$\begin{aligned}
\mathsf{ep}_{\mathbb{1}} &= \mathsf{EP}\ \mathsf{map}_{\mathbb{1}}\ \mathsf{map}_{\mathbb{1}} \\
\mathsf{ep}_{\times}\ f\ g &= \mathsf{EP}\ (\mathsf{map}_{\times}\ (\mathsf{to}\ f)\ (\mathsf{to}\ g))\ (\mathsf{map}_{\times}\ (\mathsf{from}\ f)\ (\mathsf{from}\ g)) \\
\mathsf{ep}_{+}\ f\ g &= \mathsf{EP}\ (\mathsf{map}_{+}\ (\mathsf{to}\ f)\ (\mathsf{to}\ g))\ (\mathsf{map}_{+}\ (\mathsf{from}\ f)\ (\mathsf{from}\ g)) \\
\mathsf{ep}_{\rightarrow}\ f\ g &= \mathsf{EP}\ (\mathsf{mapAR}\ (\mathsf{from}\ f)\ (\mathsf{to}\ g))\ (\mathsf{mapAR}\ (\mathsf{to}\ f)\ (\mathsf{from}\ g)) \\
\mathsf{ep}_{\rightleftarrows}\ f\ g &= \mathsf{EP}\ (\mathsf{mapEP}\ (\mathsf{to}\ f)\ (\mathsf{from}\ f)(\mathsf{to}\ g)\ (\mathsf{from}\ g)) \\
&\qquad\quad (\mathsf{mapEP}\ (\mathsf{from}\ f)\ (\mathsf{to}\ f)(\mathsf{from}\ g)\ (\mathsf{to}\ g))
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{to}\ e &= \mathsf{case}\ e\ \mathsf{of}\ \mathsf{EP}\ t\ f \rightarrow t \\
\mathsf{from}\ e &= \mathsf{case}\ e\ \mathsf{of}\ \mathsf{EP}\ t\ f \rightarrow f \\
\mathsf{mapAR}\ a\ r\ f &= r \circ f \circ a \\
\mathsf{mapEP}\ t_a\ f_a\ t_r\ f_r\ e &= EP\ (t_r \circ \mathsf{to}\ e \circ f_a)\ (t_a \circ \mathsf{from}\ e \circ f_r)
\end{aligned}$$

These instances are based on the basic instances of the previously defined function map.

Apart from the usual instances for sum, pair and unit, we have included the instances on $\rightarrow$ and $\rightleftarrows$. In particular the latter might look somewhat mysterious. The reason for specifying this instance is rather technical: it appears in the adapter of the specialized version of ep for a concrete type $T$, e.g. see section 9.1

The generic specializer generates the instance of ep specific to a generic function, again by interpreting its generic type. E.g. for mapping (with the generic type $\mathsf{Map}\ a\ b$) we get:

$$\begin{aligned}
&\mathsf{ep}_{\mathsf{Map}} :: (a_1 \rightleftarrows a_2) \rightarrow (b_1 \rightleftarrows b_2) \rightarrow (\mathsf{Map}\ a_1\ b_1 \rightleftarrows \mathsf{Map}\ a_2\ b_2) \\
&\mathsf{ep}_{\mathsf{Map}}\ a\ b = \mathsf{ep}_{\rightarrow}\ a\ b
\end{aligned}$$

Now the adaptor $adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}$ is the from-component of this embedding projection applied to $\mathsf{conv}_{\mathsf{List}}$ twice.

$$adapt_{\langle \mathsf{Map},\mathsf{List}\rangle} = \mathsf{from}\ (\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}})$$

To compare the generated version of map with its handwritten counterpart, e.g.

$$\begin{aligned}
\mathsf{map}\ f\ l = \mathsf{case}\ l\ \mathsf{of}\ \ & Nil \rightarrow Nil \\
& \mathsf{Cons}\ x\ xs \rightarrow \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)
\end{aligned}$$

we have inlined the adapter and the instance for the structural representation in the definition of $\mathsf{map}_{\mathsf{List}}$ resulting in

$$\begin{aligned}
\mathsf{map}_{\mathsf{List}}\ f = \mathsf{from}\ &(\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}}) \\
&(\mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{\mathsf{List}}\ f)))
\end{aligned}$$

Clearly, the generated version is much more complicated than the handwritten one, not only in terms of readability but also in terms of efficiency. The latter is the main concern of this paper. The reasons for inefficiency are the intermediate data structures for the structural representation and the extensive usage of higher-order functions. In the rest of the paper we present an optimization technique for generic functions which is based on fusion, and show that this technique is capable of removing all generic overhead, for a large class of generic functions.

## 3 Language

In this section we present the syntax of a simple core functional language that supports essential aspects of functional programming such as pattern matching and higher-order functions. The fusion semantics of this core language is described in the next section. First we introduce some more or less common terminology and notation.

**Notation 3.1 (Vectors).**

- We will use the *vector* $\vec{V}$ for $(V_1, \ldots, V_n)$. The length of a vector $V$ is indicated by $|\vec{V}|$
- If $V$ is a vector then $V_i$ denotes the $i^{\text{th}}$ element of $V$, and $V_{i..j}$ the (sub)vector $(V_i, \ldots, V_j)$. If $i > j$ then $V_{i..j} = ()$.
- Let $V, W$ be vectors. $V \star W$ denotes the concatenation of $V$ and $W$.

We define the syntax in two steps: expressions and functions.

**Definition 3.2 (Expressions).**

- *The set of* expressions *is defined as*

$$E ::= x \mid C\ \vec{E} \mid F\ \vec{E} \mid x\ @\ \vec{E}.$$

  *Here $x$ ranges over variables, $C$ over data constructors and $F$ over function symbols.*
- *By $E \subseteq E'$ we denote that $E$ is a subexpression of $E'$, and by $E \subset E'$ we indicate* proper *subexpressions (i.e. $E \neq E'$).*
- *Each (function or constructor) symbol has an* arity*: a natural number that indicates the maximum number of arguments to which the symbol can be applied. An expression $E$ is* well-formed *if the actual arity of applications occurring in $E$ never exceeds the formal arity of the applied symbols. From now on we will only consider well-formed expressions.*

Pattern matching is allowed only at the top level of a function definition. Moreover, only one pattern match per function is permitted and the patterns themselves have to be simple (free of nesting).

**Definition 3.3 (Functions).**

- *The set of* function bodies *is defined as follows.*

$$B ::= E \mid \mathsf{case}\ x\ \mathsf{of}\ P_1 \to E_1 \cdots P_n \to E_n$$
$$P ::= C\ \vec{x}$$

  *Variables in a pattern* $P$ *are called* pattern variables
- *The set of free variables in* $B$ *is indicated by* $\mathrm{FV}(B)$.
- *A* function definition *has the form* $F\ \vec{x} = B_F$ *with* $\mathrm{FV}(B_F) \subseteq \vec{x}$. *The* arity *of* $F$ *is* $|\vec{x}|$.
- $F$ *is called a* case function *if it starts with a pattern match* $F\ \vec{x} = \mathsf{case}\ x_i\ \mathsf{of}\ \ldots$. *We also say that* $F$ *is a* case function in $i$ *to indicate that the pattern match occurs on the* $i^{th}$ *parameter.*
- *A* component *is a set of mutually dependent functions. Let* $F$ *be a function. By* $\widehat{F}$ *we denote the component to which* $F$ *belongs.*

Data constructors are introduced via an *algebraic type definition*. Such a type definition not only specifies the *type* of each data constructor but also its *arity*. For readability reasons in this paper we will use a Haskell-like syntax in the examples.

## 4 Semantics of Fusion

Most program transformation methods use the so-called *unfold/fold* mechanism to convert expressions and functions. During an *unfold step*, a call to a function is replaced by the corresponding function body in which appropriate parameter substitutions have been performed. During a *fold step*, an expression is replaced by a call to a function of which the body matches that expression.

In the present paper we will use a slightly different way of both unfolding and folding. First of all, we do not unfold all possible function applications but restrict ourselves to so called *consumer–producer* pairs. In a function application $F(\ldots, S(\ldots), \ldots)$ the function $F$ is called a consumer and the function or constructor $S$ a producer. The intuition behind this terminology is that $F$ consumes the result produced by $S$. Suppose we have localized a consumer–producer pair in an expression $E$. More precisely, $E$ contains a subexpression $F\ \vec{E}$, with $E_i = S\ \vec{D}$. Say $k = |\vec{E}|$, and $l = |\vec{D}|$. The idea of *fusion* is to replace this pair consisting of two calls by a single call to the combined function $F_i S^l$ resulting in the application $F_i S^l\ E_{1..(i-1)} \star \vec{D} \star E_{(i+1)..k}$. Moreover, if this combined function is used the first time, a new function definition is generated that contains the body of the consumer $F$ in which $S\ (y_1, \ldots, y_l)$ is substituted for $x_i$. Note that this fusion mechanism does not require any explicit folding steps anymore. As an example consider the following definition of app, and the auxiliary function foo.

