J. Functional Programming 4 (4): 515–555, October 1994

Safe fusion of functional expressions II: Further improvements†

WEI-NGAN CHIN

Department of Information Systems and Computer Science,
National University of Singapore,
Singapore 0511 (Email: *chinwn@iscs.nus.sg*)

Abstract

Large functional programs are often constructed by decomposing each big task into smaller tasks which can be performed by simpler functions. This hierarchical style of developing programs has been found to improve programmers' productivity because smaller functions are easier to construct and reuse. However, programs written in this way tend to be less efficient. Unnecessary intermediate data structures may be created. More function invocations may be required.

To reduce such performance penalties, Phil Wadler proposed a transformation algorithm, called *deforestation*, which could automatically *fuse* certain composed expressions together to eliminate intermediate tree-like data structures. However, his technique is currently safe (terminates with no loss of efficiency) for only a subset of first-order expressions.

This paper will generalise the deforestation technique to make it safe for all first-order and higher-order functional programs. Our generalisation is explained using a model for *safe fusion* which views each function as a producer and its parameters as consumers. Through this model, syntactic program properties are proposed to classify producers and consumers as either safe or unsafe. This classification is used to identify sub-terms that can be safely fused/eliminated. We present the generalised transformation algorithm, illustrate it with examples and provide a termination proof for the transformation algorithm of first-order programs. This paper also contains a suite of additional techniques to further improve the basic safe fusion method. These improvements could be viewed as enhancements to compensate for some inadequacies of the syntactic analyses used.

Capsule Review

Wadler's *deforestation* algorithm from 1988 eliminates intermediate data structures from functional programs but is only guaranteed to terminate for *treeless* terms. Wadler's *blazed* deforestation (also from 1988) does the same as ordinary deforestation, but essentially leaves *annotated* subterms untransformed.

Improving and extending work in his PhD thesis from 1990, Chin gives in the present paper the *extended* deforestation algorithm, which is an improved version of the blazed deforestation algorithm, and a technique for finding annotations. The main theorem of the paper states that

† An earlier version of this work appeared at the *7th ACM Conference on Lisp and Functional Programming*, June 92, San Francisco.

extended deforestation applied to an arbitrary term annotated by this technique terminates. This means that deforestation can be safely applied to all first order terms.

The core in the technique for computing annotations is to annotate non-treeless subterms; Chin casts this in a producer-consumer terminology. The technique also applies a bottom-up strategy. Moreover, many improvements over the basic technique are suggested. Higher-order functions are also considered.

The techniques are syntactic and can easily be implemented in a fully automatic optimizing compiler.

1 Introduction

Consider an expression $p(q(v_1), r(v_2, s(v_3)))$, call it e, where v_1, v_2, v_3 are variables and p, q, r, s are user-defined functions. In this expression, v_1, v_2, v_3 are inputs of e, while p, q, r, s are simpler functions decomposed from the main task of e. This expression may be viewed as a modular construction from simpler reusable functions p, q, r, s. However, sub-terms of e, like $q(v_1)$ or $r(v_2, s(v_3))$ or $s(v_3)$, may be a source of large intermediate data structures that are expensive to construct, but may be garbage-collected later because they are not directly referred in the final result. This is a source of inefficiency. A possible remedy is to apply unfold/fold transformation (Burstall & Darlington, 1977) to fuse e into a piece of more tightly woven code, without its unnecessary intermediate sub-terms.

For example, if all the sub-terms of e could be safely fused, a new function f_1 could be defined (to represent e) and transformed until the original nesting of function calls disappears, as shown below (we shall use the Hope language for our programs where equations are of the form $---LHS \Leftarrow RHS$):

$---f_1(v_1, v_2, v_3) \Leftarrow p(q(v_1), r(v_2, s(v_3)))$;

transforms to

$\Leftarrow$ *..equivalent expression without the original nested function calls..*

However, not all sub-terms can be safely fused with their containing expression. If we can identify those sub-terms which are unsuitable for fusion, a simple known technique called *parameter generalisation*, can be used to abstract away the unsuitable sub-terms before fusion. For example, if the sub-term $r(v_2, s(v_3))$ cannot be fused with p of e, then a new function, f_2, can be defined with the unsuitable sub-term replaced by a new parameter variable, w. This can then be transformed, as outlined below:

$---f_2(v_1, w) \Leftarrow p(q(v_1), w)$;

transforms to

$\Leftarrow$ *..equivalent expression without the above nested functions..*

With the above function, the expression e is now equivalent to $f_2(v_1, r(v_2, s(v_3)))$, where further opportunities for fusion may be found by similar analysis and transformation of $r(v_2, s(v_3))$ itself. Thus, fusion can be selectively applied, as long as sub-terms which are unsafe to fuse can be identified.

To give a more concrete example, consider the following function *intseq*. (Note: a

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

data statement defines the algebraic *list* data type and **dec** statements specify the type of functions. Also, we use tuple type of the form *(A,B)* instead of *(A # B)* found in Hope):

```
data list(A) == nil ++ cons(A,list(A));
dec intseq: (int,int) → list(int);
dec take: (list(A),int) → list(A);
dec infint: int → list(int);
---intseq(s,d)          ⇐ take(infint(s),d);
---take(nil,n)          ⇐ nil;
---take(cons(a,as),n)   ⇐ if n=0 then nil else cons(a,take(as,n-1));
---infint(n)           ⇐ cons(n,infint(n+1));
```

This function generates a finite sequence of consecutive integers, *s,s+1,..,s+d-1*, by evaluating a nested expression *take(infint(s),d)*. When this function is evaluated, an intermediate data structure would be built by the inner function *infint* but would later be garbage-collected because it is not used in the final result.

To remove this unnecessary intermediate data, we could apply the unfold/fold transformation rules to the RHS of *intseq*. There are six elementary but powerful rules which could be used, namely *define*, *unfold*, *fold*, *instantiate*, *where abstraction* and *laws* (see Burstall & Darlington, 1977). However, our basic fusion method will make use of only the following three rules:

Define Introduce a new equation (function definition) with a unique LHS.

Unfold Replace a call by its corresponding function body, with the appropriate parameter substitutions. The unfold rule will sometimes incorporate the instantiate step when dealing with function calls with pattern-matching parameters.

Fold Replace an expression which matches a function body by its corresponding function call.

For the above example, we begin with the earlier definition of *intseq* which is already suitably generalised; so there is no need for a separate define step:

```
---intseq(s,d)          ⇐ take(infint(s),d);
```

We then unfold the inner *infint* call, followed by an unfold on the outer *take* call, before a fold back on *intseq*, as follows:

```
unfold infint call
---intseq(s,d)          ⇐ take(cons(s,infint(s+1)),d);
unfold take call
---intseq(s,d)          ⇐ if d=0 then nil else cons(s,take(infint(s+1),d-1));
fold with intseq call
---intseq(s,d)          ⇐ if d=0 then nil else cons(s,intseq(s+1,d-1));
```

The final transformed function of *infseq* is now without the unnecessary intermediate data sub-term.

This paper presents an automatic method to perform such fusion transformations. It grew out of Chapters 3 and 4 of the author's PhD thesis (Chin, 1990), and has been inspired primarily from Phil Wadler's (1988) work on deforestation. An early version of this work has appeared in Chin (1992a). This paper is both an expansion

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

and consolidation of the previous paper with full (termination) proof and more examples. It also contains a suite of new improvement techniques to further enhance the fusion method.

An overview of this paper follows. In section 2, we briefly describe the pure and blazed deforestation algorithms of Wadler. In section 3, we present a producer-consumer model of functions and propose a new annotation scheme based on safe/unsafe producers and consumers. (An earlier annotation scheme, proposed by Chin (1990, 1991), is based solely on consumers. This earlier scheme works by changing all unsafe producers to pseudo-safe. However, it hinders a further improvement on deforestation which uses laws in addition to equations.) Section 4 presents an extended deforestation algorithm for first-order programs, together with formal definitions for safe/unsafe producers and consumers. Section 5 presents a full termination proof for the first-order transformation algorithm. Section 6 outlines our extension of deforestation to a full higher-order functional language. Section 7 shows how some potential problems, which arise from the syntactic analyses used are overcome. Section 8 provides a number of new improvements to the basic fusion method. These improvements help to overcome some weaknesses of the syntactic analyses for the producers and consumers. Section 9 compares with other related work. Section 10 concludes. Throughout this paper, we will use both the terms, *deforestation* and *fusion*, interchangeable to mean the transformation technique for eliminating safe intermediate sub-terms from purely functional programs.

2 Wadler's deforestation

The deforestation technique was first proposed by Wadler (1988) as an automatic transformation algorithm for eliminating unnecessary intermediate data terms from a *sub-set* of first-order expressions. Both a simple version, called *pure deforestation*, and its enhanced version, called *blazed deforestation*, were proposed.

The first-order language used contains functions of the form:

$$---f(v_1,\ldots,v_n) \Leftarrow tf;$$

with its RHS term, tf, described by:

$$t ::= v \mid c(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) \mid \mathit{case}\ t\ \mathit{of}\ \{p_1 \rightarrow t_1;..;p_n \rightarrow t_n\}$$
$$p ::= c(v_1,\ldots,v_j)$$

Each term t can be made up of either a variable, a constructor term, a function call or a *case* expression. The above grammar is actually for a restricted first-order language because only *simple* patterns of the form, $p = c(v_1,\ldots,v_j)$, are allowed in the *case* construct. However, there is no loss in generality because translation methods exist (Augustsson, 1985; Wadler, 1987) to translate any expression with complex nested patterns (in *case* constructs) to an equivalent expression of the above restricted form.

In a reformulation of pure deforestation, a slightly different language was adopted in Ferguson & Wadler (1988), where each *case* construct is replaced by an equivalent g-type function. A f-type function (shown earlier) is a function which is defined *without* any pattern-matching parameter in its equation. In contrast, a g-type function

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

has exactly one pattern-matching parameter (the first one) and is defined using one or more equations, as shown below.

$$\begin{array}{lll} \text{---}\, g(p_1, v_1, \ldots, v_n) & \Leftarrow & tg_1; \\ \quad\vdots & & \quad\vdots \\ \text{---}\, g(p_r, v_1, \ldots, v_n) & \Leftarrow & tg_r; \end{array}$$

This new language (with both f-type and g-type functions) helps to simplify the deforestation algorithm. Its use also results in smaller transformed programs. We shall adopt this simple but complete first-order language to describe both Wadler's work and our extension.

Pure deforestation is a transformation algorithm, formulated using define, unfold and fold rules. It is applicable to all expressions which are composed *solely* from a special type of functions, called *pure treeless* functions, as defined below.

Definition 1: Pure treeless form
An expression is said to be *pure treeless* if it satisfies the grammar form below.

$$tt ::= v \mid c(tt_1, \ldots, tt_n) \mid f(v_1, \ldots, v_n) \mid g(v_0, v_1, \ldots, v_n)$$

where f and g are *pure treeless* functions and each variable, v, occurs only once in the expression.

