Improved Fusion for Optimizing Generics

Artem Alimarine and Sjaak Smetsers

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
A.Alimarine@cs.ru.nl, S.Smetsers@cs.ru.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.
Keywords: program transformation, fusion, generic/polytypic programming.

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 [6]. 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 [2] 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 [3, 4]. 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: (1) 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. (2) 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 a subject to the optimization described further in the paper. The generated code is represented in a core functional language defined in section 3. Section 4 defines the semantics of fusion with no termination analysis. Standard fusion with termination analysis [3] is described in section 5. Sections 6 and 7 introduce our extensions to the consumer and the producer analyses. Fusion of generic programs is described in section 8. The performance results are presented in section 9. Section 10 discusses related work. Section 11 reiterates our conclusions.

2 Generics

In this section we give a brief overview of the generic specialization scheme which is based on the approach by Hinze [6]. 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.

$$\textbf{data}\ \mathbb{1} = \mathsf{Unit} \qquad \textbf{data}\ a \times b = \mathsf{Pair}\ a\ b \qquad \textbf{data}\ a + b = \mathsf{Inl}\ a \mid \mathsf{Inr}\ b$$

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, and vise versa.

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.

$$g_T f_1 \cdots f_n = adapt_{\langle G,T\rangle}(g_{T^\circ} f_1 \cdots f_n)$$

where T° denotes the structural representation of T, g_{T° represents the instance of g on T°, and the adapter $adapt_{\langle G,T\rangle}$ takes care of the conversion between T and T°. We will illustrate this generic specialization scheme with a few examples. The following familiar data types are used in the examples.

$$\textbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a) \quad \textbf{data}\ \mathsf{Tree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Branch}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

They have the following structural representation

$$\textbf{type}\ \mathsf{List}^\circ\ a = \mathbb{1} + a \times \mathsf{List}\ a \qquad \textbf{type}\ \mathsf{Tree}^\circ\ a = a + \mathsf{Tree}\ a \times \mathsf{Tree}\ a$$

Note that these representations are *not* recursive: they only capture the outermost structure up to the argument types of each data constructor. 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

$$\begin{array}{l}
\mathsf{convTo_{List}} :: \mathsf{List}\ a \to \mathsf{List}^\circ\ a \\
\mathsf{convTo_{List}}\ l \quad = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Nil} \to \mathsf{Inl\ Unit};\ \ \mathsf{Cons}\ x\ xs \to \mathsf{Inr}\ (\mathsf{Pair}\ x\ xs) \\
\mathsf{convFrom_{List}} :: \mathsf{List}^\circ\ a \to \mathsf{List}\ a \\
\mathsf{convFrom_{List}}\ l = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Inl\ Unit} \to \mathsf{Nil};\ \ \mathsf{Inr}\ (\mathsf{Pair}\ x\ xs) \to \mathsf{Cons}\ x\ xs
\end{array}$$

To define a generic function g the programmer has to provide the *generic type*, say G, and the instances on the base types (the *base cases*). For example, generic mapping is given by the following generic type and base cases

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

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° of T, and by creating an appropriate adapter. For instance, the generated mapping for List° is

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

Note how the structure of $\mathsf{map}_{\mathsf{List}^\circ}$ directly reflects the structure of List°. 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

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

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.

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

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 [7]), which are defined as follows.

$$\mathbf{data}\ a \rightleftarrows b = \mathsf{EP}\ (a \rightarrow b)\ (b \rightarrow a)$$

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° as a basis. To facilitate the instance generation scheme, embedding projections are represented as instances of a single generic function ep, with generic type $a \rightleftarrows b$. The base cases for this (built in) generic function are predefined and consist, besides the usual instances for sum, pair and unit, of instances on $\rightarrow$ and $\rightleftarrows$. 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:

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

Now the adaptor $adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}$ is the from-component (i.e. the second argument of the $\rightleftarrows$ constructor) of this embedding projection applied to $\mathsf{conv}_{\mathsf{List}}$ twice.

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

The adaptor itself is not recursive, but will be invoked each time $\mathsf{map}_{\mathsf{List}}$ is applied to a list element converting the whole list to its structural representation.

To compare the generated version of map with its handwritten counterpart

$$\mathsf{map}\ f\ l = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to \mathsf{Nil};\ \mathsf{Cons}\ x\ xs \to \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

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

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

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 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 capable of removing all the generic overhead.