**Example 4.1.**

$$\begin{aligned}
\mathsf{app}\ l\ t &= \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} &&\to t\\
&\qquad\qquad\quad\ \ \mathsf{Cons}\ x\ xs &&\to \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t)\\
\mathsf{foo}\ x\ y\ z &= \mathsf{app}\ (\mathsf{app}\ x\ y)\ z
\end{aligned}$$

The first fusion step leads to the creation of a new function, say app_app, and replaces the nested applications of app by a single application of this new function. The result is shown below.

$$\begin{array}{lll} \mathsf{foo}\ x\ y\ z & = \mathsf{app\_app}\ x\ y\ z & \\ \mathsf{app\_app}\ x\ y\ z & = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} & \rightarrow \mathsf{app}\ y\ z \\ & \quad\ \mathsf{Cons}\ x\ xs & \rightarrow \mathsf{Cons}\ x\ (\mathsf{app}\ (\mathsf{app}\ xs\ y)\ z) \end{array}$$

The description of how the body of the new function app_app is created is given at the end of this section. foo itself does not contain consumer-producer pairs anymore; the only pair appears in the body of app_app, namely $\mathsf{app}\ (\mathsf{app}\ xs\ y)\ z$. Again these nested calls are replaced by app_app, and since app_app has already been created, no further steps are necessary.

$$\begin{array}{lll} \mathsf{app\_app}\ x\ y\ z = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} & \rightarrow \mathsf{app}\ y\ z \\ \quad\ \mathsf{Cons}\ x\ xs & \rightarrow \mathsf{Cons}\ x\ (\mathsf{app\_app}\ xs\ y\ z) \end{array}$$

This example shows that we need to determine whether a function - app_app in the example - has already been generated. To facilitate this, with each newly created function we associate a special unique name, a so called *symbol tree*. These symbol trees contain all the necessary information to determine whether a new function is equal to an existing one.

**Definition 4.2 (Symbol Trees).**

- *The set of* symbol trees *is defined by the following syntax. In this definition, $S$ ranges over function and constructor symbols. The special symbol $\Box$ is used to denote anonymous variables.*

$$\begin{array}{l} T ::= S\,\vec{T'} \\ T' ::= \Box \mid T \end{array}$$

- *The* root *of a tree $T = S\vec{T'}$ (denoted by $\ulcorner T \urcorner$) is the symbol $S$. The* arity *of a tree $T$ (indicated by $\mathsf{ar}(T)$) is the number of $\Box$ symbols in $T$.*
- *By $T[i{\leftarrow}V]$ we denote the term that is obtained from $T$ by substituting $V$ for the $i^{th}$ occurrence of $\Box$, in a depth-first, left-to-right numbering of $\Box$ symbols.*

$$\begin{array}{rl} \Box[1{\leftarrow}V] = V & \\ S\vec{T}[i{\leftarrow}V] = S\,(\ldots, T_j[i - a_1^{j-1}{\leftarrow}V], \ldots), & \\ \qquad \textit{for } j \textit{ such that } 0 < i - a_1^{j-1} \leq a_j & \\ \qquad \textit{where } a_i = \mathsf{ar}(T_i) \textit{ and } a_1^{j-1} = \sum_{l=1}^{j-1} a_l & \end{array}$$

- *By $\Box^k$ we denote the vector. $(\underbrace{\Box, \ldots, \Box}_{k})$.*
- *We have two auxiliary operations on trees for respectively increasing and decreasing the arity:*

1. $S\vec{T} \boxplus k = S\vec{T} \star \Box^k$
2. $S\vec{T} \boxminus k$ *is the tree obtained by removing the last* $k$ *occurrences of* $\Box$ *(using the same numbering as above). Formally:*

$$
\begin{array}{lll}
S\vec{T} \boxminus 0 & = S\vec{T} & \\
S\vec{T} \boxminus k+1 & = S(\vec{T}^{\boxminus}) \boxminus k & \\
\textit{where} & & \\
(T_1, \ldots, T_l)^{\boxminus} & = (T_1, \ldots, T_{l-1}), & \textit{if } T_l = \Box \\
 & = (T_1, \ldots, T_{l-1}, S'(\vec{V}^{\boxminus})), & \textit{if } T_l = S'\vec{V} \textit{ and } \mathsf{ar}(T_l) > 0 \\
 & = (T_1, \ldots, T_{l-1})^{\boxminus} \star (T_l), & \textit{otherwise}
\end{array}
$$

– *Let* $S$ *be a symbol, and* $T$ *a tree.* $T$ *is called* cyclic *in* $S$ *if* $T$ *contains a path on which* $S$ *occurs more than once, i.e. if for some tree* $T'$ *one has* $S(\ldots, T', \ldots) \subseteq T$ *and* $S(\ldots) \subseteq T'$.

A symbol tree can easily be converted to a function that corresponds to the original expression from which this tree has been created.

**Definition 4.3 (Converting Symbol Trees).** *Let* $T$ *be a symbol tree of arity* $k$. *The operation* $[\![T]\!]$ *yields a function* $F_T(x_1, \ldots, x_k) = E$, *of which the body* $E$ *results from* $T$ *after substituting* $x_i$ *for the* $i^{th}$ *occurrence of* $\Box$, *for all* $i \leq k$. *Substitutions are performed in parallel.*

As said before, the evaluation of a function application possibly leads to the creation of new functions. The name (symbol tree) of that new function is created from the symbol tree corresponding to the consumer and the symbol tree or data constructor of the producer.

**Definition 4.4 (Building Symbol Trees).**

– ***Basic trees:*** *With each initial function* $F$, *say with arity* $n$, *we associate the tree* $F\Box^n$
– ***New trees:*** *Let* $F$ *be a function with symbol tree* $T_F$, *and arity* $n$.
*There are two ways to introduce new functions in which* $F$ *is involved, and hence to introduce new symbol trees: (1) when* $F$ *appears as a consumer in a consumer-producer pair (definition 4.9) or (2) when the arity of* $F$ *is increased (definition 4.5).*
1. *Let* $S$ *be a function or data constructor, say with arity* $m$. *Suppose we are using* $S$ *with* $k$ *actual arguments,* $k \leq m$. *The result of combining* $F$ *with* $S$ *at argument position* $i$, $i \leq n$, *is a new symbol tree* $F_iS^k$ *defined as follows*
   - $S$ *is a function: Let* $T_S$ *be the symbol tree of* $S$. *Then*
   $$F_iS^k = T_F[i \leftarrow T_S \boxminus (m-k)].$$
   - $S$ *is a constructor:*
   $$F_iS^k = T_F[i \leftarrow S\Box^k].$$

2. *The result of increasing the arity of* $T_F$ *by* $k$ *is a symbol tree*

$$F^{\oplus k} = T_F \boxplus k.$$

The above construction of symbol trees is order independent. For instance, there are two ways to evaluate an application $F(G(H(\ldots))$, namely one can start with the $G$–$H$ pair and combine the result with $F$, or one can start with $F$–$G$ and combine the result with $H$. Both ways, however, will lead to the same tree. The same holds for an application like $F(G(\ldots), H(\ldots))$.

Unfolding (which stands in our system for the creation of new function bodies) is based on a notion of substitution for expressions. However, due to the restriction on our syntax with respect to higher-order expressions (recall that a higher-order expression should start with a variable) we cannot use a straightforward definition of substitution. Suppose we try to substitute an expression $D = F\,\vec{D'}$ for $x$ in $x \mathbin{@} \vec{E}$. This becomes problematic if $\mathsf{arity}(F) < |\vec{D'}| + |\vec{E}|$, because in the resulting application $F\,\vec{D'} \star \vec{E}$ is not well-formed. To solve this problem we introduce a new function built from the definition of $F$ by supplying it with additional arguments that *increase* its formal arity. This is made precise below.

**Definition 4.5 (Raised Functions).** *The operations* $\mathcal{R}[\![\cdot]\!]\cdot$ *and* $F^{\oplus\cdot}$ *are defined by simultaneous induction:*

– *Let* $B$ *be a function body,* $\vec{E}$ *a list of expressions. The result of applying* $B$ *to* $\vec{E}$*, denoted as* $\mathcal{R}[\![B]\!]\vec{E}$ *is given by*

$$\begin{aligned}
&\mathcal{R}[\![x]\!]\vec{E} && = x \mathbin{@} \vec{E}\\
&\mathcal{R}[\![C\,\vec{D}]\!]\vec{E} && = C\,\vec{D} \star \vec{E}\\
&\mathcal{R}[\![G\,\vec{D}]\!]\vec{E} && = G\,\vec{D} \star \vec{E},\ \textit{if } \delta \leq 0\\
& && = G^{\oplus\delta}\,\vec{D} \star \vec{E},\ \textit{otherwise}\\
& && \quad \textit{where } \delta = |\vec{D}| + |\vec{E}| - \mathsf{arity}(G)\\
&\mathcal{R}[\![x \mathbin{@} \vec{D}]\!]\vec{E} && = x \mathbin{@} \vec{D} \star \vec{E}\\
&\mathcal{R}[\![\mathsf{case}\ x\ \mathsf{of}\ \ldots P_i \to D_i \ldots]\!]\vec{E} && = \mathsf{case}\ x\ \mathsf{of}\ \cdots P_i \to \mathcal{R}[\![D_i]\!]\vec{E} \cdots
\end{aligned}$$

– *Let* $F$ *be a function* $F\,\vec{x} = B_F$*, with arity* $n$*. The result of* raising the arity of $F$ with $k$ *is a function given by:*

$$F^{\oplus k}\,\vec{x} \star (y_1, \ldots, y_k) = \mathcal{R}[\![B_F]\!](y_1, \ldots, y_k)$$

**Remark 4.6.** Observe that this operation can trigger the creation of more raised functions if the new arguments $\vec{y}$ are added to an application that requires fewer than $k$ arguments to become fully applied.