Correspondingly, a function is said to be *pure treeless* if each of its RHS term(s) is *pure treeless.*

Terms of the above grammar form are known as pure treeless terms because they do not contain nested applications of functions. Notice that *all* arguments of function calls in the pure treeless form are variables. As a result, pure treeless expressions do not construct intermediate data structures (including tree-like ones) when they are building their final results. An example of pure treeless function is:

```
dec append: (list(A),list(A)) → list(A);
---append(nil,ys)          ⇐ ys;
---append(cons(x,xs),ys) ⇐ cons(x,append(xs,ys));
```

The pure deforestation algorithm can transform any expression, which uses only pure treeless functions, to an equivalent expression that is pure treeless. For example, the expression *append(append(xs,ys),zs)* uses only pure treeless functions. It can be transformed by the deforestation algorithm to a pure treeless expression, *apptree(xs,ys,zs)*, as shown in Fig. 1.

Blazed deforestation is an extension of pure deforestation to cater for functions which are not pure treeless because of atomic-type sub-terms. Atomic-type sub-terms are those terms with simple types (like integer, char) which require small storage. Compared to tree-like sub-terms, they do not result in much gain when eliminated.

Two examples of functions which are not pure treeless because of atomic-type sub-terms are:

```
dec double: list(int) → list(int);
dec sum: list(int) → int;
---double(nil)          ⇐ nil;
---double(cons(a,as)) ⇐ cons(2*a,double(as));
---sum(nil)            ⇐ 0;
---sum(cons(a,as))     ⇐ a+sum(as);
```

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Initial expression to transform:
append(append(xs,ys),zs)
Define new function *appthree* **and fold:**
⇒ *appthree(xs,ys,zs)*
New function *appthree*:
---appthree(xs,ys,zs) ⇐ *append(append(xs,ys),zs)*
Unfold inner *append* **call**
Equation from the *nil* **case:**
---appthree(nil,ys,zs) ⇐ *append(ys,zs)*
Equation from the *cons(x,xs)* **case:**
---appthree(cons(x,xs),ys,zs) ⇐ *append(cons(x,append(xs,ys)),zs)*
unfold outer *append* **call:**
⇐ *cons(x,append(append(xs,ys),zs))*
fold with *appthree* **function:**
⇐ *cons(x,appthree(xs,ys,zs))*

Fig. 1. Transformation steps applied by pure deforestation.

The sub-expressions which do not conform to pure treeless form are shown underlined. Blazed deforestation handles such functions by using an annotation scheme which marks each atomic-type sub-term with ⊖, and each tree-type sub-term with ⊕. Periodically, before each fusion sequence, all sub-terms annotated as ⊖ are abstracted using the *let* constructs to prevent them from being fused. As a consequence, atomic-type sub-terms are allowed to be nested, and their variables be non-linear. This type-based blazing scheme is simple to implement but it cannot be used to extend deforestation to all first-order expressions. This is because this scheme does not guarantee deforestation algorithm's termination (for example, some tree-like sub-terms cannot be safely fused).

In the next section, we propose a new model for safe fusion that can identify more accurately sub-terms which can be fused, from those which cannot. This model will be used to generalise deforestation to all first-order and higher-order programs.

3 Producer-consumer model

For the purpose of determining where fusion is possible in an expression, we propose the use of a model which views each function as both a **producer** of data through its *result*, and a **consumer** of data through its *parameter*. Presently, each of the *f*- or *g*-type function takes a number of parameters and returns a single result. Each parameter of these functions will be viewed as a consumer. Similarly, the single result (represented by the RHS of the function definition) will be viewed as a producer.

3.1 Conditions for safe fusion

Consider a nested application of two function calls: *p(q(x))*. In this nested application, the sub-term *q(x)* is used to produce an intermediate data which is to be consumed by the sole parameter of *p*. An important question to raise is under what conditions can this nested application be *safely* and *effectively* fused. We distinguish between

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

safe and *effective* fusion †. A nested application is said to be *safely* fused if the transformation sequence terminates and there is no loss of efficiency experienced by the transformed expression when compared to the original expression. A nested application is *effectively* fused if the need for its intermediate data disappears and there is a gain in efficiency. In this paper, efficiency refers to the number of reduction steps and heap storage needed to evaluate a given expression.

Ideally, we would like to know exactly when safe and effective fusion can take place. However, we shall present a more modest result outlining *sufficient* criteria which conservatively determine when safe fusion is possible. If the safe fusion also happens to be effective, then a gain in efficiency will result.

In our proposed model, we will classify producers and consumers as either *safe* or *unsafe* with respect to their amenability to safe fusion. The static properties which can determine whether a given producer or consumer is safe (or unsafe) will be given in section 4.

For the moment, we propose that any expression, *p(q(x))*, can be *safely fused* if:

(i) *q(x)* is a *safe* producer, and

(ii) the parameter of *p* is a *safe* consumer.

Conversely, if either *p* is an unsafe consumer and/or *q* is an unsafe producer, then fusion *may* fail. Using the above model, Wadler's pure treeless functions can be viewed as functions which are both safe producers and safe consumers, with any expression composed from them being totally fusable. Also, blazed deforestation treats all atomic-type sub-terms as unsafe, and allows only safe tree-like sub-terms to be used. Hence, it cannot handle expressions which contain unsafe tree-like sub-terms. Our model for safe fusion can cater to a more general deforestation algorithm. For example, it can tolerate *q* as an unsafe consumer and/or *p* as an unsafe producer and yet permit *p(q(x))* to be still fusable.

3.2 Double annotation scheme for safe fusion

With the need to take into account the fusability properties of producers and consumers, we propose a *double annotation* scheme to help identify sub-terms that could be fused. We use the same basic annotation symbols of blazed deforestation, $\oplus$ and $\ominus$, but augment them with appropriate subscripts. In this scheme, each sub-term is either marked as a $\ominus_p$ if it is an unsafe producer, or a $\oplus_p$ if it is not an unsafe producer. In addition, the same sub-term may inherit a $\oplus_c$ annotation if it presently lies within a safe consumer, or a $\ominus_c$ annotation if it lies within an unsafe consumer.

Producer-based annotation is considered a *location-independent* property of the sub-terms. It can be formally contained in the following annotated grammar:

$$
\begin{array}{lll}
t & ::= & v^{\oplus_p} \mid c(t_1,\ldots,t_n)^{\oplus_p} \mid safeP \mid unsafeP \\
safeP & ::= & g(t_0,\ldots,t_n)^{\oplus_p} \mid f(t_1,\ldots,t_n)^{\oplus_p} \\
 & & \text{where } f, g \text{ are } safe \text{ producers} \\
unsafeP & ::= & g(t_0,\ldots,t_n)^{\ominus_p} \mid f(t_1,\ldots,t_n)^{\ominus_p}
\end{array}
$$

† During a visit to Glasgow in October 1991, Phil Wadler helpfully pointed out to the author the need to differentiate between the notions of *safe*, as opposed to, *real/effective* fusion.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

where f, g are *unsafe* producers

Consumer-based annotation is a *location-dependent* property of the sub-terms. This is because the annotations are dependent on the parameter positions that the sub-terms are located. Sub-terms which lie in the safe parameters (consumers) of function calls will be annotated with a $\oplus_c$; while those which lie in unsafe parameters will be annotated with a $\ominus_c$. The rest of the sub-terms, for example arguments of constructors, which do *not* lie in the parameters of f- or g-type function calls, need not be provided with any consumer-based annotations.

The criteria for classifying consumers and producers as either safe or unsafe are given later in Section 4.3.1 and 4.3.2, respectively.

With this scheme, each sub-term is annotated either once or twice. Sub-terms which are arguments of function calls are annotated twice, while the rest of the sub-terms are annotated only once. For fusion purpose, we are primarily concerned with sub-terms which are also arguments of function calls. For these sub-terms, we mark each of them with either a $\oplus$ if it is safe to eliminate, or a $\ominus$ if it is not safe to eliminate; according to the following double annotation scheme:

$\oplus_p$ combines with $\oplus_c$ to give $\oplus$
$\oplus_p$ combines with $\ominus_c$ to give $\ominus$
$\ominus_p$ combines with $\oplus_c$ to give $\ominus$
$\ominus_p$ combines with $\ominus_c$ to give $\ominus$

In the next section, we present appropriate syntactic criteria for this double annotation scheme, together with the transformation algorithm for first-order programs.

4 Fusion of first-order programs

This section describes how safe fusion can be achieved for all first-order functional programs. We present an extended result of Wadler's deforestation theorem that is able to fuse safe sub-terms and skip over the unsafe ones. This is followed by a corresponding annotation scheme for consumers/producers to achieve the above result. Finally, we show how the extended deforestation algorithm could be effectively applied to each first-order function by a bottom-up procedure.

4.1 Extended deforestation

Wadler's main theorem on pure deforestation states that any expression which is composed from pure treeless functions could always be transformed to an equivalent expression that is pure treeless. In this section, we provide an extension to Wadler's theorem which could be used to handle all first-order functional expressions. Our result uses a more general form of treelessness, called *extended-treeless* form, which is formally defined below.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Definition 2: Extended-treeless form

An expression is said to be *extended-treeless* (or *e-treeless*) if it satisfies the grammar below:

$$et \quad ::= v \mid c(et_1,\ldots,et_n) \mid f(arg_1,\ldots,arg_n) \mid g(arg_0,\ldots,arg_n)$$
$$arg \ ::= v^{\oplus} \mid et^{\ominus}$$

where only variables appear in arguments annotated as $\oplus$; and f, g are *e-treeless* functions.

Correspondingly, a function is said to be *e-treeless*, if each of its definition's RHS term(s) is *e-treeless*.

The *e-treeless* form allows sub-terms which are unsafe to fuse (marked as $\ominus$) to remain in its expressions. Also, sub-terms which are safe to fuse (marked as $\oplus$) are assumed to be transformed away because only variables are allowed in these sub-terms. This e-treeless form could be used to cover all first-order functions‡, as we shall see in section 4.4.

To illustrate the e-treeless form, consider the following two functions whose RHS are e-treeless. Note that because primitive functions (like *) cannot be unfolded, we have to annotate their arguments as unsafe.

$$\begin{aligned}
&---rev_it(nil,w) && \Leftarrow w;\\
&---rev_it(cons(a,as),w) && \Leftarrow rev_it(as^{\oplus},cons(a,w)^{\ominus});\\
&---double(nil) && \Leftarrow nil;\\
&---double(cons(a,as)) && \Leftarrow cons(2^{\ominus} *a^{\ominus},double(as^{\oplus}));
\end{aligned}$$

Our main result, called the *extended deforestation theorem*, can be stated as follows:

Theorem 3: Extended deforestation

Every expression that is made up of only data constructors and e-treeless functions can be safely transformed (without loss of efficiency and non-termination) to an equivalent e-treeless expression by the extended deforestation algorithm.

The extended deforestation algorithm is presented in the next sub-section. It will attempt to remove all safe sub-terms by fusion transformation and leave the unsafe sub-terms untouched via parameter generalisation. Simple induction proofs on the algorithm can show that there is no increase in reduction steps for the transformed expression and that only e-treeless expressions result from the algorithm. These trivial proofs are skipped in this paper.

The termination proof for the transformation algorithm (under the conditions of the above theorem) is not trivial, and is presented in detail in section 5. In fact, our termination proof will cover a more general result that allows the extended deforestation algorithm to transform any expression that is composed solely from data constructors and *e-treeless with constant* functions to an equivalent *e-treeless* expression. The *e-treeless with constant* expression form is similar to the *e-treeless* form, except for also allowing constant functions in the safe parameter positions. Its formal definition is given below.