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

Definition 3.1 (The language).

- *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. The vector $\vec{V}$ *stands for* $(V_1, \ldots, V_n)$
- *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.*
- *The set of* function bodies *is defined as follows.*

$$B ::= E \mid \mathsf{case}\ x\ \mathsf{of}\ C_1\,\vec{x_1} \to E_1 \cdots C_n\,\vec{x_n} \to E_n$$

- *A* function definition *has the form $F\,\vec{x} = B_F$ with $\mathrm{FV}(B_F) \subseteq \vec{x}$. The* arity *of F is $|\vec{x}|$ (i.e. the length of $\vec{x}$).* $\mathrm{FV}(B)$ *stands for the free variables of B.*

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).

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 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 R. More precisely, R contains a subexpression $F\,\vec{E} \star (E_i) \star \vec{E'}$, with $E_i = S\,\vec{D}$. Here $\star$ denotes vector concatenation, and $F\,\vec{E} \star \vec{D}$ should be read as $F\,(\vec{E} \star \vec{D})$; not as $(F\,\vec{E}) \star \vec{D}$. The idea of *fusion* is to replace this pair of two calls in R by a single call to the combined function F_iS resulting in the application $F_iS\,\vec{E} \star \vec{D} \star \vec{E'}$. (Here F_iS stands for the name of the combined function, and should not be confused with, for instance a function F_i applied to S.) In addition, a new function is defined which contains the body of the consumer F in which $S\,\vec{y}$ is substituted for x_i. Note that this fusion mechanism does not require any explicit folding steps.

As an example consider the following definition of app, and the auxiliary function foo.

$$\begin{array}{ll} \mathsf{app}\ l\ t & = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow t;\ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t) \\ \mathsf{foo}\ x\ y\ z & = \mathsf{app}\ (\mathsf{app}\ x\ y)\ z \end{array}$$

The first fusion step leads to the creation of a new function, say $\mathsf{app_1app}$, and replaces the nested applications of app by a single application of this new function. The result is shown below.

$$\begin{array}{ll} \mathsf{foo}\ x\ y\ z & = \mathsf{app_1app} x\ y\ z \\ \mathsf{app_1app}\ x\ y\ z & = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{app}\ y\ z;\ \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ h\ (\mathsf{app}\ (\mathsf{app}\ t\ y)\ z) \end{array}$$

A precise description of how the body of the new function $\mathsf{app_1app}$ is created can be found in [1]. The function foo does not contain consumer-producer pairs anymore; the only pair appears in the body of $\mathsf{app_1app}$, namely $\mathsf{app}\ (\mathsf{app}\ xs\ y)\ z$. Again these nested calls are replaced by $\mathsf{app_1app}$, and since $\mathsf{app_1app}$ has already been created, no further steps are necessary.

$$\mathsf{app_1app}\ x\ y\ z = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{app}\ y\ z;\ \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ h\ (\mathsf{app_1app}\ t\ y\ z)$$

This example shows that we need to determine whether a function - $\mathsf{app_1app}$ 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*. In fact, these symbol trees represent the creation history of the corresponding function in terms of the original set of functions. E.g. in the above example the name $\mathsf{app}_1\mathsf{app}$ expresses that this function was obtained from fusing app with itself. A formal description of symbol trees, as well as of the fusion process itself is given in [1]. Here we restrict ourself to a few remarks on unfolding Unfolding is based on a notion of substitution for expressions. However, due to the restriction on our syntax with respect to cases and to higher-order expressions (recall the selector of a case expression as well as the first part of higher-order expression should be a variable) we cannot use a straightforward definition of substitution. Suppose we try to substitute an expression $D = F\,\vec{D'}$ for x in $(x\ \vec{E})$, which results in the application $F\,\vec{D'} \star \vec{E}$. However, this expression is not well-formed if $\mathsf{arity}(F) < |\vec{D'}| + |\vec{E}|$. To solve this problem we introduce a new function built from the definition of F by supplying it with additional arguments which *increases* its formal arity. In λ-calculus this operation is called *eta-expansion*. Besides, we have to be careful if a substitution is performed on a function body that starts with a pattern match: the result of such a substitution leads to an invalid expression if it substitutes a non-variable expression for the selector. In [1] we show that this problem can be solved by combining consumers and producers is such a way that no illegal (body) expressions are formed.