Now, the definition of substitution becomes straightforward.

**Definition 4.7 (Substitution).** *A substitution $\rho$ is a function that assigns expressions to variables. This induces the following operation on expressions*

$$\begin{aligned}
\Sigma[\![x]\!]\rho &= \rho(x) \\
\Sigma[\![F\,\vec{E}]\!]\rho &= F\,\Sigma[\![\vec{E}]\!]\rho \\
\Sigma[\![C\,\vec{E}]\!]\rho &= C\,\Sigma[\![\vec{E}]\!]\rho \\
\Sigma[\![x\ @\ \vec{E}]\!]\rho &= \mathcal{R}[\![\rho(x)]\!](\Sigma[\![\vec{E}]\!]\rho)
\end{aligned}$$

If a substitution is applied to a function body $B$ we have to be careful when $B$ starts with a pattern match. For, the result of such a substitution does not lead to a valid expression if it substitutes a non-variable expression for the selector. We solve this problem by combining consumers and producers in a more sophisticated way. This has been done below. But first we introduce an auxiliary operation to perform pattern matching.

**Definition 4.8 (Pattern Matching).** *Let $F\,\vec{x} = E_F$ be a case function in $i$ with arity $k$, and $B$ be a function body. The result of substituting $B$ for $x_i$ in $F$, denoted as $\mathcal{M}_i[\![B]\!]\langle F\,\vec{x} = E_F\rangle$, is defined by induction on $B$ in the following way.*

$$\begin{aligned}
&\mathcal{M}_i[\![y]\!]\langle F\,\vec{x} = \mathsf{case}\ x_i\ \mathsf{of}\ \ldots\rangle && = \mathsf{case}\ y\ \mathsf{of}\ \ldots \\
&\mathcal{M}_i[\![G\,\vec{E}]\!]\langle F\,\vec{x} = \ldots\rangle && = F\,x_{1..i-1} \star (G\,\vec{E}) \star x_{i+1..k} \\
&\mathcal{M}_i[\![y\ @\ \vec{E}]\!]\langle F\,\vec{x} = \ldots\rangle && = F\,x_{1..i-1} \star (y\ @\ \vec{E}) \star x_{i+1..k} \\
&\mathcal{M}_i[\![C_j\,\vec{E}]\!]\langle F\,\vec{x} = \mathsf{case}\ x_i\ \mathsf{of}\ \ldots C_j\,\vec{y_j} \to A_j \ldots\rangle && = \Sigma[\![A_j]\!][\vec{E}/\vec{y_j}] \\
&\mathcal{M}_i[\![\mathsf{case}\ y_k\ \mathsf{of}\ \ldots P_j \to E_j \ldots]\!]\langle F\,\vec{x} = E_F\rangle && \\
&\qquad = \mathsf{case}\ y_k\ \mathsf{of}\ \ldots P_j \to \mathcal{M}_i[\![E_j]\!]\langle F\,\vec{x} = E_F\rangle \ldots &&
\end{aligned}$$

*Here $[E/x]$ denotes the substitution of $E$ for $x$.*

**Definition 4.9 (Fused Functions).** *Let $F$ be a function, and let $S$ be a function or constructor symbol. Assume that the arities of $F, S$ are $m, n$ respectively, and that $S$ is applied to $k$ arguments, $k \leq n$. The result of* fusing *$F$ with such a $k$-ary version of $S$ at argument position $i$, $i \leq m$ is a function $F_i S^k$ defined as follows.*

1. *$F\,\vec{x} = \mathsf{case}\ x_i\ \mathsf{of}\ \cdots$. Then we distinguish the following two cases.*
   (a) *$S$ is a function, say with definition $S\,\vec{z} = B_S$. Then*

$$F_i S^k\ x_{1..(i-1)} \star \vec{z} \star x_{(i+1)..m} = \mathcal{M}_i[\![B_S]\!]\langle F\,\vec{x} = \mathsf{case}\ x_i\ \mathsf{of}\ \cdots\rangle$$

   (b) *$S$ is a constructor. Then*

$$F_i S^k\ x_{1..(i-1)} \star \vec{y} \star x_{(i+1)..m} = \mathcal{M}_i[\![S\,\vec{y}]\!]\langle F\,\vec{x} = \mathsf{case}\ x_i\ \mathsf{of}\ \cdots\rangle\ \ (|\vec{y}| = k)$$

2. *$F\,\vec{x} = E$. In that case*

$$F_i S^k\ x_{1..(i-1)} \star \vec{y} \star x_{(i+1)..m} = \Sigma[\![E]\!][S\,\vec{y}/x_i]\ \ (|\vec{y}| = k)$$

Observe that the body of $F$ in the latter case might start with a pattern match. But this pattern match is not on the variable $x_i$ for which $S$ is substituted, and hence this substitution will not produce an illegal function body.

**Example 4.10.** The body of the function app_app of example 4.1 results from applying rule 1a of definition 4.9:

$$\mathcal{M}_1[\![\mathsf{case}\ l\ \mathsf{of}\ \cdots]\!]\langle \mathsf{app}(l,t) = \mathsf{case}\ l\ \mathsf{of}\ \cdots\rangle$$

As a result, the case of the consumer is pushed into the alternatives of the producer where it is eliminated, leading to:

$$\begin{array}{lll} \mathsf{app\_app}\ x\ y\ z = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} & \rightarrow \mathsf{app}\ y\ z \\ \qquad\qquad\qquad\qquad\quad \mathsf{Cons}\ x\ xs & \rightarrow \mathsf{Cons}\ x\ (\mathsf{app}\ (\mathsf{app}\ xs\ y)\ z) \end{array}$$

During fusion (parts of) the user defined functions are examined for consumer-producer pairs that can be fused. If such a fusion introduces a new function this new function itself also becomes a source for new consumer-producer pairs. This leads to the following algorithm.

**Definition 4.11 (Fusion).** *Let $\mathcal{F}$ be a set of functions. Evaluation of $\mathcal{F}$ consists of repeatedly performing the following three steps until no more consumer-producer pairs can be found (step 1 is no longer successful).*

1. *Look for a function body $B$ in $\mathcal{F}$ that contains a* consumer-producer pair
$$R = F\ E_{1..(i-1)} \star (S\ \vec{D}) \star E_{(i+1)..|\vec{E}|}$$
2. *Let $F_i S^k$ be the symbol tree that corresponds to that consumer-producer pair. Replace $R$ by the expression*
$$F_i S^k\ E_{1..(i-1)} \star \vec{D} \star E_{(i+1)..|\vec{E}|}$$
3. *Set $\mathcal{F} = \mathcal{F} \cup \{F_i S^k\ \vec{z} = B'\}$.*
*Here $B'$ is given by the definitions 4.9. To determine whether $F_i S^k$ is already present in $\mathcal{F}$ we use the equality on symbol trees; not on expressions, i.e. we do not compare function bodies.*

**Remark 4.12.** We do not impose any evaluation order, since this order is irrelevant for the outcome. (See also property 5.2.)

**Example 4.13.**

$$\begin{array}{ll} \mathsf{plusOrMin}\ s\ m\ n & = s\ (\mathsf{Pair\ Plus\ Min})\ m\ n \\ \mathsf{First}\ p & = \mathsf{case}\ p\ \mathsf{of}\ \mathsf{Pair}\ x\ y \rightarrow x \\ \mathsf{foo}\ m\ n & = \mathsf{plusOrMin\ First}\ m\ n \end{array}$$

The only consumer-producer pair occurs in foo. It will lead to the creation of a new function, $\mathsf{plusOrMin}_1\mathsf{First}^0$. In the new body of this new function the application of First will have 3 arguments, so the arity of First has to be increased by

2 in order to obtain a well-formed expression. Hence, the following two functions are generated.

$$\begin{aligned}\mathsf{plusOrMin}_1\mathsf{First}^0\ m\ n &= \mathsf{First}^{\oplus 2}(\mathsf{Pair\ Plus\ Min})\ m\ n\\ \mathsf{First}^{\oplus 2}\ p\ m\ n &= \mathsf{case}\ p\ \mathsf{of}\ \mathsf{Pair}\ x\ y \rightarrow x\ m\ n\end{aligned}$$

During the next step $\mathsf{First}^{\oplus 2}$ is fused with Pair resulting in a new function $\mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2$ (in which the pattern match has been eliminated), and a replacement of the original pair in $\mathsf{plusOrMin}_1\mathsf{First}^0$

$$\begin{aligned}\mathsf{plusOrMin}_1\mathsf{First}^0\ m\ n &= \mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2\ \mathsf{Plus\ Min}\ m\ n\\ \mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2\ x\ y\ m\ n &= x\ m\ n\end{aligned}$$

During the last two steps $\mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2$ first consumes Plus followed by Min. The Plus will be applied to $m, n$ whereas Min will disappear.