‡ A trivial approach to make every first-order function conform to the e-treeless form is by annotating the RHS so that all variables are marked as safe and the rest as unsafe. Thanks to Morten H Sorensen for pointing this out. We use a slightly more sophisticated scheme.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Definition 4: Extended treeless with constant form
An expression is said to be in the *e-treeless with constant* form if it satisfies the grammar below:

$$et \quad ::= v \mid c(et_1, \ldots, et_n) \mid f(arg_1, \ldots, arg_n) \mid g(arg_0, \ldots, arg_n)$$
$$arg \ ::= v^{\oplus} \mid f()^{\oplus} \mid et^{\ominus}$$

where only variables and constants (represented by $f()$) appear in arguments annotate as $\oplus$; and f, g are *e-treeless with constant* functions.

Correspondingly, a function is said to be *e-treeless with constant*, if each of its definition's RHS term(s) is *e-treeless with constant*.

4.2 Transformation algorithm

The extended deforestation algorithm is made up of six syntax-directed rules given in Fig. 2. The first rule, dealing with a variable, has nothing to fuse. The second rule, dealing with a constructor as the outermost term, has to skip over the constructor because it is not able to use the constructor as a consumer.

The next four rules deal with expressions that may have a nesting of function calls that could be fused. We use a context notation, $\cdots C \cdots$ (similar to that used by Ferguson & Wadler, 1988), to help identify an inner call, C, that is about to be unfolded by normal-order reduction strategy. This inner call lies within a nesting of $\oplus$ pattern-matching arguments of g-type calls, e.g. $g_1(g_2(\cdots g_n(C^{\oplus}, ..)\cdots^{\oplus}, ..)^{\oplus}, ..)$ where $n \geq 0$. A formal grammar specification for this context notation is given below:

$$\cdots C \cdots ::= C \mid g(\cdots C \cdots^{\oplus}, t_1, \ldots, t_n)$$

Four rules ($\mathcal{T}3, 4, 5, 6$) are used to transform expressions of the form $\cdots C \cdots$. Depending on appropriate conditions, these rules use one of four different transformation steps, in the order of preference shown below:

1. *direct unfold* ($\mathcal{T}3a, 4a$).
2. *skip over* ($\mathcal{T}4d, 5c, 6c$).
3. *fold* ($\mathcal{T}3b, 4b, 5a, 6a$).
4. *define followed by an unfold* ($\mathcal{T}3c, 4c, 5b, 6b$).

A *direct unfold* is taken if the inner call is either $g(c_i(t'_1, \ldots, t'_j)^{\oplus}, t_1, \ldots, t_n)$ or $f(t_1, \ldots, t_n)$ where f is non-recursive and only variables appear in the unsafe (and non-linear) consumers. With only variables in the unsafe arguments, there is no possibility of large arguments being duplicated or accumulated by direct unfolding. This step is always taken in preference to the other steps (including the *define & unfold* step) because it helps to obtain better transformed code. The above conditions guarantee that there will be no infinite direct unfolding (see termination proof).

A *skip over* is taken if the outermost function call contains no safe sub-terms to fuse.

A *fold* is taken if a previously defined function matches the current expression. All previously defined functions are stored in a function definition set, called *def_set*.

If none of these situations are met, then a *define & unfold* step is taken. In this

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

(1) $\mathscr{T}[v] \Rightarrow v$

(2) $\mathscr{T}[c(t_1,\ldots,t_n)] \Rightarrow c(\mathscr{T}[t_1],\ldots,\mathscr{T}[t_n])$

(3) $\mathscr{T}[\cdots g(c_i(t'_1,\ldots,t'_j)^{\oplus},t_1,\ldots,t_n)\cdots]$

a) IF $(\forall a \in 1..n.\, t_a^{\ominus} \rightarrow t_a \equiv v)$ *(direct unfold)*
$\Rightarrow \mathscr{T}[\cdots tg_i[t'_1/v'_1,\ldots,t'_j/v'_j,t_1/v_1,\ldots,t_n/v_n]\cdots]$

b) IF $f\_old(vn_1,\ldots,vn_k,ve_1,\ldots,ve_s) \Leftarrow \cdots g(c_i(t'^{\triangle}_1,\ldots,t'^{\triangle}_j),t_1^{\triangle},\ldots,t_n^{\triangle})\cdots^{\triangle}$
$\in$ *def_set* *(fold)*
$\Rightarrow f\_old(vo_1,\ldots,vo_k,\mathscr{T}[te_1],\ldots,\mathscr{T}[te_s])$

WHERE $(\cdots g(c_i(t'^{\triangle}_1,\ldots,t'^{\triangle}_j),t_1^{\triangle},\ldots,t_n^{\triangle})\cdots^{\triangle},\ te_1\ldots te_s,ve_1\ldots ve_s$
$(\ ,\ vo_1\ldots vo_k,vn_1\ldots vn_k) = \mathscr{G}[\cdots g(c_i(t'_1,\ldots,t'_j),t_1,\ldots,t_n)\cdots]$

c) OTHERWISE *(define & unfold)*
$\Rightarrow f\_new(vo_1,\ldots,vo_k,\mathscr{T}[te_1],\ldots,\mathscr{T}[te_s])$

DEFINE & ADD TO *def_set*
$f\_new(vn_1,\ldots,vn_k,ve_1,\ldots,ve_s)$
$\Leftarrow \cdots g(c_i(t'^{\triangle}_1,\ldots,t'^{\triangle}_j),t_1^{\triangle},\ldots,t_n^{\triangle})\cdots^{\triangle}$

UNFOLD $\Leftarrow \mathscr{T}[\cdots tg_i[t'^{\triangle}_1/v'_1,\ldots,t'^{\triangle}_j/v'_j,t_1^{\triangle}/v_1,\ldots,t_n^{\triangle}/v_n]\cdots^{\triangle}]$

WHERE $(\cdots g(c_i(t'^{\triangle}_1,\ldots,t'^{\triangle}_j),t_1^{\triangle},\ldots,t_n^{\triangle})\cdots^{\triangle},\ te_1\ldots te_s,ve_1\ldots ve_s$
$,\ vo_1\ldots vo_k,vn_1\ldots vn_k) = \mathscr{G}[\cdots g(c_i(t'_1,\ldots,t'_j),t_1,\ldots,t_n)\cdots]$

(4) $\mathscr{T}[\cdots f(t_1,\ldots,t_n)\cdots]$

a) IF *non-recursive*$(f) \wedge (\forall a \in 1..n.\, t_a^{\ominus} \rightarrow t_a \equiv v)$ *(direct unfold)*
$\Rightarrow \mathscr{T}[\cdots tf[t_1/v_1,\ldots,t_n/v_n]\cdots]$

b) SIMILAR TO *(fold)* OF $\mathscr{T}3b$

c) SIMILAR TO *(define & unfold)* OF $\mathscr{T}3c$

d) IF $\cdots f(t_1,\ldots,t_n)\cdots \equiv f(t_1,\ldots,t_n) \wedge \forall a \in 1..n.\, t_a^{\oplus} \equiv v$ *(skip over)*
$\Rightarrow f(\mathscr{T}[t_1],\ldots,\mathscr{T}[t_n])$

(5) $\mathscr{T}[\cdots g(v_0^{\oplus},t_1,\ldots,t_n)\cdots]$

a) SIMILAR TO *(fold)* of $\mathscr{T}3b$

b) SIMILAR TO *(define & unfold)* OF $\mathscr{T}3c$

c) SIMILAR TO *(skip over)* OF $\mathscr{T}4d$

(6) $\mathscr{T}[\cdots g(t_0^{\ominus},t_1,\ldots,t_n)\cdots]$

a) SIMILAR TO *(fold)* OF $\mathscr{T}3b$

b) SIMILAR TO *(define & unfold)* OF $\mathscr{T}3c$

c) SIMILAR TO *(skip over)* OF $\mathscr{T}4d$

where

$\mathscr{G}[t^{\ominus}] \Leftarrow$ (nv,[t],[nv],[],[]) ! nv is a new variable

$\mathscr{G}[v] \Leftarrow$ (nv,[],[],[v],[nv]) ! nv is a new variable

$\mathscr{G}[c(t_1,\ldots,t_n)] \Leftarrow (c(t_1^{\triangle},\ldots,t_n^{\triangle}),te\_lt,ve\_lt,vo\_lt,vn\_lt)$
where $(t_1^{\triangle}\ldots t_n^{\triangle},te\_lt,ve\_lt,vo\_lt,vn\_lt) = \mathscr{GL}[t_1\ldots t_n]$

$\mathscr{G}[f(t_1,\ldots,t_n)] \Leftarrow (f(t_1^{\triangle},\ldots,t_n^{\triangle}),te\_lt,ve\_lt,vo\_lt,vn\_lt)$
where $(t_1^{\triangle}\ldots t_n^{\triangle},te\_lt,ve\_lt,vo\_lt,vn\_lt) = \mathscr{GL}[t_1\ldots t_n]$

$\mathscr{G}[g(t_0,\ldots,t_n)] \Leftarrow (g(t_0^{\triangle},\ldots,t_n^{\triangle}),te\_lt,ve\_lt,vo\_lt,vn\_lt)$
where $(t_0^{\triangle}\ldots t_n^{\triangle},te\_lt,ve\_lt,vo\_lt,vn\_lt) = \mathscr{GL}[t_0\ldots t_n]$

$\mathscr{GL}[t_1\ldots t_n] \Leftarrow (t_1^{\triangle}\ldots t_n^{\triangle},te\_lt_1\ldots te\_lt_n,ve\_lt_1\ldots ve\_lt_n,vo\_lt_1\ldots vo\_lt_n$
$,vn\_lt_1\ldots vn\_lt_n)$
where $\forall i \in 1..n\ (t_i^{\triangle},te\_lt_i,ve\_lt_i,vo\_lt_i,vn\_lt_i) = \mathscr{G}[t_i]$

Fig. 2. Extended deforestation algorithm for first-order expressions.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

```
T [rev_it(double(as),double(w)) ]                                  ; T 5b define revdb
                    ⇒ revdb(as,T [double(w) ])                     ; T 5c skip over
                    ⇒ revdb(as,double(w))
Define a new function, revdb
---revdb(as,ws)     ⇐ rev_it(double(as),ws)                        ; unfold double(as)
---revdb(nil,ws)    ⇐ T [rev_it(nil,ws) ]                          ; T 3a unfold rev_it
                    ⇐ T [ws ]                                      ; T 1 skip
                    ⇐ ws
---revdb(cons(a,as),ws) ⇐ T [rev_it(cons(2*a,double(as)),ws) ]      ; T 3a unfold rev_it
                    ⇐ T [rev_it(double(as),cons(2*a,ws)) ]         ; T 5a fold revdb
                    ⇐ revdb(T [as ],T [cons(2*a,ws) ])             ; T 1,T 2 skip
                    ⇐ revdb(as,cons(T [2*a ],T [ws ]))             ; T 4d,T 1 skip
                    ⇐ revdb(as,cons(T [2 ]*T [a ],ws))             ; T 2,T 1 skip
                    ⇐ revdb(as,cons(2*a,ws))
```

Fig. 3. Application of generalised deforestation algorithm $\mathcal{T}$

step, the expression to be transformed is first *generalised* by replacing all arguments of unsafe consumers (unsafe sub-terms) with new variables. In addition, all the other variables (not extracted via generalisation) are *renamed*. Renaming helps to faciliate folding. Without it, variables which occur more than once must be properly synchronised by the fusion method when a fold is required. As an example, consider the fusion of the RHS of function *syn* below:

```
---syn(xs)                    ⇐ zip(double(xs),second(xs));
---zip(cons(x,xs),cons(y,ys)) ⇐ cons((x,y),zip(xs,ys));
---zip(xs,ys)                 ⇐ nil;
---second(nil)                ⇐ nil;
---second(cons(a,as))         ⇐ second'(as);
---second'(nil)               ⇐ nil;
---second'(cons(a,as))        ⇐ cons(a,second(as));
```

If the identical variables of *zip(double(xs),second(xs))* are not renamed, then fusion could fail because the two occurrences of *xs* could not be synchronised by the transformation. Renaming the variables at each *define & unfold* step helps to avoid this problem. It allows the intermediate sub-terms to be eliminated, without the need to synchronise the duplicated variables.