The following example illustrates how invalid case expressions are prevented. (The definition app is the same as above.)

$$\begin{aligned}&\mathsf{len}\ l &&= \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to 0;\ \ \mathsf{Cons}\ x\ xs \to \mathsf{Inc}\ (\mathsf{len}\ xs)\\ &\mathsf{foo}\ x\ y &&= \mathsf{len}\ (\mathsf{app}\ x\ y)\end{aligned}$$

The only consumer-producer pair occurs in foo. It will lead to the creation of a new function $\mathsf{len}_1\mathsf{app}$. Without any precautions, the body of this new function becomes

$$\begin{aligned}\mathsf{len}_1\mathsf{app}\ x\ y = \mathsf{case}\ &(\mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to t;\ \ \mathsf{Cons}\ x\ xs \to \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t))\ \mathsf{of}\\ &\mathsf{Nil} \to 0;\quad \mathsf{Cons}\ x\ xs \to \mathsf{Inc}\ (\mathsf{len}\ xs)\end{aligned}$$

which clearly violates the syntax for expressions. A correct body is obtained by pushing the outermost case into the alternatives of innermost one. In the Cons branch this leads to the elimination of the pattern match; in the Nil branch the len-function is re-introduced, leading to

$$\mathsf{len}_1\mathsf{app}\ x\ y = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Nil} \to \mathsf{len}\ y;\ \ \ \mathsf{Cons}\ z\ zs \to \mathsf{Inc}\ (\mathsf{len}\ (\mathsf{app}\ zs\ y))$$

Now the only fusion candidate appears in this newly created function, namely $\mathsf{len}\ (\mathsf{app}\ zs\ y)$. It can directly be replaced by a recursive call to $\mathsf{len}_1\mathsf{app}$, which eliminates the last consumer–producer pair.

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. We can characterize these results by the following syntax.

Definition 4.1 (Fusion Normal Form). *The set of expressions in* fusion normal form *(FNF) is defined as follows.*

$$N ::= N' \mid F\,\vec{N'} \mid C\,\vec{N} \qquad N' ::= v \mid v\ \vec{N}$$

Observe that, in this form, functions are only applied to variables and higher-order applications, and never to constructors or functions. Our aim is show that fusion eliminates all basic data constructors. In [2] this is done by establishing a relation between the typing of an expression and the data constructors it contains after (symbolic, i.e. compile-time) 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. Note that this is more complicated than for symbolic evaluation because *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 σ. (For a precise definition of $CE(\cdot)$, $CT(\cdot)$, see [2]). Then we have the following property.

Property 4.2 (Typing FNF). *Let $\mathcal{F}$ be a collection of functions in FNF. Suppose $F \in \mathcal{F}$ has type $\vec{\sigma} \to \tau$. Then $CE(F) \subseteq CT(\vec{\sigma}) \cup CT(\tau)$. In words: the data constructors possibly generated by F belong to the set of types indicated by the typing for F.*

To reach out goal, we can use this property in the following way. Let $\mathcal{F}$ be a set of functions. By applying fusion to the body of each function $F \in \mathcal{F}$ we eliminate the data constructors of all data types that do not appear in the typing for F In particular, 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.

5 Standard fusion

Without any precautions the process of repeatedly eliminating consumer producer pairs might not terminate 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 [3] a separate analysis phase is used to determine proper consumers and producers. The following definitions are more or less directly taken from [3].

Definition 5.1 (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 B contains a subexpression E such that*
  - $E = \mathsf{case}\ x\ \mathsf{of}\ \ldots$, *or*
  - $E = x\ \ \ldots$, *or*
  - $E = F\,\vec{D}$, *such that $D_i = x$ and act(F)$_i$.*

  By $\mathrm{AV}(E)$ *we denote the set of active variables occurring in E.*
- *Let $F\,\vec{x} = B_F$ be a function. F is active in x_i (notation act(F)$_i$) if $x_i \in \mathrm{AV}(B_F)$.*

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

Definition 5.2 (Accumulating Parameter). *Let $F_1 = B_1, \ldots, F_n = B_n$ be a set of mutually recursive functions. The function $F = F_j$ is* accumulating *in its i^{th} parameter (notation $acc(F)_i$) if either*

– *there exists a right-hand side B_k containing a subexpression $F\vec{D}$ such that D_i is open but not just a variable.*
– *B_j itself contains a subexpression $F_k\vec{D}$ 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.