## 5 Basic properties of fusion

In this section we will briefly discuss some basic properties of fusion.

Soundness of fusion can be proved by first defining a semantics for our language, and then by showing that a fusion step of an expression leads to an expression that is semantically equivalent to its original.

As an example we will use a so called *natural operational* (or *big step*) semantics, specifying the result of a computation by means of syntax-driven derivation system. (See also [NN92,AJ02])

**Definition 5.1 (Equivalence).** *Let $E, V$ be expressions, and let $E \Downarrow V$ denote that $E$* evaluates *to $V$ (according to the underlying semantics). We say that two functions $F, F'$ are* semantically equivalent *(notation $F \sim F'$) if for all expressions $\vec{E}$*

$$F\,\vec{E} \Downarrow V \Leftrightarrow F'\,\vec{E} \Downarrow V$$

The following property shows that fusion preserves semantics. It can be used, e.g. for proving fusion is confluent (the order in which expressions are combined is not relevant).

**Property 5.2.** *Let $R = F(\ldots, S\,\vec{(}\ldots), \ldots)$ be a consumer-producer pair, where $S$ is used with arity $k$ at argument position $i$ of $F$. Let $F_iS^k$ be the function obtained when $F$ and $S$ are fused. Then*

$$[\![F_iS^k]\!] \sim F_iS^k,$$

*where the symbol tree and the function definition of $F_iS^k$ are given by definition 4.2 and 4.9 respectively.*

Evaluation by fusion leads to a subset of expressions, so called expressions in *fusion normal form*, or briefly *fusion normal forms*. Also functions bodies are subject to fusion, leading to more or less the same kind of results. These results are characterized by the following syntax.

**Definition 5.3 (Fusion Normal Form).**

- *The set of expressions in* fusion normal form (*FNF*) *is defined as follows.*

$$\begin{aligned} N &::= N' \mid F\,\vec{N'} \mid C\,\vec{N} \\ N' &::= v \mid v \mathbin{@} \vec{N} \end{aligned}$$

- *Function bodies in FNF have the following shape.*

$$\begin{aligned} N_B &::= N \mid \mathsf{case}\ x\ \mathsf{of}\ P_1 \to N_1 \cdots P_k \to N_k \\ P &::= C\,\vec{x} \end{aligned}$$

- *A function is in FNF if its body is, and a collection of functions is in FNF if all functions are.*

**Remark 5.4.** Observe that, in this form, functions are only applied to variables and higher-order applications, and never to constructors or functions.

In [AS04] a relation is established between the typing of an expression and the data constructors it contains after symbolic evaluation: an expression in symbolic normal form does not contain any data constructors that is not included in a typing for that expression. In case of fusion normal forms (*FNF*s), we can derive a similar property, although *FNF*s may still contain function applications. More specifically, let $CE(N)$ denote the collection of data constructors of the (body) expression $N$, and $CT(\sigma)$ denote the data constructors belonging to the type $\sigma$. (For a precise definition of $CE(\cdot)$, $CT(\cdot)$, see [AS04]). Then we have the following property.

**Property 5.5 (Typing FNF).**

- *Let $N$ be an expression in FNF. Suppose $N$ is typable, i.e. for some basis $B$ and type $\sigma$ we have $B \vdash N : \sigma$. Then $CE(N) \subseteq CT(B) \cup CT(\sigma)$.*
- *Let $\mathcal{F}$ be a collection of functions in FNF. Suppose $F \in \mathcal{F}$ has type $\vec{\sigma} \to \tau$. Then $CE(F) \subseteq CT(\vec{\sigma}) \cup CT(\tau)$.*

We can use this property in the following way. Let $\mathcal{F}$ be a set of functions. The first step is to apply fusion to the body of each function $F\,\vec{x} = E_F \in \mathcal{F}$.

Then (standard) evaluation of any application of $F$ will only involve objects using data constructors that are contained in the typing for $F$. More specifically, if $F$ is an instance of a generic function on a user defined data type, then fusion will remove all data constructors of the base types $\{\rightleftarrows, \mathbb{1}, \times, +\}$, provided that neither the generic type of the function nor the instance type itself contains any of these base types.

## 6 Guaranteeing termination

Without any precautions the process of repeatedly eliminating consumer producer pairs might not terminate, or in our setting, will generate an infinite number of new functions.

**Standard fusion**

To avoid non-termination we will not reduce all possible pairs but restrict reduction to pairs in which only *proper* consumers and producers are involved. In [AGS03] a separate analysis phase is used to determine proper consumers and producers. The following definitions are more or less directly taken from [AGS03]

**Definition 6.1 (Active Parameter).** *The notions of* active occurrence *and* active parameter *are defined by simultaneous induction.*

- *We say that a variable $x$ occurs actively in a (body) expression $B$ if there exists a subexpression $E \subseteq B$ such that*
  - $E = \mathsf{case}\ x\ \mathsf{of}\ \ldots$, *or*
  - $E = x\ @\ \ldots$, *or*
  - $E = FD_{1..(i-1)} \star (x) \star D_{(i+1)..k}$, *such that* $act(F)_i$ *and* $\mathsf{arity}(F) = k$.
- *A function $F\,\vec{x} = B_F$ is active in $x_i$ (notation $act(F)_i$) if $x_i$ occurs actively in $B_F$.*
- *Let $F\,\vec{x} = B_F$ be a function, $E \subset B_F$. $E$ is* active (in $F$) *if either $E$ contains variables in which $F$ is active, or (if $F$ is a case function) $E$ contains pattern variables.*

The notion of *accumulating parameter* is used to detect potentially growing recursion.

**Definition 6.2 (Accumulating Parameter).** *Let $F_1, \ldots, F_n$ be a set of mutually recursive functions with respective right-hand sides $B_1, \ldots, B_n$. The function $F_j$ is* accumulating *in its $i^{th}$ parameter (notation $acc(F_j)_i$) if either*

- *there exists a right-hand side $B_k$ such that $F_j\vec{D} \subseteq B_k$,and $D_i$ is active*[1] *in $F_k$ but not just a variable (i.e. $z \subset D_i$ for some active or pattern variable $z$).*
- *there exists a subexpression $F_k\vec{D} \subseteq B_j$ such that $F_k$ is accumulating in $l$, and $D_l = x_i$.*

Observe that the active as well as the accumulating predicate are defined recursively. This will amount to solving a least fixed point equation with respect to the ordering 'false' $\leq$ 'true'.

**Definition 6.3 (Proper Consumer).** *A function $F$ is a* proper consumer *in its $i^{th}$ parameter (notation $con(F)_i$) if $act(F)_i$ and $\neg acc(F)_i$.*

**Definition 6.4 (Proper Producer).** *Let $F_1, \ldots, F_n$ be a set of mutually recursive functions with respective right-hand sides $B_1, \ldots B_n$.*

- *A body $B_k$ is called* unsafe *if it contains a subexpression $G\,\vec{E}$, such that $con(G)_i$ and $E_i = F_j(\cdots)$, for some $G, j$. In words: $B_k$ contains a call to $F_j$ on a consuming position.*

[1] Here we deviate form the original definition as given in [AGS03] or [Chi94] which required that the accumulating expression should be open i.s.o. active.

– *All functions $F_k$ are* proper producers *if none of their right-hand sides is unsafe. Hence, if one of the bodies is unsafe, the complete set becomes improper.*

**Remark 6.5.** It is important to note that non-recursive functions are *always* proper producers.

**Example 6.6.** The well-know function for reversing the elements of a list can be defined in two different ways. In the first definition an auxiliary function rev2 is used.

$$\begin{array}{l} \mathsf{rev}\ l \quad = \mathsf{rev2}\ l\ \mathsf{Nil} \\ \mathsf{rev2}\ l\ a = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \quad\quad\quad \rightarrow a \\ \qquad\qquad\qquad\quad\ \ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{rev2}\ xs\ (\mathsf{Cons}\ x\ a) \end{array}$$

Both rev and rev2 are proper producers. The second definition uses app.

$$\begin{array}{l} \mathsf{rev}\ l = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \quad\quad\quad \rightarrow \mathsf{Nil} \\ \qquad\qquad\qquad\quad\ \ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{app}\ (\mathsf{rev}\ xs)\ (\mathsf{Cons}\ x\ \mathsf{Nil}) \end{array}$$

Now rev is no longer a proper producer: the recursive call to rev appears on a consuming position, since app is consuming in its first argument. Consequently a function like $\mathsf{foo}\ l = \mathsf{len}\ (\mathsf{rev}\ l)$ with

$$\begin{array}{l} \mathsf{len}\ l = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \quad\quad\quad \rightarrow 0 \\ \qquad\qquad\qquad\quad\ \ \mathsf{Cons}\ x\ xs \rightarrow 1 + \mathsf{len}\ xs \end{array}$$

will only be transformed if rev is defined in the first way. By the way, the effect of the transformation w.r.t. the gain in efficiency is almost negligible.