The procedure for generalisation and renaming, called $\mathcal{G}$, will ensure that (i) no unsafe sub-terms are fused, and (ii) the expression is linear. This procedure helps ensure transformation algorithm's termination. Given an expression, t, the $\mathcal{G}$ procedure will return a tuple of five items, namely:

$$(t^{\triangle}, te_1 \ldots te_s, ve_1 \ldots ve_s, vo_1 \ldots vo_k, vn_1 \ldots vn_k)$$

where $t^{\triangle}$ is a notation to represent the generalised (and renamed) expression of t; $te_1 \ldots te_s$ is a list of unsafe sub-terms extracted from t; $ve_1 \ldots ve_s$ are the new variables in $t^{\triangle}$ to replace $te_1 \ldots te_s$; $vo_1 \ldots vo_k$ is a list of variable occurrences in t that do not belong to unsafe sub-terms, $vn_1 \ldots vn_k$ are the new unique variables in $t^{\triangle}$ to replace $vo_1 \ldots vo_k$.

As an illustration of the new transformation algorithm, consider the expression

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

rev_it(double(as)$^{\oplus}$,double(w)$^{\ominus}$) which uses only e-treeless functions. This expression can be transformed by the new algorithm, as shown in Fig. 3.

An important characteristic of this transformation algorithm is that it preserves the lazy semantics of its transformed programs. In Runciman *et al.*(1989), it was shown that the instantiation step (on *g*-type functions) can alter the strictness behaviour under certain circumstances. To avoid this, each variable to be instantiated must lie in a strict location before it is safe to instantiate. Our algorithm chooses only *g*-type calls, which lie in a nesting of outer *g*-type function calls, for instantiation. This chosen call is always in a strict location relative to the context of the expression. As a result, our transformation algorithm preserves the non-strict behaviour of its transformed programs.

4.3 Consumer/producer annotations

The extended deforestation theorem can be related in a natural way to the producer/consumer model proposed earlier. This relation is used to help explain the safe fusion method and its further improvements later. In particular, the syntactic criteria used for classifying the consumers and producers as either *safe* or *unsafe* will be used to help *functions/expressions conform to the e-treeless (with constant) form.*

Our annotation scheme is *simple* to compute, *safe* to use, and *extensible* to further enhancements. The initial annotation scheme is *monovariant*, in the sense that every call to the same function definition will have consistently annotated parameters (consumers) and results (producers) throughout the entire program. In section 8, we shall also consider a *polyvariant* extension.

4.3.1 Handling consumers

We define the *non-accumulating* parameter criterion here.

> *Definition 5*: Non-accumulating parameter criterion
> Given a set of *mutually recursive* functions, $h_1, \ldots, h_k$, where $k \geq 1$. The jth parameter, v_j, of the ith function, h_i, with definitional equations of the form:
>
> $$---h_i(p_1, \ldots, p_j, \ldots, p_n) \Leftarrow th_i\,;$$
> *where* $p ::= v \mid c(v_1, \ldots, v_m)$
>
> is considered to be *non-accumulating* if each recursive call of the form $h_i(t_1, \ldots, t_j, \ldots, t_n)$ in the RHS of functions, $h_1, \ldots, h_k$, has the jth-argument, t_j, as a *variable* or a *constant.*
>
> Also, all parameters of *non-recursive* functions are trivially regarded as non-accumulating, since there are *no* recursive calls in their definitions.

To illustrate the non-accumulating parameter criterion, consider a recursive function, h, with a single recursive call in its RHS:

$$---h(c(v_1), v_2, v_3, v_4, v_5, v_6) \Leftarrow ..h(v_1, v_2, v_4, v_3, const, acc(v_6))..$$

This function has six parameters. The first parameter is a constructor pattern argument which gets smaller§ with each recursive call. The second parameter is

§ This criterion of structurally smaller arguments for the successive recursive calls is also

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

unchanged across recursion. The third and fourth parameters are swapped around with each recursive call. The fifth parameter becomes a constant with subsequent recursive calls. These five parameters are *non-accumulating* because their sizes are bounded across recursion. The last parameter is accumulating because each successive recursive call can accumulate up a sub-expression of the form $acc(\_)$.

Another useful parameter criterion is linearity, as defined below.

Definition 6: Linear parameter criterion
A parameter of a function is *linear* if its variable(s) occurs only once in each RHS term of its function.

Based on the above two parameter criteria, we could now define a safe consumer as follows:

Definition 7: Safe/unsafe consumers
A parameter of a function definition is classified as a *safe consumer* if it is *linear* and *non-accumulating*; otherwise it is classified as an *unsafe consumer.*

The linearity criterion is used to help avoid loss of efficiency by not duplicating large non-linear arguments (to avoid redundant evaluation) during deforestation. This duplication risk of non-linear arguments is already a well considered issue by the partial evaluation community (Sestoft, 1988).

Non-accumulating parameters are considered to be safe parameters (for fusion) because their arguments will not grow in size during transformation. In contrast, accumulating parameters may result in successively larger expressions during transformation and cause non-terminating transformation.

As an example, consider the fusion of $h(v_1, v_2, v_3, v_4, v_5, double(v_6))$ where the last parameter is assumed to be a list of numbers. The first unfold on h will instantiate $v_1 = c(v_{1_1})$ and result in $h(v_{1_1}, v_2, v_4, v_3, const, acc(double(v_6)))$. A further unfold on h will instantiate $v_{1_1} = c(v_{1_2})$ and result in an even larger expression $h(v_{1_2}, v_2, v_3, v_4, const, acc(acc(double(v_6))))$ that is to be further transformed. Notice that the last argument experiences an increasing size. This process to fuse the last argument of h will go on forever without any success because it is an accumulating parameter.

Another way of viewing the non-accumulating parameter criterion is that it helps to ensure that all arguments of the current recursive functions conform to the *e-treeless with constant* form.

4.3.2 Handling producers

We formally define safe/unsafe producers as follows:

known as the *inductive parameter* criterion. It was used in Sestoft (1988) to ensure that infinite unfolding of calls with one or more such known arguments cannot happen. The reason is that known inductive arguments will get smaller with each unfolding and this cannot happen forever with finitely-sized structures. The inductive parameter is a special case of the non-accumulating criterion.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Definition 8: Safe/unsafe producers
A set of mutually recursive functions is classified as *safe producer(s)* if none of its recursive calls (in the RHS of their definitions) are currently lying in safe parameter positions; otherwise it is classified as *unsafe producer(s)*.

An example of safe producer is the function, *double*. Its single recursive call is lying within a constructor term and not in a safe argument position. When it is regarded as a safe producer, its recursive call satisfies the e-treeless form. Not all functions are safe producers. An example is the function, *rev_flatten*:

dec *rev_flatten: list(list(A)) → list(A);*
---rev_flatten(nil) ⇐ *nil;*
---rev_flatten(cons(as,ass)) ⇐ *append(rev_flatten(ass)*$^{\oplus c}$*,as*$^{\oplus c}$*);*

This function is not considered as a safe producer because its recursive call, *rev_flatten(ass)*, is presently lying in a safe consumer of the *append* function. This is a violation of e-treeless grammar because only variables or unsafe producers may appear in the positions of safe consumers. However, if *rev_flatten* is regarded as an unsafe producer, then this violation disappears. Note that this happens only after *rev_flatten* has been regarded as an unsafe producer.

As illustrated above, this definition of safe/unsafe producer is meant to help all functions conform to the e-treeless form, so that they may be safely handled by the extended deforestation algorithm. One way to handle unsafe producers is *not* to unfold them, as producers, during fusion. Unsafe producers can be seen as *producing* function calls, in addition to constructor terms. For example, the *rev_flatten* function can be viewed as a recursive function which produces *append* function calls during unfolding. These generated *append* calls cannot be readily consumed by most outer functions. Using pattern-matching equations, we could guarantee that constructors are safely consumed but not the produced functions. We refer to functions produced by unsafe producers (like *append* calls from *rev_flatten*) as *obstructing function calls* because they may obstruct safe fusion. In particular, obstructing function calls may result in successively larger expressions when attempts are made to fuse their producers with safe consumers.

To illustrate this phenomenon, consider an expression *length(rev_flatten(ass))* which contains an unsafe producer within a safe consumer. An unfold on the unsafe producer will use the instantiation $ass=cons(as_1, ass_1)$ before resulting in a new expression $length(append(rev\_flatten(ass_1),as_1))$. This expression can be further unfolded using the instantiation, $ass_1=cons(as_2, ass_2)$, to result in an even larger expression $length(append(append(rev\_flatten(ass_2),as_2),as_1))$. This fusion involving *rev_flatten* as a producer will not terminate because the obstructing *append* calls cannot be consumed by the equation of the outer *length* function. As a result, the expression to be transformed gets successively larger.

In our transformation algorithm, all unsafe producers in safe and unsafe consumers are generalised out prior to each fusion sequence. This conservative strategy can avoid non-termination by preventing all unsafe producers from being unfolded, as producers. (A slightly less conservative strategy is to generalise out only those unsafe producers which are about to be unfolded as producers. To support this, a small change to the double annotation scheme is required whereby a term is

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

annotated as unsafe if it lies in an unsafe consumer or it is an unsafe producer that is about to be unfolded. However, we shall not consider this strategy here because it may render an improvement technique in section 8 inapplicable.)

Even though unsafe producers should not be unfolded as producers, there is nothing to stop them from being unfolded as safe consumers. For example, the parameter of *rev_flatten* is a safe consumer because it is linear and non-accumulating. In an expression like *rev_flatten(append(xs,ys))* where *append(xs,ys)* is a safe sub-term, our algorithm is allowed to unfold *rev_flatten*. This treatment helps to maximise the opportunities for safe fusion.

4.4 First-order functions

Presently, the transformation algorithm is formulated to transform expressions which are composed from e-treeless (with constant) functions. However, our real aim is to use it to transform all first-order functions. To do that, the transformation algorithm can be applied to the RHS terms of each first-order function. The procedure for converting each function to e-treeless (with constant) form is as follows.