Definition 5.3 (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 5.4 (Proper Producer). *Let $F_1 = B_1, \ldots, F_n = B_n$ be a set of mutually recursive functions.*

– *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.*
– *All functions F_k are* proper producers *if none of their right-hand sides is unsafe.*

Consider, for example, the function for reversing the elements of a list. It can be defined in two different ways. In the first definition ($\mathsf{rev1}$) an auxiliary function revacc is used; the second definition ($\mathsf{rev2}$) uses app.

$$\begin{array}{ll}\mathsf{rev1}\ l & = \mathsf{revacc}\ l\ \mathsf{Nil} \\ \mathsf{revacc}\ l\ a & = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Nil} \rightarrow a;\ \ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{revacc}\ xs\ (\mathsf{Cons}\ x\ a) \\ \mathsf{rev2}\ l & = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{Nil};\ \ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{app}\ (\mathsf{rev2}\ xs)\ (\mathsf{Cons}\ x\ \mathsf{Nil})\end{array}$$

Both $\mathsf{rev1}$ and revacc are proper producers; $\mathsf{rev2}$ however is not: since app is consuming in its first argument, the recursive occurrence of $\mathsf{rev2}$ is on a consuming position. Consequently a function like $\mathsf{foo}\ l = \mathsf{len}\ (\mathsf{rev1}\ l)$ will be transformed, whereas $\mathsf{bar}\ l = \mathsf{len}\ (\mathsf{rev2}\ l)$ will remain untouched. By the way, the effect of the transformation w.r.t. the gain in efficiency is negligible.

6 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.

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

Due to the intermediate Pair constructor the function app is no longer a proper consumer. (The (indirect) recursive call has this pair as an argument and the non-accumulating requirement prohibits this.) It is hard to imagine that a normal programmer will ever 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 5.2). 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 of constructors is harmless.

Definition 6.1 (Depth). *The functions occ and dep are specified below by simultaneous induction (again leading to a fixed point construction).*

$$\begin{array}{lll}
occ(v,x) & = 0, & \textit{if } v = x\\
 & = \perp, & \textit{otherwise}\\
occ(v, C\,\vec{E}) & = \max_i(1 + occ(v,E_i)) & \\
occ(v, F\,\vec{E}) & = \max_i(dep(F)_i + occ(v,E_i)) & \\
occ(v, x\ \vec{E}) & = \max(occ(v,x), \max_i(occ(v,E_i))) & \\
\multicolumn{3}{l}{occ(v, \mathsf{case}\ x\ \mathsf{of}\ \ldots C_i\vec{y} \rightarrow E_i \ldots)}\\
 & = \max(-\infty, \max_i(\max(occ(v,E_i)), & \\
 & \qquad\qquad \max_k(occ(y_k,E_i)) - 1))), & \textit{if } v = x\\
 & = \max_i(occ(v,E_i)), & \textit{otherwise}\\
dep(F)_i & = occ(x_i, B_F), & \textit{where } F\,\vec{x} = B_F
\end{array}$$

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

The depths of app and app2 above are $dep(\mathsf{app}) = dep(\mathsf{app2}) = (0, +\infty)$.

The following definition gives an improved version of the accumulation property (definition 5.2).

Definition 6.2 (Accumulating With Depth). *Let $F_1 = B_1, \dots, F_n = B_n$ be a set of mutually recursive functions. The function $F = F_j$ is* accumulating *in its i^{th} parameter (notation $acc(F)_i$) if either*

1. *$dep(F)_i = +\infty$, or*
2. *there exist a right-hand side B_k containing a subexpression $F\vec{D}$ such that $\mathrm{AV}(D_i) \neq \emptyset$, and D_i is not just a variable, or*
3. *B_j has a subexpression $F_k\vec{D}$ such that F_k is accumulating in l, and $D_l = x_i$.*

We illustrate this definition with an example in which a function ident is defined that collects all subsequent characters from an input list that obey a predicate p.

$$\begin{array}{lll}\mathsf{ident}\ p\ i\ r = \mathsf{case}\ \ i\ \ \mathsf{of} & \mathsf{Nil} \rightarrow r \\ & \mathsf{Cons}\ h\ t \rightarrow \mathsf{if}\ (p\ h)\ (\mathsf{ident}\ p\ t\ (\mathsf{app}\ r\ (\mathsf{Cons}\ h\ \mathsf{Nil})))\ r\end{array}$$

This function is accumulating in its third parameter because of requirement (2). Note that the depth of ident in r is 0. Indicating this argument as accumulating will prevent infinite fusion of ident, e.g. in the body of ident itself.