## 7 Improved Consumer Analysis

If functions are not too complex, standard fusion will produce good results. In particular, this also holds for many generic functions. However, in some cases the fusion algorithm fails due to both consumer and producer limitations. We will first examine what can go wrong with the current consumer analysis. For this reason we have adjusted the definition of app slightly.

**Example 7.1.**

$$\begin{array}{l} \mathsf{app}\ l\ t \quad = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \quad\quad\quad \rightarrow t \\ \qquad\qquad\qquad\quad\ \ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{app2}\ (\mathsf{Pair}\ x\ xs)\ t \\ \mathsf{app2}\ p\ t = \mathsf{case}\ p\ \mathsf{of}\ \mathsf{Pair}\ x\ xs \rightarrow \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t) \end{array}$$

Due to the intermediate Pair constructor the function app is no longer a proper consumer. (The (indirect) recursive call has this active pair as an argument and the non-accumulating requirement prohibits this.)

It is hard to imagine that a normal programmer will write such a function directly. However, keep in mind that the optimization algorithm, when applied

to a generic function, introduces many intermediate functions that communicate with each other via basic sum and product constructors. For exactly this reason many relatively simple generic functions cannot be optimized fully.

One might think that a simple inlining mechanism should be capable of removing the Pair constructor. In general, such 'append-like' functions will appear as an intermediate result of the fusion process. Hence, this inlining should be combined with fusion itself which makes it much more problematic. Experiments with very simple inlining show that it is practically impossible to avoid non-termination for the combined algorithm.

To solve the problem illustrated above, we extend fusion with *depth analysis*. Depth analysis is a refinement of the accumulation check (definition 6.2). The original accumulation check is based on a purely syntactic criterion. The improved accumulation check takes into account how the size of the result of a function application increases or decreases with respect to each argument. The idea is to count how many times constructors and destructors (pattern matches) are applied to each argument of a function. If this does not lead to an 'infinite' depth (an infinite depth is obtained if a recursive call extends the argument with one or more constructors) accumulation is still harmless.

**Definition 7.2 (Depth).** *The functions occ and dep are specified below by simultaneous induction.*

$$
\begin{array}{lll}
occ(v, x) & = 0, & \text{if } v = x \\
 & = \bot, & \text{otherwise} \\
occ(v, C\,\vec{E}) & = \max_i(1 + occ(v, E_i)) & \\
occ(v, F\,\vec{E}) & = \max_i(dep(F)_i + occ(v, E_i)) & \\
occ(v, x \mathbin{@} \vec{E}) & = \max(occ(v, x), \max_i(occ(v, E_i))) & \\
occ(v, \mathsf{case}\ x\ \mathsf{of}\ \ldots C_i\vec{y} \to E_i \ldots) & & \\
 & = \max(-\infty, \max_i(\max(occ(v, E_i), & \\
 & \qquad \max_k(occ(y_k, E_i)) - 1))), & \text{if } v = x \\
 & = \max_i(occ(v, E_i)), & \text{otherwise}
\end{array}
$$

*Moreover, for each function* $F\,\vec{x} = B_F$

$$dep(F)_i = occ(x_i, B_F)$$

*using* $\bot + x = \bot$, $\max() = \bot$, *and* $(-\infty) + (+\infty) = +\infty$.

**Remark 7.3.** These two functions are defined as a fixed point equation on $\bot \cup \mathbb{Z} \cup \{+\infty, -\infty\}$, with $\bot \leq -\infty \leq z \leq +\infty$ for all $z \in \mathbb{Z}$. An implementation of this fixed point construction has to limit the domain to a finite subset of $\mathbb{Z}$, extended with $\bot$. The boundaries of this subset can be determined on basis of the structure of the function bodies.

**Example 7.4.** The depths of the two functions appearing in example 7.1 are $dep(\mathsf{app}) = dep(\mathsf{app2}) = (0, +\infty)$.

The following definition gives an improved version of the consuming property (definition 6.3).

**Definition 7.5 (Consuming With Depth Analysis).** *A function* $F\,\vec{x} = B_F$ *is a* proper consumer *in its* $i^{th}$ *argument (notation* $con(F)_i$*) if*

$$act(F)_i \wedge (\neg acc(F)_i \vee dep(F)_i < +\infty).$$

## 8 Improved Producer Analysis

In some cases not the consumer but the producer classification (definition 6.4) is responsible for not getting optimal transformation results. The problem occurs, for instance, when the type of a generic function contains recursive type constructors. Take, for example, the monadic mapping function for the list monad mapl. The base type of mapl is

$$\textbf{type}\ \mathrm{MapL}\ a\ b = a \rightarrow \mathsf{List}\ b$$

Recall that the specialization of mapl to any data type, e.g. Tree, will use the embedding-projection specialized to MapL (see section 2). This embedding projection is based on $\mathsf{ep}_{\mathsf{List}}$: the generic embedding projection specialized to lists. Since List is recursive, $\mathsf{ep}_{\mathsf{List}}$ is recursive as well. Moreover, one can easily show the recursive call to $\mathsf{ep}_{\mathsf{List}}$ appears on a consuming position, and hence $\mathsf{ep}_{\mathsf{List}}$ is not a proper producer. As a consequence, the transformation of a specialized version of mapl gets stuck when it hits on $\mathsf{ep}_{\mathsf{List}}$ appearing as a producer. We illustrate the essence of the problem with a much simpler example based on the data type:

$$\textbf{data}\ \mathsf{Id}\ a = \mathsf{Id}\ a$$

**Example 8.1.**

$$\begin{aligned} \mathsf{unId}\ i &= \mathsf{case}\ i\ \mathsf{of}\ \mathsf{Id}\ x \rightarrow x \\ \mathsf{foo} &= \mathsf{Id}\ (\mathsf{unId}\ \mathsf{foo}) \\ \mathsf{bar} &= \mathsf{unId}\ \mathsf{foo} \end{aligned}$$

Obviously, the function unId is consuming in its argument. Since the recursive call to foo appears as an argument of unId, this function foo is an improper producer. Consequently, the right-hand side of bar cannot be optimized. On the other hand, it seems to be harmless to ignore the producer requirement in this case and to perform a fusion step. As long as we do not evaluate too far termination is not a problem. But how do we prevent of getting into a non-terminating reduction sequence, in case we are dealing with a situation that is less clear?

The solution to this problem is simple: allow improper producers to be unfolded once. But how do we detect whether we have already performed such an unfold step? Actually, this is not as easy as it seems. The transformation algorithm could be parameterized with some kind of *evaluation history* of the improper producers that were unfolded in order to obtain the current expression. However, such a history will make the outcome of the transformation sensible to the evaluation order, which makes reasoning about the transformation much more difficult.

In our transformation algorithm, however, we can use our special tree representation of new function symbols as a substitute for the evaluation history. Remember that a symbol tree contains the information of how the corresponding function was created in terms of the initial set functions and data constructors. Suppose we have a fusion pair consisting of a function $F$ consuming in its $i^{th}$ argument and an improper producer $G$, say with arity $k$. The idea is to detect possible non-termination by examining the symbol tree $F_i G^k$. If this tree contains a cyclic occurrence of some improper producer, we don't fuse; otherwise a fusion step is performed. This leads to the following improved fusion algorithm.

**Definition 8.2 (Improved producer analysis).** *Let $\mathcal{F}$ be a set of functions.*

- *Let $T$ be a symbol tree. Such a tree is called* unsafe *if there exists an improper producer, say with symbol tree $G$, such that $T$ is cyclic in $\ulcorner G \urcorner$. Otherwise the tree is called* safe.
- *Let $F\,\vec{x} = B_F \in \mathcal{F}$. A safe* consumer-producer pair *in $F$ is an expression $R \subseteq B_F$ of the form*

$$R = G\,E_{1..(i-1)} \star S\,\vec{D} \star E_{(i+1)..|\vec{E}|}$$

  *such that $con(G)_i$, and for $S$ one of the following properties holds:*

  - *$S$ is a constructor, or*
  - *$S$ is partially applied function (i.e. $\mathsf{arity}(S) < |\vec{D}|$), or*
  - *$S$ is a proper producer, or*
  - *$S \notin \widehat{F}$ and $F_i S^{|\vec{D}|}$ is safe.*
- *Only safe consumer-producer pairs are fused.*

**Remark 8.3.** We can further improve fusion by replacing unused arguments of functions with $\perp$. More precisely, let $G\,\vec{E}$ be an expression such that $E_i$ is not just a variable. If $dep(G)_i = \perp$ it is safe to replace this expression by

$$G\,E_{1..(i-1)} \star \perp \star E_{(i+1)..|\vec{E}|}$$

**Remark 8.4.** The last requirement in the definition of redex is that the application of an improper producer $S$ occurs outside the component to which $S$ belongs. ($S \notin \widehat{F}$) This requirement is not essential for guaranteeing termination, but it leads to better results for fusing generics.