Procedure 9: Fusion on first-order programs

1. Apply the transformation algorithm according to the *bottom-up order*, so that functions lower in the calling hierarchy are transformed before those above them.
2. Transform each set of mutually recursive functions, say $f_i..f_j$, simultaneously and regard the recursive calls of $f_i..f_j$ as potentially unsafe producers and consumers.
3. After transformation, analyse the set of recursive functions to provide appropriate consumer/producer annotations. These annotations will ensure that the functions are in the *e-treeless with constant* form.
4. (Option) If we re-apply the deforestation algorithm to the RHS of each function again, we could now convert the functions from *e-treeless with constant* form to equivalent *e-treeless* form.

The *bottoms-up order* is used to ensure that each child (or auxiliary) function is transformed before its parent function(s). A function, f_1, is considered to be a *child function* of another function, f_2, if f_2 calls f_1 but not vice-versa. If f_1 also calls f_2, then we have *sibling* or *mutually recursive* functions. (Indirect calls through intermediate functions are also included by taking a transitive closure of the calling relationship.)

As an example of the bottoms-up order, consider the following program with a set of five functions:

```
---m(..)    <= ..p(..)..f(..)..;
---f(..)    <= ..f(..)..r(..)..;
---p(..)    <= ..q(..)..;
---q(..)    <= ..p(..)..r(..)..;
---r(..)    <= ....;
```

At the bottom of the calling hierarchy of the above program is the function $\{r\}$ which is non-recursive. Its parents are $\{f\}$ which is self-recursive and $\{p,q\}$ which is

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

a set of two mutually recursive (or sibling) functions. The last function $\{m\}$ is the top-most function of the call graph. A possible bottoms-up order for transforming the four functions is $\{r\}$, followed by $\{f\}$, then $\{p,q\}$, before $\{m\}$. Bottoms-up order ensures that child functions are always converted to e-treeless (with constant) form before their parent functions. This approach helps to meet the requirement that expressions to be fused is composed from only e-treeless (with constant) functions.

The second step requires that each set of sibling (mutually recursive) functions must be simultaneously transformed and regarded as *potentially* unsafe producers and unsafe consumers. As potentially unsafe functions, their function calls will *not* be unfolded during their functions' transformation. For example, when we transform the RHS terms of the two mutually recursive functions, say $\{p,q\}$, from the earlier example, we must regard all recursive p and q function calls as unsafe producers and consumers. After the set of sibling functions have been transformed to the e-treeless (with constant) form, we can use the static analyses of sections 4.3.1 and 4.3.2 to determine if the newly transformed functions are safe or unsafe producers, and their parameters are safe or unsafe consumers.

This analysis can be used later when we transform the RHS of their parent functions (e.g. function m). We apply static analyses to each function after transformation because syntactic properties are often changed (from unsafe to safe) by the transformation.

Under the bottoms-up order for transforming program, our analysis and transformation are *interleaved*. In particular, each set of functions that has been transformed will have to be analysed before their parent functions are transformed. With this interleaving, is our method considered an on-line or an off-line transformation method? An *on-line* method performs analysis during transformation, while an *off-line* method has its analysis done before the transformation. We feel that this interleaved analysis is basically still an off-line method. This is because the analysis is always performed separately before the transformation on each set of sibling functions. Interleaving merely help us achieve a better result using just a simple annotation scheme.

The bottoms-up order for transforming functional programs was originally proposed by Feather (1982). Apart from the fact that it gives better transformation result, it also reduces transformation time. This is because the body of each function is optimised only once. In contrast, the top-down approach (typically used by partial evaluation transformations) may apply the same optimisation to the body of a function more than once, depending on the number of times the function is called.

5 Termination proof

In this section, we present the termination proof of the extended deforestation algorithm for first-order programs. The algorithm terminates if, for any given expression, the number of recursive $\mathscr{T}$ applications is finite. This termination property of $\mathscr{T}$ can be stated as the following theorem:

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Theorem 10: Termination of $\mathcal{T}$
Given an expression, e, the number of transformation steps used by $\mathcal{T}[e]$ is always finite.

Proof
In general, the recursive $\mathcal{T}$ applications branches out in a tree-like fashion. This tree has finite branching factor because we are dealing with programs of finite sizes. As a result, we need only show that each sequence of $\mathcal{T}$ applications (in the tree) is finite. There are two main steps in this proof.

Firstly, we shall show that the number of *define steps* by $\mathcal{T}3c, 4c, 5b, 6b$ (which have corresponding fold steps) in any sequence of $\mathcal{T}$ applications is finite. Secondly, we shall show that the number of the other steps ($\mathcal{T}2, 3a, 3b, 4a, 4b, 4d, 5a, 5c, 6a, 6c$), called the *non-define steps*, which can occur between each pair of the define steps is finite.

With these two proofs, the proposed algorithm is terminating because finite numbers of *non-define steps*, between a finite number of *define steps*, implies that the total number of steps in each sequence of $\mathcal{T}$ applications is also finite. The next two sub-sections cover these two proof steps in detail.

5.1 Finite number of define steps

Our first proof step for the termination theorem can be stated as follows:
Proof Step 1: Given an expression e, the total number of define steps ($\mathcal{T}3c, 4c, 5b, 6b$) in any sequence of $\mathcal{T}[e]$ application is finite.

The number of define steps ($\mathcal{T}3c, 4c, 5b, 6b$) can be proved to be finite by showing that there exists an upper bound on the size of expressions used to define new functions. An upper bound means that there can only be finitely many different new functions (formed from a finite set of function and constructor symbols) and hence a finite number of define steps as all re-occurring expressions will be folded instead of repeating the define steps.

To prove that an upper bound exists for the size of new function definitions, we prove the following three statements, namely:

1. Consider a nesting measure $\mathcal{N}$ which computes the maximum depth of function nesting for safe subterms, as follows:

$$\begin{aligned}
&\mathcal{N}[v] && = 0\\
&\mathcal{N}[f()] && = 0\\
&\mathcal{N}[c(t_1,\ldots,t_n)] && = max\{\mathcal{N}[t_1],\ldots,\mathcal{N}[t_n]\}\\
&\mathcal{N}[f(t_1,\ldots,t_n)] && = max(1 + max\{\mathcal{N}[t_i] \mid i \in 1..n, safe(t_i)\}\\
&&&\qquad ,max\{\mathcal{N}[t_i] \mid i \in 1..n, unsafe(t_i)\})\\
&\mathcal{N}[g(t_0,\ldots,t_n)] && = max(1 + max\{\mathcal{N}[t_i] \mid i \in 0..n, safe(t_i)\}\\
&&&\qquad ,max\{\mathcal{N}[t_i] \mid i \in 0..n, unsafe(t_i)\})
\end{aligned}$$

where

$$\begin{aligned}
&safe(t^{\oplus}) && = true\\
&safe(t^{\ominus}) && = false\\
&unsafe(t) && = not\ (safe(t))
\end{aligned}$$

Each expression encountered by the $\mathcal{T}$ rule is bounded by the nesting measure, $\mathcal{N}$, as provided by the following lemma.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Lemma 1: Nesting measure $\mathcal{N}$ is bounded by $\mathcal{T}$
For each transformation rule, $\mathcal{T}[t] \Rightarrow ....\mathcal{T}[t_i]....$, we have:
$$\forall i.\ \mathcal{N}[t] \geq \mathcal{N}[t_i]$$

Proof
According to the definition of $\mathcal{N}$, sub-expressions always have smaller or equal measures. Hence, rules $\mathcal{T}$2,3b,4b,4d,5a,5c,6a,6c satisfy Lemma 1. Also, the $\mathcal{N}$ measure is bounded whenever an e-treeless (with constant) function call, with variables as unsafe arguments, is unfolded. Hence, rules $\mathcal{T}3a$,3c, 4a,4c,5b,6b also satisfy Lemma 1.

2. Each expression being transformed by $\mathcal{T}$ satisfies a grammar form, called *eft*, as stated by the following lemma.

Lemma 2: Grammar form $eft\{\Omega\}$ is preserved by $\mathcal{T}$
For each transformation rule: $\mathcal{T}[t] \Rightarrow ....\mathcal{T}[t_i]....$, we have:
$$\mathcal{N}[t] \in eft\{\Omega\} \Rightarrow \forall i.\ \mathcal{N}[t_i] \in eft\{\Omega\}$$

The grammar rule for *eft* is given below. It is parameterised by a number, called Ω, which represents the maximum number of levels allowed for *contiguous* nesting of constructors between function calls:

$$\begin{aligned} eft &::= eft\{\Omega\} \\ eft\{x\} &::= eft\{x-1\} \mid c(eft\{x-1\}_1, \ldots, eft\{x-1\}_n) \\ eft\{0\} &::= v \mid f(eft_1, \ldots, eft_n) \mid g(eft_1, \ldots, eft_n) \end{aligned}$$

The grammar form *eft* represents a class of grammars where Ω is the maximum level of contiguous nesting allowed for constructors. In any finite program, this maximum level is a finite number. Given that a number, Ω, is currently observed by an expression and each of its e-treeless functions, we can prove that this maximum level will not be breached by the $\mathcal{T}$ transformation rules, as outlined below.

Proof
According to the grammar definition of $eft\{\Omega\}$, sub-expressions of $eft\{\Omega\}$ always satisfy $eft\{\Omega\}$. Hence, rules $\mathcal{T}2, 3b, 3b, 4b, 4d, 5a, 5c, 6a, 6c$ satisfy Lemma 2. Also, rules $\mathcal{T}3a, 3c, 4a, 4c, 5b, 6b$ satisfy Lemma 2 because of the following result. Each e-treeless (with constant) function call, with variables as unsafe arguments and $eft\{\Omega\}$ as safe arguments, could result in an $eft\{\Omega\}$ expression when unfolded. For this, we assume that the e-treeless (with constant) grammar form contains additional identity function calls, $id(v)$, in the place of variables, v, to prevent variables from lying directly inside data constructors. For example, an expression $c(v_1, g(v_2))$ is assumed to be in the form $c(id(v_1), g(v_2))$:

$$\begin{aligned} et &::= et\{\Omega\} \\ et\{x\} &::= et\{x-1\} \mid c(et\{x-1\}_1, \ldots, et\{x-1\}_n) \\ et\{0\} &::= id(v) \mid f(arg_1, \ldots, arg_n) \mid g(arg_1, \ldots, arg_n) \\ arg &::= v^{\oplus} \mid f()^{\oplus} \mid et^{\ominus} \end{aligned}$$

The *id* functions are presumed to be added to help maintain the $eft\{\Omega\}$ form during unfolding. It need not be physically inserted because whenever each

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

of these functions is encountered by the deforestation algorithm, it will be removed by either a direct unfold or a fold (as a specialised argument). As a result, the final outcome of the transformation is the same with or without these identity functions.