7 Improved Producer Analysis

In some cases the producer classification (definition 5.4) 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 $a \rightarrow \mathsf{List}\ b$. 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:

$$\begin{array}{ll}\mathbf{data}\ \mathsf{Id}\ a = \mathsf{Id}\ a & \\ \mathsf{unId}\ i & = \mathsf{case}\ i\ \mathsf{of}\ \mathsf{Id}\ x \rightarrow x \\ \mathsf{foo} & = \mathsf{Id}\ (\mathsf{unId}\ \mathsf{foo}) \\ \mathsf{bar} & = \mathsf{unId}\ \mathsf{foo}\end{array}$$

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. To avoid non-termination in general, we use our special tree representation of new function symbols. Remember that these symbol trees contain 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 proper consumer and an *improper* producer. The combined symbol tree is used as follows: if it contains a path on which an improper producer appears twice, we don't fuse; otherwise a fusion step is performed. In essence this producer check avoids repetitive unfolding of the same improper producer. For a full formalization of this idea we refer to [1].

To illustrate the effect of our refinement we go back to the example. Now, the application in the body of bar is a redex (i.e. a fusion candidate). It will be replaced by $\mathsf{unId}_1\mathsf{foo}$, and a new function for this symbol is generated with unId foo as initial body. Indeed, this expression is identical to the expression from which it descended. Again the expression will be recognized as a redex and replaced by $\mathsf{unId}_1\mathsf{foo}$, finishing the fusion process.

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 4.1). To show that such a result does not contain any unwanted data constructors, we first give 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 7.1 (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:*

$$N_T ::= N'_T \mid F\,\vec{N'_T} \mid C\,\vec{N_T} \qquad N'_T ::= v \mid v\ \vec{N_T} \mid S\,\vec{N'_T}$$

with the restriction that for each application of $S\,\vec{N'_T}$ it holds that $S \not\supseteq_{|\vec{N'_T}|} T$.

For functions in *T-FF* we have a property similar to property 4.2 of functions in *FNF.*

Property 7.2. *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 $F \not\supseteq T \;\Rightarrow\; CE(F) \cap CT(T) = \emptyset$.*

8 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 illustrate that under certain conditions on the generic base cases, the generic function types, and the instance types the presented fusion algorithm completely removes generic overhead. According to property 7.2 this boils down to showing that fusion leads to $\{\rightleftarrows, \mathbb{1}, \times, +\}$-*FF*.

As mentioned in section 2, an instance of a generic function consists of an adaptor and the code for the structural representation. Treating a generic function completely would require several pages of example code. For this reason we restrict ourselves to the adapter part of a generic function and illustrate how the EPs are eliminated. The first example shows how the to projection of EP is fused with a recursive instance of ep for the List type. We assume that the instance on lists $\mathsf{ep}_{\mathsf{List}}$ is already fused and is in $\{+, \times, \mathbb{1}\}$-*FF*.

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

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:

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

Obviously, the resulting code is in $\rightleftarrows$-*FF*, and due to property 7.2 this function will not generate any EP-constructor.

The next example shows how the recursive instances of ep for lists and trees are fused together. The instance for tree after fusion is

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

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

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

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

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\underline{\mathsf{epto}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f\ r\ l} \\
\rightsquigarrow \mathsf{case}\ t\ \mathsf{of} & \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\mathsf{to}\ (\mathsf{ep}_{\mathsf{Tree}}\ f)\ x) & & \textit{unfolding}\ \mathsf{epto}_{\mathsf{Tree}} \\
 & \mathsf{Branch}\ x\ y & \rightarrow \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) & & \\
\rightsquigarrow \mathsf{case}\ t\ \mathsf{of} & \mathsf{Leaf}\ x & \rightarrow \mathsf{Leaf}\ (\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ x) & & \textit{folding}\ \mathsf{to}, \mathsf{ep}_{\mathsf{List}} \\
 & \mathsf{Branch}\ x\ y & \rightarrow \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) & &
\end{array}
$$

Fusion has eliminated an intermediate EP produced by $\mathsf{ep}_{\mathsf{List}}$ and consumed by $\mathsf{ep}_{\mathsf{Tree}}$ from the original expression $\mathsf{ep}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)$. This has led to a function whose structure is similar to the structure of $\mathsf{ep}_{\mathsf{List}}$ or $\mathsf{ep}_{\mathsf{Tree}}$. Therefor fusing the projection to with this function will yield a $\rightleftarrows$-*FF*.