To illustrate the effect of our refinement we go back to example 8.1. Now, the application in the body of bar is a redex. It will be replaced by $\mathsf{unId}_1\mathsf{foo}^0$, and a new function for this symbol is generated. Following the rules for the introduction of new functions (definition 4.9) the initial body of this function is unId foo, indeed, identical to the expression from which it descended. Again the expression will be recognized as a redex and replaced by $\mathsf{unId}_1\mathsf{foo}^0$, finishing the fusion process.

### Properties

Since we no longer fuse *all* consumer-producers pairs but restrict ourselves to *proper* consumers and producers we cannot expect that the result of a fused expression will always be in *FNF*(as defined in definition 5.3). Consequently, such a result might still contain data constructors that we were trying to eliminate. Assume that initially all functions are consuming in all their arguments, and that all functions appearing on a consuming position are proper producers. Even then it is still not guaranteed that fusion leads to *FNF*. During fusion new functions are introduced which do not necessarily fulfill these requirements or the properties of existing functions might change. Take for instance the function foo from the previous example. An alternative (and equivalent) definition for this function using the Y-combinator $\mathsf{Y}(f) = f\ @\ (\mathsf{Y}(f))$ is $\mathsf{foo} = \mathsf{Y}(\mathsf{Id} \circ \mathsf{unId})$. Now, the example does not contain any improper producers anymore. However, fusing the body of foo will introduce an auxiliary function identical to the original version of foo. And, as we have seen, this function is not a proper producer. Observe that the body of an improper producer is not in *FNF*, even after is has been optimized. Hence, the characterization of fusion normal forms is no longer correct.

We solve this problem by first giving a more liberal classification of the fusion results. Remember that our main concern is not to eliminate all consumer–producer pairs, but only those communicating intermediate objects caused by the structural representation of data types. The new notion of fusion normal forms is based on the types of functions and data constructors.

**Definition 8.5 ($T$-Free Forms).** *Let $T$ be a type constructor.*

- *Let $S$ be a function or data constructor, say with arity $n$, and type $\vec{\sigma} \to \tau$, where $|\vec{\sigma}| = n$. We say that a $k$-ary version of $S$* excludes *$T$, $k \leq n$, (notation $S \not\supseteq_k T$) if*
$$CT(\sigma_{k+1}, \ldots, \sigma_n, \tau) \cap CT(T) = \emptyset$$
*We abbreviate $S \not\supseteq_n T$ to $S \not\supseteq T$.*
- *The set $N_T$ of expressions in $T$-free form is defined as:*
$$N_T ::= N'_T \mid F\ \vec{N'_T} \mid C\ \vec{N_T}$$
$$N'_T ::= v \mid v\ @\ \vec{N_T} \mid S\ \vec{N'_T}$$
*with the additional restriction that for each application of $S\ \vec{N'_T}$ it holds that $S \not\supseteq_{|\vec{N'_T}|} T$.*
- *A function is in $T$-FF if its body is.*

In the next section we show why this new notion of *$T$-FF* is sufficient to obtain the desired result in case of generic functions. This notion enables us to reason about fusion in a more abstract way. For instance, we can now investigate how an improper producer is combined with its surrounding context, and that this combination again will be in the required form.

For functions in *$T$-FF* we have a property comparable to property 5.5 of functions in *FNF*.

**Property 8.6.** *Let $T$ be type constructor, and $\mathcal{F}$ be a collection of functions in $T$-FF. Then, for any $F \in \mathcal{F}$ we have*

$$F \not\supseteq T \;\Rightarrow\; CE(F) \cap CT(T) = \emptyset$$

# 9 Fusion of Generic Instances

In this section we deal with the optimization of instances generated by the generic specializer. An instance is considered to be optimized if the resulting code does not contain constructors belonging to the basic types $\{\rightleftarrows, \mathbb{1}, \times, +\}$. Our goal is to show that under some conditions on the generic base cases, the generic function types, and the instance types the presented fusion algorithm completely removes generic overhead.

Let $g$ be a generic function of type $G$, and let $T$ be a type constructor. Consider the specialization of $g$ to $T$. As mentioned in section 2, a generated instance consists of an adaptor and the code for the structural representation and has the shape

$$g_T \vec{f} = adapt_{\langle G,T\rangle}(g_{T^\circ}\vec{f})$$

The generic constructors that we want to eliminate are EP, Pair, Inl, Inr and Unit. EP can be found in the adaptor only, whereas the other constructors appear in both adaptor and $g_{T^\circ}$.

In practice, optimizing $g_T$ can only be successful if there are no EPs left in the adaptor $adapt_{\langle G,T\rangle}$. Therefore, we start with examining how the adaptor is optimized.

## 9.1 Fusing adaptors

We can split the adaptor in two parts corresponding to $G$ and $T$ respectively. The adaptor has the general shape

$$adapt_{\langle G,T\rangle} = adapt_{\langle G\rangle}\ adapt_{\langle T\rangle}\ \ldots\ adapt_{\langle T\rangle}$$

where $adaptT$ is repeated for each (generic) argument of $G$.

$$\begin{array}{ll} adapt_{\langle G\rangle} & :: (a \rightleftarrows b) \to \ldots \to (a \rightleftarrows b) \to (G\ a \ldots a) \to (G\ b \ldots b) \\ adapt_{\langle G\rangle}\ \vec{x} & = \mathsf{from}\ (\mathsf{ep}_G\ \vec{x}) \\ adapt_{\langle T\rangle} & :: T\ \vec{a} \rightleftarrows T^\circ\ \vec{a} \\ adapt_{\langle T\rangle} & = \mathsf{conv}_T \end{array}$$

Here $\mathsf{ep}_G$ is the specialization of ep to $G$, see section 2. Note that $\rightleftarrows$ is the only basic type appearing in $adapt_{\langle G\rangle}$. This means that if we are able to show that $adapt_{\langle G\rangle}$ is fused to $\rightleftarrows$-*FF* we have eliminated all other basic constructors, because of property 8.6. The instance $\mathsf{ep}_G$ is built from the base cases for the generic function ep, and instances of the form $\mathsf{ep}_P$, where $P$ is a type constructor appearing in $G$. We will first focus on the structure of $\mathsf{ep}_P$ Note that if the type

$P$ is recursive, the generic instance $\mathsf{ep}_P$ will be recursive as well. Since $\mathsf{ep}_P$ is a generic instance, it can be written as

$$\mathsf{ep}_P\ \vec{f} = adapt_{\langle \rightleftarrows, P\rangle}\ (\mathsf{ep}_{P^\circ}\ \vec{f})$$

where the adaptor has the form

$$\begin{aligned} adapt_{\langle \rightleftarrows, P\rangle} &= adapt_{\langle \rightleftarrows\rangle}\ \mathsf{conv}_P\ \mathsf{conv}_P \\ adapt_{\langle \rightleftarrows\rangle}\ a\ b &= \mathsf{from}\ (\mathsf{ep}_{\rightleftarrows}\ a\ b) \end{aligned}$$

It is easy to show that the function $adapt_{\langle \rightleftarrows, P\rangle}$ can be written as

$$adapt_{\langle \rightleftarrows, P\rangle} = \mathsf{EP}(\mathsf{epto}_{\rightleftarrows, P})(\mathsf{epfrom}_{\rightleftarrows, P})$$

where

$$\begin{aligned} \mathsf{epto}_{\rightleftarrows, P}\ e &= \mathsf{mapAR}\ \mathsf{convTo}_P\ \mathsf{convFrom}_P\ (\mathsf{to}\ e) \\ \mathsf{epfrom}_{\rightleftarrows, P}\ e &= \mathsf{mapAR}\ \mathsf{convTo}_P\ \mathsf{convFrom}_P\ (\mathsf{from}\ e) \end{aligned}$$

Fusion of the original $adapt_{\langle \rightleftarrows, P\rangle}$ leads to a more or less similar result.

In this subsection our goal is to show that the resulting code for adaptors is EP free, i.e in $\rightleftarrows$-*FF*. We illustrate how fusion eliminates intermediate EPs by means of examples. The general case can be treated similarly, but is omitted because it does not help the explanation. The adaptor is built from the combination of EP projections (to and from) and EP instances (e.g. $\mathsf{ep}_{\mathsf{List}}$). The first example shows how the to projection of EP is fused with a recursive instance. The second example shows two recursive instances of EP are fused together. And the third example shows how to is fused with the combinations of two recursive instances. This should convince the reader, that the transformation leads to the adaptors that are free from EPs.