3. For each RHS of new functions, $\text{---} new_i(...) \Leftarrow tf_i$, the actual size measure, $\mathscr{S}$, of tf_i is bounded by its nesting measure, $\mathscr{N}$, as follows: $\forall i.\ \mathscr{S}[tf_i] \leq fn(\mathscr{N}[tf_i])$ where fn is some linear function. The actual size measure, $\mathscr{S}$, computes the maximum depth of an expression (including constructors) and is directly proportional to the size of its expression (assuming finite branching). This measure is defined as:

$$\begin{array}{ll}
\mathscr{S}[v] & = 0 \\
\mathscr{S}[f()] & = 0 \\
\mathscr{S}[c(t_1,\ldots,t_n)] & = 1+max\{\mathscr{N}[t_1],\ldots,\mathscr{N}[t_n]\} \\
\mathscr{S}[f(t_1,\ldots,t_n)] & = 1+max\{\mathscr{N}[t_1],\ldots,\mathscr{N}[t_n]\} \\
\mathscr{S}[g(t_0,\ldots,t_n)] & = 1+max\{\mathscr{N}[t_0],\ldots,\mathscr{N}[t_n]\}
\end{array}$$

Each new RHS of function definition introduced by $\mathscr{T}3c, 4c, 5b, 6b$ is a generalised version of the grammar form eft^{Δ}. This generalised grammar form can be defined as:

$$\begin{array}{lll}
eft^{\Delta} & ::= & eft\{\Omega\}^{\Delta} \\
eft\{x\}^{\Delta} & ::= & eft\{x-1\}^{\Delta} \mid c(eft\{x-1\}^{\Delta}{}_1,\ldots,eft\{x-1\}^{\Delta}{}_n) \\
eft\{0\}^{\Delta} & ::= & v \mid f(arg_1,\ldots,arg_n) \mid g(arg_1,\ldots,arg_n) \\
arg & ::= & v^{\ominus} \mid eft\{\Omega\}^{\Delta\oplus}
\end{array}$$

To prove that the size of new functions introduced by the define steps is bounded, we merely have to prove that $\mathscr{S}$ is bounded by $\mathscr{N}$ for each expression of the form eft^{Δ}. This proof is sufficient because $\mathscr{N}$ is already bounded by $\mathscr{T}$ and all RHS introduced by the define step satisfy the eft^{Δ} grammar form. We formulate this requirement as Lemma 3 below.

Lemma 3: The $\mathscr{S}$ measure is bounded by the $\mathscr{N}$ measure for each expression from the grammar $eft\{\Omega\}^{\Delta}$, as follows:

$$\forall e.\ e \in eft\{\Omega\}^{\Delta} \Rightarrow\ \mathscr{S}[e] \leq (\Omega+1)+(\Omega+1)*\mathscr{N}[e]$$

Proof

The above lemma can be proved inductively over each production rule of the form $eft\{x\}^{\Delta} ::= something$ by using a tighter formula:

$$\mathscr{S}[eft\{x\}^{\Delta}] \leq (x+1)+(\Omega+1)*\mathscr{N}[eft\{x\}^{\Delta}]$$

This proof is straightforward and is left to the reader.

These three statements prove that there is a finite number of define steps. In particular, $\mathscr{N}$ is bounded by $\mathscr{T}$ and the grammar form of eft is preserved by $\mathscr{T}$. Furthermore, the size ($\mathscr{S}$ measure) of the RHS of new functions (of form eft^{Δ}) is itself bounded by $\mathscr{N}$. Hence, there exists an upper bound on the size of expressions used for each new function introduced by the define steps. As a result, the number of define steps is finite as all re-occurring expressions will be folded.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

5.2 Finite number of non-define steps

Our second proof step for the termination theorem can be stated as follows:

Proof Step 2: Between each pair of define steps ($\mathcal{T}3c, 4c, 5b, 6b$), there could only be a finite number of the other non-define steps ($\mathcal{T}2, 3a, 3b, 4a, 4b, 4d, 5a, 5c, 6a, 6c$).

Between each pair of the define steps, there may be an unknown number of non-define steps. This can cause non-termination unless we can show that there are only finite numbers of them.

We show this by proving that there is a well-founded decreasing measure for these non-define rules. We observe that rules $\mathcal{T}$2,3b,4b,4d,5a, 5c,6a,6c operate on successively smaller expressions, while rule $\mathcal{T}4a$ unfolds non-recursive functions, and $\mathcal{T}3a$ consumes a constructor by unfolding a g-type function call. A well-founded decreasing measure may thus be formed by a combined measure of expression size, constructor numbers and function hierarchy number. The function hierarchy number (obtained by a function named H) is a positive number associated with each function to indicate where the function lies in the calling hierarchy. Functions at the top of the calling hierarchy will have bigger numbers, while those at the same level (e.g. sibling functions) will have identical numbers. The combined measure, called $\mathcal{M}$, is actually a multi-set (set where duplicates are allowed). It must be collectively decreased by each of the non-define steps and is defined as follows:

$$
\begin{aligned}
\mathcal{M}(x,y)[v] &= \{\} \\
\mathcal{M}(x,y)[f()] &= \{\} \\
\mathcal{M}(x,y)[c(t_1,\ldots,t_n)] &= \{(x,y+1,0)\} \cup \textstyle\bigcup_{i=1}^{n} \mathcal{M}(x,y+1)[t_i] \\
\mathcal{M}(x,y)[f(t_1,\ldots,t_n)] &= \{(x+1,0,H(f))\} \cup \textstyle\bigcup_{i=1}^{n} \{\mathcal{M}(x+1,0)[t_i] \mid \mathit{safe}(t_i)\} \\
&\qquad \cup \textstyle\bigcup_{i=1}^{n} \{\mathcal{M}(x,0)[t_i] \mid \mathit{unsafe}(t_i)\} \\
\mathcal{M}(x,y)[g(t_0,\ldots,t_n)] &= \{(x+1,0,H(g))\} \cup \textstyle\bigcup_{i=0}^{n} \{\mathcal{M}(x+1,0)[t_i] \mid \mathit{safe}(t_i)\} \\
&\qquad \cup \textstyle\bigcup_{i=0}^{n} \{\mathcal{M}(x,0)[t_i] \mid \mathit{unsafe}(t_i)\}
\end{aligned}
$$

Notice that this measure provides a triple (a tuple of three integers) for every subterm which is either a constructor or function call. The triple consists of a function nesting level (among safe sub-term), a constructor nesting level (from the last function call) and a function hierarchy number (for the current function).

The $\mathcal{M}$ measure has a finite largest measure (for each expression) and a finite smallest measure. It has a finite largest measure because the definition of $\mathcal{M}$ results in a finitely-sized multi-set for each finitely-sized expression. Also, the smallest measure for the multi-set is $\{\}$. To prove that $\mathcal{M}$ also decreases for each of the non-define steps, we have the following lemma:

Lemma 4: Bag measure $\mathcal{M}$ is decreasing for the non-define steps of $\mathcal{T}$
For each non-define step, $\mathcal{T}[t] \Rightarrow \ldots\ldots\mathcal{T}[t_i]\ldots\ldots$,
We have: $\forall i, x, y.\ \mathcal{M}(x,y)[t] > \mathcal{M}(x,y)[t_i]$

Proof

Using this measure, proper sub-terms of an expression will always have a smaller measure. Hence, $\mathcal{T}2, 3b, 4b, 4d, 5a, 5c, 6a, 6c$ satisfy Lemma 4. Also, each e-treeless (with constant) expression, e, satisfies $\mathcal{M}(x,y)[e] < \{(x+1,1,0)\}$. As a result, the

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

direct unfolds of $\mathscr{T}3a, 4a$ (with variables for unsafe arguments and arbitrary linear safe arguments) will not increase the multi-set $\mathscr{M}$ measure. In addition, the $\mathscr{M}$ measure is actually decreased because $\mathscr{T}3a$ loses a constructor and $\mathscr{T}4a$ gets a smaller function hierarchy number during these direct unfoldings.

6 Fusion of higher-order programs

Higher-order functional languages treat functions as first-class citizens where they are allowed to be passed as *arguments* and be returned as *results*. This facility increases the expressive power of the language and permits more succint and reusable program codes to be written. However, the facility comes at a price. Higher-order programs, being more general, are more difficult to analyse for optimisations and transformations.

Our approach to handling higher-order programs is to use another transformation technique, called *higher-order removal* (Chin, 1990; Chin & Darlington, 1992), that is capable of converting *most* higher-order expressions to either first or lower order. Some residual higher-order features may remain, but the new expression form is simpler and easier to handle (by optimisation and transformation techniques) than the full higher-order expression form.

Consider the following grammar for higher-order expressions:

$$t \quad ::= v \mid c(t_1, \ldots, t_n) \mid t(t_1, \ldots, t_n) \mid f \mid g \mid \mathit{lambda}\ (v_1, \ldots, v_n) \rightarrow t\ \mathit{end}$$

Compared to the grammar for first-order expressions, some new language features that have to be handled include applications, $t(t_1, \ldots, t_n)$, lambda abstractions, *lambda* $(v_1, \ldots, v_n) \rightarrow t$ *end*, and in general, function-type arguments and function-type results. Of particular interest are two specific classes of higher-order expressions that can be eliminated, namely: *curried applications* and *instantiated function-type arguments that are non-accumulating*.

Curried applications include all applications except *function calls*, $f(t_1, \ldots, t_n)$ or $g(t_0, \ldots, t_n)$, and *variable applications*, $va(t_1, \ldots, t_n)$, where:

$$va ::= v \mid va(t_1, \ldots, t_n)$$

Curried applications can always be eliminated by a technique, called *lump uncurrying*, which replaces each curried application by an equivalent uncurried function call. Instantiated function-type non-accumulating arguments, on the other hand, can be eliminated by a function specialisation transformation which works in a similar way to deforestation's elimination of safe sub-terms. Both techniques can be combined into a higher-order removal algorithm, named $\mathscr{R}$ (Chin, 1990; Chin & Darlington, 1992), which has been proven to be terminating, as long as only well-typed higher-order programs are used. Well-typed programs are programs which pass the Hindley-Milner type-checking algorithm (Milner, 1978). This property is needed to ensure that the higher-order removal algorithm terminates. In a well-typed program, each expression can always be given a finite type. As the order of an expression can be defined in terms of its type, each well-typed expression will always have a finite order to start with. Higher-order removal attempts to lower this order via

transformation. This can be shown not to go on forever for expressions that have finite orders.

Using $\mathscr{R}$, each higher-order expression can be transformed to an equivalent expression of the following restricted higher-order form:

Definition 11: Higher-order specialised form
An expression is said to be *higher-order specialised* (or *HO-specialised*) if it satisfies the grammar below:

$$t \quad ::= v \mid c(t_1,\ldots,t_n) \mid rapp \mid f \mid g \mid lambda\,(v_1,\ldots,v_n) \rightarrow t\ end$$
$$rapp ::= va(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_m,v_{m+1}^{\oplus_h},\ldots,v_n^{\oplus_h})$$
$$\mid g(t_0,\ldots,t_m,v_{m+1}^{\oplus_h},\ldots,v_n^{\oplus_h})$$

where variables $v_{m+1}^{\oplus_h},\ldots,v_n^{\oplus_h}$ are the non-accumulating function-type arguments annotated with $\oplus_h$ and f, g are HO-specialised functions.

Correspondingly, a function is said to be *HO-specialised*, if each of its definition's RHS term(s) is *HO-specialised*.