It appears that not all adaptors can be fused to $\rightleftarrows$-*FF*. In particular, this occurs when the generic type contains type constructors with a *nested* or with a *contra-variant* definition. Nested types are types like

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

i.e. recursive types in which the arguments of the recursive occurrence(s) are not just variables. These nested type definitions give rise to accumulating eps, and hence to improper consumers (see section 6). Contra-variantly recursive types are types like

$$\mathbf{data}\ \ \mathsf{Contra} = \mathsf{Contra}\ (\mathsf{Contra} \rightarrow \mathrm{Int})$$

i.e. recursive types in which a recursive occurrence appears on a contra-variant position (the first argument of the $\rightarrow$-constructor). For these contra-variant type definitions the improved producer check will fail (see section 7), obstructing further fusion.

Apart from these type constructor requirements, we have the additional restriction that the type of the generic function is free of *self-application*, e.g. $\mathsf{List}\ (\mathsf{List}\ a)$. For self application again the producer check will fail. As an example, consider a generic non-deterministic parser with type.

$$\mathbf{type}\ \ \mathsf{Parser}\ a = (\mathsf{List}\ \mathrm{Char}) \rightarrow \mathsf{List}\ (a, \mathsf{List}\ \mathrm{Char})$$

Here, the nested List application will hamper complete fusion. This problem can be avoided by choosing different types for different results: The input stream, of type $\mathsf{List}\ \mathrm{Char}$, and the resulting parses, of type $\mathsf{List}\ (a, \mathsf{List}\ \mathrm{Char})$ could alternatively be represented by using an auxiliary list type List'.

9 Performance Evaluation

We have implemented the improved fusion algorithm as a source-to-source translator for the core language presented in section 3. The input language has been extended with syntactical constructs for specifying generic functions. These generic functions are translated into ordinary Clean functions, optimized as well as unoptimized. Then we used the Clean compiler to evaluate the performance of the optimized and unoptimized code. We have investigated many example programs, but in this section we will only present the result of examples that are realistic and/or illustrative: Simple mapping (map), monadic mapping (mapl, mapr, for the List and Rose monad respectively), generic reduce (reduce) (used to implement folding), and non-deterministic parsing (parser). We only mention execution times; heap usage is not shown. The latter because in all example programs memory consumption is improved with approximately the same factor as execution time. Moreover, all optimized programs are more or less as efficient as their handwritten counterparts. Finally, to illustrate the difference between successful and partial (unsuccessful) fusion we include the performance results of a monadic mapping function (mapn) with a nested monad type.

program	*unoptimized (sec)*	*optimized (sec)*	*speedup (times)*
map	72.9	13.4	7.9
mapl	29.8	1.5	19.9
mapr	304.5	10.9	82.3
reduce	37.9	3.7	10.2
parser	45.65	0.51	89.5
mapn	168.4	164	1.03

These figures might appear too optimistic, but other experiments with a complete generic XML-parser confirm that these results are not exaggerated.

10 Related Work

The present work is based on the earlier work [2] 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 [2].

The present optimization algorithm is an improvement of fusion algorithm [3], which is in turn based on Chin's fusion [4] and Wadler's deforestation [9]. We have improved both consumer and producer analyses to be more semantically than syntactically based. Chin and Khoo [5] 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 [6]. However, we believe that our algorithm will also improve code generated by other generic schemes, e.g PolyP [8].

11 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 eliminates generic overhead for a large class of programs. This class is described; it covers many practical examples. 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. 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

1. Artem Alimarine and Sjaak Smetsers. Fusing generic functions. Technical report NIII-R0434, University of Nijmegen, Sep 2004.
2. 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.
3. 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.
4. Wei-Ngan Chin. Safe fusion of functional expressions II: further improvements. *Journal of Functional Programming*, **4**(4):515–555, October 1994.
5. Wei-Ngan Chin and Siau-Cheng Khoo. Better consumers for program specializations. *Journal of Functional and Logic Programming*, 1996(4), November 1996.
6. Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universität Bonn, October 2000.
7. 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.
8. 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.
9. 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.