**Example 9.1 (Fusing projection with an instance).** We assume that the instance on lists $\mathsf{ep}_{\mathsf{List}}$ is already fused and is in $\{+, \times, \mathbb{1}\}$-*FF*.

$$\begin{aligned} \mathsf{ep}_{\mathsf{List}}\ f &= \mathsf{EP}\ (\mathsf{epto}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f))\ (\mathsf{epfrom}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f)) \\ \mathsf{epto}_{\mathsf{List}}\ f\ r\ l &= \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{Nil} \\ &\qquad\qquad \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ (\mathsf{to}\ f\ h)\ (\mathsf{to}\ r\ t) \\ \mathsf{epfrom}_{\mathsf{List}}\ f\ r\ l &= \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{Nil} \\ &\qquad\qquad \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ (\mathsf{from}\ f\ h)\ (\mathsf{from}\ r\ t) \end{aligned}$$

Consider the application $\mathsf{to}\ (\mathsf{ep}_{\mathsf{List}}\ f)$. Fusion will introduce a function $\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}$ (we indicate new symbols by underlining the corresponding consumer and producer, and also leave out the argument number and the actual arity of the producer). The body of this function is optimized as follows:

$$\begin{aligned} &\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ l \\ &\rightsquigarrow \mathsf{to}\ (\mathsf{EP}\ (\mathsf{epto}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f))\ (\ldots))\ l && \textit{unfolding } \mathsf{ep}_{\mathit{List}} \\ &\rightsquigarrow \mathsf{epto}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f)\ l && \textit{unfolding } \mathsf{to} \\ &\rightsquigarrow \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{Nil} && \textit{unfold } \mathsf{epto}_{\mathit{List}} \\ &\qquad\qquad \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ (\mathsf{to}\ f\ h)\ (\mathsf{to}\ (\mathsf{ep}_{\mathsf{List}}\ f)\ t) \\ &\rightsquigarrow \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{Nil} && \textit{folding } \mathsf{to}, \mathsf{ep}_{\mathit{List}} \\ &\qquad\qquad \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ (\mathsf{to}\ f\ h)\ (\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ t) \end{aligned}$$

It can easily be seen that the only EP involved in the resulting code is the formal argument $f$. This EP is immediately consumed by the to selector.

**Example 9.2 (Fusing two instances).** We assume fusion of the instance for the list and tree types. The instance for tree after fusion is

$$\begin{array}{lll}
\mathsf{ep}_{\mathsf{Tree}}\ f & = \mathsf{EP}\ (\mathsf{epto}_{\mathsf{Tree}}\ f\ (\mathsf{ep}_{\mathsf{Tree}}\ f))\ (\mathsf{epfrom}_{\mathsf{Tree}}\ f\ (\mathsf{ep}_{\mathsf{Tree}}\ f)) \\
\mathsf{epto}_{\mathsf{Tree}}\ f\ r\ t & = \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\mathsf{to}\ f\ x) \\
 & \qquad\qquad\quad \mathsf{Branch}\ x\ y & \rightarrow \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) \\
\mathsf{epfrom}_{\mathsf{Tree}}\ f\ r\ t & = \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\mathsf{from}\ f\ x) \\
 & \qquad\qquad\quad \mathsf{Branch}\ x\ y & \rightarrow \mathsf{Branch}\ (\mathsf{from}\ r\ x)\ (\mathsf{from}\ r\ y)
\end{array}$$

Fusion of $\mathsf{ep}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)$ proceeds as follows.

$$\begin{array}{ll}
\underline{\mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f & \\
\rightsquigarrow \mathsf{EP}\ (\mathsf{epto}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)\ (\mathsf{ep}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)))\ (\ldots)) & \textit{unfolding } \mathsf{ep}_{\mathit{Tree}} \\
\rightsquigarrow \mathsf{EP}\ (\mathsf{epto}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)\ (\underline{\mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f))\ (\ldots)) & \textit{folding } \mathsf{ep}_{\mathit{Tree}}, \mathsf{ep}_{\mathit{List}} \\
\rightsquigarrow \mathsf{EP}\ (\underline{\mathsf{epto}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f\ (\underline{\mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f))\ (\ldots)) & \textit{fusing } \mathsf{epto}_{\mathit{Tree}}, \mathsf{ep}_{\mathit{List}}
\end{array}$$

where fusion of $\mathsf{epto}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)$ proceeds as

$$\begin{array}{lll}
\underline{\mathsf{epto}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f\ r\ l & & \\
\rightsquigarrow \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\mathsf{to}\ (\mathsf{ep}_{\mathsf{Tree}}\ f)\ x) & \textit{unfolding } \mathsf{epto}_{\mathit{Tree}} \\
\quad\ \mathsf{Branch}\ x\ y & \rightarrow \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) & \\
\rightsquigarrow \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ x) & \textit{folding } \mathsf{to}, \mathsf{ep}_{\mathit{List}} \\
\quad\ \mathsf{Branch}\ x\ y & \rightarrow \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) &
\end{array}$$

Fusion has eliminated an intermediate EP produced by $\mathsf{ep}_{\mathsf{List}}$ and consumed by $\mathsf{ep}_{\mathsf{Tree}}$ from the original expression $\mathsf{ep}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)$.

**Example 9.3 (Projection of fused instances).** Fusion of to with $\underline{\mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}$ is similar to the first example. It yields

$$\begin{array}{lll}
\underline{\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}}\ f\ t = \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ x) & \\
\qquad\qquad\qquad\quad \mathsf{Branch}\ l\ r & \rightarrow \mathsf{Branch}\ (\underline{\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}}\ f\ l) & \\
 & \qquad\quad (\underline{\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}}\ f\ r) &
\end{array}$$

These examples illustrate that fusion of adaptors leads to $\rightleftarrows$-*FF*. Provided that the generic and the instance types do not involve EPs, the adaptor is the only part of a generated instance that originally contains EPs. Therefore, fusion transforms instances into $\rightleftarrows$-*FF*.

## 9.2 Requirements

As said before, not all adaptors can be fused to $\rightleftarrows$-*FF*. In fact, when certain type constructors are involved in a generic type, that generic type results in an adaptor that cannot be optimized to $\rightleftarrows$-*FF*. Namely, (1) *nested* and (2) *contravariantly recursive* types lead to such adaptors. We explain both kinds with an example.

– *Nested types* are types like

$$\textbf{data}\ \mathsf{Nest}\ a = \mathsf{NNil} \mid \mathsf{NCons}\ a\ (\mathsf{Nest}\ (a, a))$$

i.e. recursive types in which the arguments of the recursive occurrence(s) are not just variables. The ep that is generated for Nest is

$$\mathsf{ep}_{\mathsf{Nest}}\ a = \mathsf{from}\ (\mathsf{ep}_{\mathsf{EP}}\ \mathsf{conv}_{\mathsf{Nest}}\ \mathsf{conv}_{\mathsf{Nest}})\ (\mathsf{ep}_{+}\ \mathsf{ep}_{\mathbb{1}}\ (\mathsf{ep}_{\times}\ a\ (\mathsf{ep}_{\mathsf{Nest}}(\mathsf{ep}_{(,)}\ a\ a))))$$

This function is accumulating, and hence not a proper consumer. Fusion will therefore not be able to eliminate all EPs in a term like $\mathsf{ep}_{\mathsf{Nest}}\ \mathsf{conv}_T$.

– *Contra-variantly recursive types* are types like

$$\textbf{data}\ \ \mathsf{Contra} = \mathsf{Contra}\ (\mathsf{Contra} \to \mathrm{Int})$$

i.e. recursive types in which one or more of the recursive occurrence(s) appear on a contra-variant position (the first argument of the $\to$-constructor). We will not go into further details to explain why instance of ep for these types are not of the right form.

It is important that these requirements are on type constructors that occur in the generic type, but not on the instance types. We believe that all the above requirements on type constructors are not very restrictive in practice.

Apart from these type constructor requirements, we have the additional restriction that the type $G$ of the generic function itself if free of *self-application*. Self application of types means applying a type constructor to itself, e.g. $\mathsf{List}\ (\mathsf{List}\ a)$. Self application of types will lead to self-application of functions, in particular of embedding projections. The problem is that most eps are improper producers, and hence a nested application of such functions will immediately create a cyclic symbol tree of the corresponding consumer-producer pair. It will therefore not be accepted as a proper redex. Consider, for instance, a generic non-deterministic parser. This parser could have the following type.

$$\textbf{type}\ \ \mathsf{Parser}\ a = (\mathsf{List}\ \mathrm{Char}) \to \mathsf{List}\ (a, \mathsf{List}\ \mathrm{Char})$$

This leads to the following embedding projection

$$\mathsf{ep}_{\mathsf{Parser}}\ a = \mathsf{ep}_{\to}\ (\mathsf{ep}_{\mathsf{List}}\ \mathsf{ep}_{\mathrm{Char}})\ (\mathsf{ep}_{\mathsf{List}}\ (\mathsf{ep}_{(,)}\ a\ (\mathsf{ep}_{\mathsf{List}}\ \mathsf{ep}_{\mathrm{Char}})))$$

After a few steps this function will lead to a self application of $\mathsf{ep}_{\mathsf{List}}$, which obstructs further fusion. This problem can be avoided by choosing different types for different purposes. For instance, the parser's type can be changed into

$$\textbf{type}\ \ \mathsf{Parser}\ a = (\mathsf{List}\ \mathrm{Char}) \to \mathsf{List'}\ (a, \mathsf{List}\ \mathrm{Char})$$

where List' is just another list

$$\textbf{data}\ \ \mathsf{List'}\ a = \mathsf{Nil'} \mid \mathsf{Cons'}\ a\ (\mathsf{List'}\ a)$$