A new type of annotation, $\oplus_h$, has been introduced to mark function-type arguments which are non-accumulating (identical to the syntactic criterion given in section 4.3.1). All instantiated versions of such arguments will have been removed by the transformation algorithm, $\mathscr{R}$, as shown in the above restricted grammar form. Also, some residual higher-order features are still present in the above form. However, they are easier to handle than the full higher-order form. Let us show how these residual higher-order features can be handled by the deforestation algorithm.

Firstly, in each of the function calls of the form:

$$f(t_1,\ldots,t_m,v_{m+1}^{\oplus_h},\ldots,v_n^{\oplus_h})\ or\ g(t_0,\ldots,t_m,v_{m+1}^{\oplus_h},\ldots,v_n^{\oplus_h})$$

we may still have function-type sub-terms and variable applications as arguments. These higher-order sub-terms need not be removed by deforestation. They can be handled by marking them as unsafe.

Secondly, four additional $\mathscr{T}$ rules ($\mathscr{T}7-10$ of Fig. 4) are needed to directly handle the residual higher-order features. These rules simply skip over each encountered residual feature.

Lastly, during deforestation, it is possible for new (non-residual) higher-order expressions to re-appear. These can appear when the deforestation algorithm attempts to eliminate intermediate data constructor terms which contain function-type arguments. Such constructor terms are residual features that were not eliminated by the higher-order removal method. Instead, they can be eliminated by deforestation but, in the process, they may result in new higher-order expressions. To remove these new non-residual higher-order expressions, we use $\mathscr{T}11$ of Fig. 4 to re-apply the $\mathscr{R}$ rule.

The above set of new rules have been shown in Chin (1990) not to affect the termination property of $\mathscr{T}$. As an illustration of this extended set of $\mathscr{T}$ rules, consider the following higher-order program:

dec *main: (list(int),int) → list(int);*
dec *map_ho: (list(A → B),A) → list(B);*
dec *add_five: list(int) → list(int → int);*

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

(7) $\mathscr{T}[f] \Rightarrow f$
(8) $\mathscr{T}[g] \Rightarrow g$
(9) $\mathscr{T}[va(t_1,\ldots,t_n)] \Rightarrow \mathscr{T}[va](\mathscr{T}[t_1],\ldots,\mathscr{T}[t_n])$
(10) $\mathscr{T}[lambda\ (v_1,\ldots,v_n) \rightarrow t\ end] \Rightarrow lambda\ (v_1,\ldots,v_n) \rightarrow \mathscr{T}[t]\ end$
(11) $\mathscr{T}[\cdots ho\_exp \cdots] \Rightarrow \mathscr{T}[\cdots \mathscr{R}[ho\_exp] \cdots]$
where *ho_exp* is either a curried application or a function call with non-accumulating non-variable function-type arguments

Fig. 4. Additional $\mathscr{T}$ Rules for higher-order programs.

```
---main(xs,y)          <= T [map_ho(add_five(xs),y) ]          ;T 5b
---main(nil,y)         <= T [map_ho(nil,y) ]                   ;T 3a
                       <= T [nil ]                             ;T 2
                       <= nil
---main(cons(x,xs),y)  <= T [map_ho(cons(lambda z -> z+5*x end,add_five(xs)),y) ]
                                                               ;T 3a
                       <= T [cons(lambda z -> z+5*x end (y),map_ho(add_five(xs),y)) ]
                                                               ;T 2
                       <= cons(T [lambda z -> z+5*x end(y) ]
                             ,T [map_ho(add_five(xs),y) ])     ;T 11, 5a
                       <= cons(T [y+5*x ],main(xs,y))          ;T 1, 2, 4d
                       <= cons(y+5*x,main(xs,y))
```

Fig. 5. An Example of higher-order deforestation.

```
---main(xs,y)          <= map_ho(add_five(xs),y);
---map_ho(nil,y)       <= nil;
---map_ho(cons(f,fs),y)<= cons(f(y),map_ho(fs,y));
---add_five(nil)       <= nil;
---add_five(cons(x,xs)) <= cons(lambda z -> z+5*x end, add_five(xs));
```

This program contains a function, *add_five*, which builds up a list of functions and another function, *map_ho*, which applies each function from its list of functions (as its first parameter) to its second parameter. The above program is already in the HO-specialised form. Using the new (higher-order) deforestation algorithm, the intermediate list (of functions) from *main* can be removed by transforming *main* to the following first-order function:

```
---main(nil,y)         <= nil;
---main(cons(x,xs),y)  <= cons(y+5*x,main(xs,y));
```

The transformation steps to achieve this is illustrated in Fig. 5. Notice the invocation of rule $\mathscr{T}11$ in order to remove a new higher-order expression that has re-appeared.

Wadler (1988) has also considered higher-order extension for deforestation. However, his solution is not general, as it relies on a restricted higher-order facility, called *higher-order macro*, whose use must be converted to first-order equivalent before deforestation. Higher-order macros essentially correspond to higher-order functions with *fixed* function-type parameters that do not change across recursion. They can neither return function-type results, nor support constructor terms which contains functions. The advantage of the higher-order macro scheme is its simplicity, but it requires the user to adopt a restricted higher-order language.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

7 Are the syntactic criteria adequate?

Our generalisation of the deforestation algorithm relies primarily on sufficient syntactic properties for classifying producers and consumers as either safe or unsafe. The advantage of using syntactic (as opposed to semantic) properties is that they are simple. However, there is also a potential danger that these properties can be easily changed via seemingly harmless syntactic manipulations! This section presents some real, as well as perceived, dangers posed by syntactic differences in our program, and shows why they may not be so serious after all. Nevertheless it must be noted that the syntactic criteria proposed in this paper are based on safe approximations. They do not detect all possible opportunities for effective fusion, merely a sub-class of them. In fact, section 8 contains further techniques which could be used to uncover more opportunities for safe fusion. These could be considered as enhancements to compensate for the inadequacies of our simple syntactic analysis scheme.

In an earlier work by the author (Chin, 1990, 1991), unsafe producers were syntactically changed to pseudo-safe equivalent. This was done by using the *let* construct to abstract out each sub-term (unsafe recursive call) that did not conform to the e-treeless form. As an example, the *rev_flatten* function can be converted to this pseudo-safe form, by extracting out the recursive *rev_flatten* call with *let*, as shown below:

$$\begin{aligned} &\text{---}\mathit{rev\_flatten(nil)} && \Leftarrow \mathit{nil};\\ &\text{---}\mathit{rev\_flatten(cons(as,ass))} && \Leftarrow \mathit{let\ v= rev\_flatten(ass)\ in\ append(v^{\oplus c},as^{\oplus c})};\end{aligned}$$

The newly introduced *let* construct is not allowed to be substituted by deforestation. In addition, new rules (for deforestation) must be added to handle this *let* construct. With these additional considerations, the original unsafe producer now appears pseudo-safe. In particular, notice that the two safe arguments of the *append* call are now variables. The new function definition is e-treeless even after it has been regarded as a safe producer. It can therefore be unfolded safely as a producer but this will not result in real fusion. This is because the pseudo-safe sub-term will be abstracted by the *let* construct, rather than eliminated, during deforestation. Apart from the added complication of a new *let* construct, a minor problem of pseudo-safe sub-terms is that they tend to result in larger transformed programs. Another problem of the *let* construct is that it is not compatible with an improvement technique which uses laws to handle unsafe producers (see section 8.6).

Similarly, *let* constructs can also be used to trivially convert non-linear and/or accumulating parameters to linear and non-accumulating parameters. This results in pseudo-safe consumers but does not cause any further effective fusion.

Simple syntactic changes can also convert safe consumers or producers to equivalent pseudo-unsafe ones. This is more alarming because less fusion than ought to, may occur! A very simple syntactic change is to use the identity function, $\text{---}\mathit{id(x)} \Leftarrow x$, to wrap around appropriate sub-terms; so that safe producers become pseudo-unsafe, or non-accumulating parameters become accumulating. An example of this is the following modified definition of *double* which is now a pseudo-unsafe producer and has a pseudo-unsafe consumer:

$$\begin{aligned} &\text{---}\mathit{double(nil)} && \Leftarrow \mathit{nil};\\ &\text{---}\mathit{double(cons(a,as))} && \Leftarrow \mathit{cons(2*a,id(double(id(as))))};\end{aligned}$$

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Fortunately, the extended deforestation algorithm is able to remove these simple wrapping functions by direct unfoldings (using $\mathcal{T}4a$). This is done during the transformation of each of the functions. As the static properties are analysed after each function's transformation, these simple wrapping functions do not cause any real problem.

However, more elaborate wrapping functions that are *g*-type, such as the one shown below, can cause problems to our algorithm:

```
---id(nil)        <= nil;
---id(cons(x,xs)) <= cons(x,xs);
```

Unlike non-recursive *f*-type function calls which could be directly unfolded by rule $\mathcal{T}4a$, there is presently no equivalent rule to directly unfold non-recursive *g*-type function calls. As a result, our current algorithm is unable to cope with these more elaborate wrapping functions. Given the rather contrived technique used to deceive the algorithm, this is a shortcoming we could tolerate. (Nevertheless, some enhanced techniques in section 8 for handling *g*-type functions could overcome even these more elaborate wrapping functions.)

Another potential problem of the syntactic criteria is that they rely heavily on the use of pattern-matching equations. However, most functional languages also support conditional expressions, together with the associated programming style. For example, the *double* function can be expressed in the conditional style as follows:

```
---double(xs)      <= if(null(xs), nil, cons(hd(xs)*2,double(tl(xs))));
---tl(cons(x,xs))  <= xs ;
---hd(cons(x,xs))  <= x ;
---null(nil)       <= true ;
---null(cons(x,xs)) <= false ;
---if(true,x,y)    <= x ;
---if(false,x,y)   <= y ;
```

The conditional construct, *if*, is defined as a function using pattern-matching equations instead of a language primitive. Similarly, functions to manipulate the list structure, such as *hd*, *tl* and *null*, can be defined using equations. A noticeable concern is that this alternative definition of *double* now has the syntactic form of both an unsafe producer and an unsafe consumer! Does this means that we have to change our syntactic criteria or can we change this function back to the earlier safe form? Fortunately, it is still possible to transform the above definition of *double* back to its earlier safe form. This is so because *null* is a safe producer and the parameters of *if* are safe consumers. However, because of the renaming procedure (to rename the multiple variables of *xs*), the deforestation algorithm would initially obtain the following:

```
---double(xs)              <= double'(xs,xs,xs)
---double'(nil,ys,zs)      <= nil ;
---double'(cons(x,xs),yx,zs) <= cons(hd(ys)*2,double(tl(zs)));
```

The renaming procedure was introduced to help ensure safe fusion for re-curring expressions. However, in the above example, the expression used to define the intermediate function *double'* did not re-occur (no fold back to *double'* occurred). Hence, this intermediate function is not actually needed but we presently do not

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

have a way of predicting this in advance. In section 8.3, we shall introduce a post-processing technique to enhance safe fusion by removing unnecessary intermediate mutual recursion. In the above example, the *double'* call could be eliminated by a direct unfold (followed by further transformation) to obtain the following:

```
---double(nil,)          <= nil ;
---double(cons(x,xs))    <= cons(x*2,double(xs));
```

Hence, the extended deforestation algorithm and its associated enhancements could help us convert some (unsafe) conditional-style functions to the (safe) pattern-matching form.