Another useful trick to overcome this problem is to automatically replace closed EP terms like $(\mathsf{ep}_{\mathsf{List}}\ \mathsf{ep}_{\mathrm{Char}})$ with the identity $\mathsf{epid} = \mathsf{EP}\ \mathsf{id}\ \mathsf{id}$. This is possible because mapping for types of kind $\star$ is identity. Then the instance becomes

$$\mathsf{ep}_{\mathsf{Parser}}\ a = \mathsf{ep}_{\to}\ \mathsf{epid}\ (\mathsf{ep}_{\mathsf{List}}\ (\mathsf{ep}_{(,)}\ a\ \mathsf{epid}))$$

### 9.3 Fusing generic instances

So far we have shown that the adaptor is free from EPs. Adaptor is the only part of a generated instance that contains EPs. Now our goal is to show that a generated instance is free from sums, products and units. A generated instance can be written in the following form

$$g_T\vec{f} = adapt_{\langle G,T\rangle}\ (g_{\Sigma\Pi}\ \ldots(g_\tau\ \vec{f})\ \ldots)$$

where $g_{\Sigma\Pi}$ is a combination of the base cases and $g_\tau$-s are free from the base cases and the base types. For instance, consider mapping for the rose trees.

$$\mathsf{map}_{Rose}\ f = adapt_{\langle Map,Rose\rangle}\ (\mathsf{map}_\times\ f\ (\mathsf{map}_{List}\ (\mathsf{map}_{Rose}\ f)))$$

Here $g_{\Sigma\Pi}$ is $\mathsf{map}_\times$ and $g_\tau$-s are $f$ and $\mathsf{map}_{List}\ (\mathsf{map}_{Rose}\ f)$). Sums, products and units appear only in the adaptor and in the $g_{\Sigma\Pi}$ part. They do not appear in the $g_\tau$ part. The type of the expression $adapt_{\langle G,T\rangle}\ (g_{\Sigma\Pi}\ \vec{x})$ does not contain the base types. For instance,

$$\begin{aligned}&\lambda x.\lambda y.adapt_{\langle Map,Rose\rangle}(\mathsf{map}_\times\ x\ y)\\ &\quad :: (a \to b) \to (\mathsf{List}\ (\mathsf{Rose}\ a) \to \mathsf{List}\ (\mathsf{Rose}\ b)) \to (\mathsf{Rose}\ a \to \mathsf{Rose}\ b)\end{aligned}$$

Therefore, it is enough to show that under some conditions fusion of the adaptor with the $g_{\Sigma\Pi}$ part will lead to elimination of the basic constructors, i.e. to $\{+, \times, \mathbb{1}\}$-*FF*. The idea is that the functions used in the base cases should not prevent the basic constructors coming close to the corresponding destructors. Consider, for example, the base case for products of the monadic mapping for lists (section 8).

$$\begin{aligned}&\mathsf{mapl}_\times\ l\ r\ p = \mathsf{case}\ p\ \mathsf{of}\\ &\qquad\qquad \mathsf{Pair}\ x\ y \to l\ x \gg\!= \lambda x'.r\ y \gg\!= \lambda y'.\mathsf{return}\ (\mathsf{Pair}\ x'\ y')\end{aligned}$$

The Pair produced in this instance is consumed in the adaptor. Fusion brings the constructor and the destructor close together, so that they are eliminated. The monadic operations (for lists) that surround the pair constructor do not constitute a problem, because they are "sufficiently consuming and producing". So far, we have not found a way to precisely state what "sufficiently consuming and producing" is. However, all the examples we tried had the base cases amenable for optimization. In practice the restriction is not severe.

Accumulation in the base cases can prevent elimination of the basic constructors. The base cases are essentially accumulating only when the generic function's type refers to a nested type, such as Nest above. However, we have already excluded nested type constructor from generic types in the previous subsection.

## 10 Performance Evaluation

We have implemented the improved fusion algorithm as a source-to-source translator for the core language presented in 3. The input language has been extended

with syntactical constructs for specifying generic functions. Apart from the usual checks for statical semantics, the translator is also able to infer types. We used the Clean compiler [PvE01] to evaluate the performance of the optimized and unoptimized code.

Of course, we have investigated many example programs, but in this section we will only present the result of two examples that are realistic and/or illustrative: simple mapping and non-deterministic parsing. The types of these generic function are:

$$\textbf{type}\ \mathsf{Map}\ a\ b = a \rightarrow b$$
$$\textbf{type}\ \mathsf{Parser}\ a = (\mathsf{List}\ \mathrm{Char}) \rightarrow \mathsf{List'}\ (a, \mathsf{List}\ \mathrm{Char})$$

with List' as defined in section 9.2.

The generic map function was used to apply the increment function to a list of 2.7 $10^8$ integers. We have computed the overhead due to the creation of the list and the evaluation of the applied function and subtracted this from the measured execution times. The language used for the non-deterministic parser was extremely ambiguous leading to more than 200.000 different parses for an input consisting of a list of only 12 characters separated by spaces.

| *program* | *unoptimized (sec)* | *optimized (sec)* | *speedup (times)* |
|---|---|---|---|
| map | 66.78 | 8.42 | 7.9 |
| parser | 45.65 | 0.51 | 89.5 |

The parser example shows a gain in efficiency by a factor of 90. Not mentioned in table is the fact that the optimized version also uses considerably less memory: we had to increase the heap size of the unoptimized version to 128 MB, whereas the optimized version could easily run within in a few MB. The execution time of 45.65 sec can be split up into real execution time (12.0 sec) and garbage collection time (33.65 sec). These figures might appear too optimistic, but other experiments with a complete XML-parser defined generically confirm that these results are certainly not exaggerated.

Finally, for people who want to experiment themselves with the presented optimization technique, the sources of prototype compiler are available and can be obtained by sending an email to one of the authors.

## 11 Related Work

The present work is based on the earlier work [AS04] that used partial evaluation to optimize generic programs. To avoid non-termination we used fix-point abstraction of recursion in generic instances. This algorithm was, therefore, specifically tailored for optimization of generic programs. The algorithm presented here has also been designed with optimization of generic programs in mind. However it is a general-purpose algorithm that can improve other programs. The present algorithm completely removes generic overhead from a considerably larger class of generic programs than [AS04].

The present optimization algorithm is an improvement of fusion algorithm [AGS03], which is in turn based on Chin's fusion [Chi94] and Wadler's deforestation [Wad88]. We have improved both consumer and producer analyses to be more semantically than syntactically based.

Chin and Khoo [CK96] improve the consumer analysis using the *depth* of a variable in a term. In their algorithm, *depth* is only defined for constructor terms, i.e. terms that are only built from variables and constructor applications. This approach is limited to first order functions. Moreover, the functions must be represented in a special *constructor-based* form. In contrast, our *depth* is defined for arbitrary terms of our language. Our algorithm does not require functions in to be represented in a special form, and it can handle higher order functions.

The present paper uses a generic scheme based on type-indexed values [Hin00]. However, we believe that our algorithm will also improve code generated by other generic schemes, e.g PolyP [JJ97].

## 12 Conclusions and Future Work

In this paper we have presented an improved fusion algorithm, in which both producer and consumer analyses have been refined. We have shown how this algorithm *completely* eliminates generic overhead for a large class of programs. This class is described; it covers many practical examples. Presented performance figures show that the optimization leads to a huge improvement in both speed and memory usage.

In this paper we have ignored the aspect of data sharing. Generic specialization does not generate code that involves sharing, although sharing can occur in the base cases provided by the programmer. A general purpose optimization algorithm should take sharing into account to avoid duplication of work and code bloat. In the future we would like to extend the algorithm to take care of sharing. We believe that it will not affect the results for optimization of generic programs.

Additionally, we want to investigate other applications of this algorithm than generic programs. For instance, many programs are written in a combinatorial style using monadic or arrow combinators. Such combinators normally store functions in simple data types, i.e. wrap functions. To actually apply a function they need to unwrap it. It is worth looking at elimination of the overhead of wrapping-unwrapping.

## References

[AGS03] Diederik van Arkel, John van Groningen, and Sjaak Smetsers. Fusion in practice. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 51–67. Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Springer, 2003.

[AJ02] Klaus Aehlig and Felix Joachimski. Operational aspects of normalization by evaluation. In Julian Bradfield, editor, *Proceedings of CSL02*, volume 2471 of *LNCS*, pages 59–73. Springer-Verlag, 2002.

[AS04] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexted Kozen, editor, *Mathematics of Program Construction*, number 3125 in LNCS, pages 16–31, Stirling, Scotland, UK, July 2004. Springer.

[Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: further improvements. *Journal of Functional Programming*, **4**(4):515–555, October 1994.

[CK96] Wei-Ngan Chin and Siau-Cheng Khoo. Better consumers for program specializations. *Journal of Functional and Logic Programming*, 1996(4), November 1996.

[Hin00] Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universität Bonn, October 2000.

[HP01] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.

[JJ97] P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *The 24th ACM Symposium on Principles of Programming Languages, POPL '97*, pages 470–482. ACM Press, 1997.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing, 1992. ISBN 0 471 92980 8.

[PvE01] M.J. Plasmeijer and M. van Eekelen. *Clean Language Report Version 2.0.* University of Nijmegen, The Netherlands, 2001. draft.

[Wad88] Phil Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, number 300 in LNCS, pages 344–358, Berlin, Germany, March 1988. Springer-Verlag.