8 Further improvements

Further improvements to our safe fusion method are possible. These improvements can help remove more intermediate data structures from user programs. In the last section, we saw that the syntactic criteria used are merely approximations which conservatively estimate where safe fusion is possible. There are therefore scope for further improvements.

One obvious avenue for improving fusion is to improve the analysis scheme so that more producers/consumers can be classified as safe. Specifically, we could try to formulate a semantic-based analysis rather than the syntactic one that was presented. However, this could complicate the fusion method and may also involve significant changes to the transformation algorithm.

To keep things simple, we will adhere closely to the current syntactic analysis and instead use new transformation and re-annotation techniques to help convert more functions to safe producers/consumers. Seven of these techniques are described next. These techniques are safe (terminates with no loss of efficiency) and deterministic (not heuristic).

8.1 Parameter linearisation

Some non-linear pattern-matching parameters could be made safe by linearising the parameters. This technique can be used to convert certain unsafe consumers to equivalent safe ones. Consider the function:

```
dec square: list(int) → list(int);
---square(nil^⊖c)          <= nil;
---square(cons(x, xs)^⊖c)  <= cons(x*x,square(xs));
```

The parameter of this function is presently unsafe because the auxiliary variable, *x*, occurs twice in the RHS of the second equation. This makes the whole pattern-matching parameter unsafe (accordingly annotated above), even though the main recursive variable, *xs*, is linear and non-accumulating. A simple technique that can make such a pattern-matching parameter safe (or linear) is to abstract out the non-linear auxiliary variable with an intermediate function, call it *f_int*, as follows:

```
---square(cons(x, xs)^⊕c) <= f_int(x,xs);
---f_int(x^⊖,xs^⊕)        <= cons(x*x,square(xs));
```

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

This linearisation technique can be applied to each non-accumulating pattern-matching parameter that is non-linear because of auxiliary variables. Once linearised, the pattern-matching parameter can be classified as a safe consumer for fusion purpose. Alternatively, we could also use the *let* construct to abstract out each non-linear auxiliary variable. However, this is not done here because, for exposition purpose, we would like to keep our transformation algorithm simple by using a smaller language without *let*. (Note: The intermediate function introduced here will be marked as an essential intermediate that will not be removed by the technique of section 8.3.)

8.2 Unfolding non-recursive g-type calls

A second possible way to obtain more safe consumers is to convert some of the accumulating parameters to non-accumulating ones. In section 7, we showed that some pseudo-unsafe consumers (wrapped with the *id* function) can be converted to safe ones through the direct unfolding of non-recursive *f*-type functions. If non-recursive *g*-type functions could also be unfolded in a similar way, it may result in more safe consumers. Consider the following function, *rem*, which is used to return the remainder of a list after the first *n* elements have been removed:

```
dec rem: (int,list(A)) → list(A);
---rem(0,ys)     ⇐ ys ;
---rem(n+1,ys) ⇐ rem(n,tl(ys)) ;
```

Currently, the second parameter of this function is an accumulating parameter because it has a non-recursive *g*-type function call (*tl*) wrapped around the second argument of its recursive call. If a direct unfold can be performed on this *tl* call, there may be a chance that this parameter can be changed to a non-accumulating one. An unfold on this *tl* call can be carried out as follows:

```
---rem(n+1,ys)      ⇐ rem(n,tl(ys)) ;        ! define frem & fold
                    ⇐ frem(n,ys) ;
---frem(ys,n)       ⇐ rem(n,tl(ys)) ;        ! unfold tl
---frem(cons(y,ys),n) ⇐ rem(n,ys) ;
```

We introduce an intermediate function, called *frem*, before performing an unfold on *tl*. Such intermediate functions are required when unfolding each *g*-type function call which needs to have its pattern-matching argument instantiated. This is to help ensure that the transformed functions remain *g*-type, i.e. single simple pattern-matching parameter per function.

In addition, each non-recursive *g*-type function call can only be so unfolded if it presently lies in a *strict* context. Consider an expression ...*t*... where the ellipses represent an arbitrary context, and *t* is a term in the hole of the context. This context is said to be *strict* if the term *t* will be evaluated when the whole expression is evaluated. This requirement is needed in order to preserve the lazy semantics of the transformed code. Specifically, if *t* is a non-recursive *g*-type call of the form $g(v,t_1,\ldots,t_n)$, then applying an unfold on this *g*-type call¶ will cause its pattern-

¶ Unfolding a *g*-type call within a context has essentially the same effect as floating out a case construct over a context, namely ...case v of $\{p_1 \rightarrow t_1; \ldots; p_n \rightarrow t_n\}$... ⟺

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

matching argument, *v*, to be instantiated and thus made strict. This should only be allowed if the *g*-type call is lying in a strict context; otherwise the lazy semantics of this code may be altered (Runciman *et al.*,1989).

Hence, more safe consumers can be obtained by directly unfolding non-recursive *g*-type function calls that lie in strict context. This step could be selectively applied on non-recursive g-type function calls which are the cause of accumulating parameters.

8.3 Removing unnecessary mutual recursion

Unnecessary non-recursive function calls are currently handled by $\mathscr{T}4a$ (for *f*-type calls) and section 8.2 (for *g*-type calls). Their removal by unfolding can help obtain more efficient programs with better opportunities for fusion. Similarly, it is also advantageous to remove unnecessary intermediate *mutual recursive* function calls from the original and transformed programs.

An example of unnecessary mutual recursive *f*-type function is shown below:

```
---rev_it(nil,w)          <= w;
---rev_it(cons(a,as),w)   <= intm(as,cons(a,w));
---intm(as,w)             <= rev_it(as,w);
```

The two functions, *intm* and *rev_it*, are mutually recursive. However, *intm* is an unnecessary intermediate in the RHS of *rev_it*. In this definition, the second parameter of *rev_it* can be considered as non-accumulating; however its original accumulating condition has merely been transferred to the second parameter of the intermediate function, *intm*. This intermediate function is undesirable because it makes the program larger and results in only pseudo-safe form.

We shall attempt to remove, where possible, intermediate mutual recursive function that are not *self-recursive*. A function definition is said to be not *self-recursive* if it does not contain any calls back to itself in its RHS. For example, the function, *intm*, is not self-recursive because its RHS does not contain calls to *intm*. Self-recursive functions cannot be completely eliminated by unfolding because its RHS contains another similar call. To remove the unnecessary intermediate call in the RHS of *rev_it*, we could perform a direct unfold on *intm* to obtain the following program.

```
---rev_it(nil,w)          <= w;
---rev_it(cons(a,as),w)   <= rev_it(as,cons(a,w));
---intm(as,w)             <= rev_it(as,w);
```

This direct unfold results on a smaller, more efficient self-recursive *rev_it* function. Also, *intm* is now a non-recursive function. Further calls to *intm* by functions higher-up in the calling hierarchy could be similarly unfolded to call *rev_it* directly.

Some unnecessary mutual recursive functions may prevent fusion from taking place. Consider the function:

case v of $\{p_1 \rightarrow ...t_1...;..;\ p_n \rightarrow ...t_n...\}$. To preserve lazy semantics, this step should only be allowed if the case construct is currently lying in a strict context. Thanks to Lennart Augustsson for pointing this out.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

```
---prog(nil,w)          <= w;
---prog(cons(a,as),w)   <= prog_int(as,double(w));
---prog_int(as,w)       <= prog(as,double(w));
```

When the deforestation method is applied to the above functions, the intermediate recursive call, *prog_int*, will not be unfolded because it is a potentially unsafe producer/consumer. However, this intermediate function is currently preventing a safe sub-term, *double(double(w))* from being discovered. The safe sub-term is exposed after we apply a direct unfold on the *prog_int* call, as shown below:

```
---prog(nil,w)          <= w;
---prog(cons(a,as),w)   <= prog(as,double(double(w)));
```

This elimination of unnecessary mutual recursive f-type function calls can help us obtain more efficient program with better opportunities for optimisation. However, we must ensure that no large arguments are duplicated.

More formally, a f-type function call, $f(t_1,\ldots,t_n)$, could be *directly unfolded* if it satisfies the following:

1. There is no self-recursive f function call in the RHS of the function. This is to avoid infinite unfolding of self-recursive functions.
2. All non-linear arguments of the call must be trivial expressions (i.e. variables or constants). This is to avoid loss of efficiency via code duplication.

Similarly, it is also desirable to eliminate unnecessary mutual recursive g-type function calls. As an example, consider the transformed program from section 7:

```
---double(xs)              <= double'(xs,xs,xs)
---double'(nil,ys,zs)      <= nil ;
---double'(cons(x,xs),yx,zs) <= cons(hd(ys)*2,double(tl(zs)));
```

An intermediate *double'* call was introduced which contains three identical occurrences of the variable *xs*. This intermediate g-type function is currently contributing to the accumulating condition of the parameter of *double*. As this call is lying in a strict context, we could perform a direct unfold to eliminate the unnecessary intermediate function and at the same time propagate the multiple variable *xs* forward. Doing so results in the following:

```
---double(nil)          <= nil ;
---double(cons(x,xs))   <= cons(hd(cons(x,xs))*2,double(tl(cons(x,xs))));
```

After unfolding away the *hd* and *tl* calls, we obtain a *double* function which is both a safe producer and has a safe consumer.

In general, it is advantageous to eliminate a mutually recursive g-type function call, $g(v,t_1,\ldots,t_n)$, if one or more of the following extra conditions are present.

1. The to-be-instantiated variable v occurs more than once. This condition can facilitate further optimisation as shown above.
2. One or more of the safe arguments of $t_1,\ldots,t_n$ must be non-trivial. This condition could help reveal safe sub-terms which are hidden by the mutual recursion.
3. The g-type call is lying in a strict context. This allows the removal of an intermediate g-type function call by direct unfolding.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

8.4 Re-annotating unsafe producers

Presently, a function is considered to be an unsafe producer if it has *obstructing* function calls (see section 4.3.2 for a definition) that contain some recursive producer calls as their safe arguments. Such a producer is unsafe because there is a chance that the obstructing function calls may be accumulated during transformation (since pattern-matching equations cannot consume them). However, the wholesale treatment of functions with obstructing calls as unsafe producers may be more stringent than necessary. This is because some consumers may disregard certain parts of unsafe producers. To illustrate this phenomenon, consider the following unsafe producer:

```
dec fp: list(int) → list(int);
---fp(nil)          ⇐ nil ;
---fp(cons(a,as)) ⇐ cons(sum(fp(as)),fp(as)) ;
```

This function is an unsafe producer because one of its recursive calls is lying in the safe consumer of *sum*. Hence, the *sum* function call is an obstructing call that may be accumulated when *fp* is used as a producer. However, some consumers may actually bypass the obstructing calls and avoid the accumulation of these calls during fusion. As an example, consider another function:

```
dec lfp,length: list(int) → int;
---lfp(as)          ⇐ length(fp(as)) ;
---length(nil)      ⇐ 0;
---length(cons(x,xs)) ⇐ 1+length(xs);
```

In the RHS of the *lfp* function, the *length* function call is nested with the unsafe producer, *fp*. According to our annotation scheme, this nesting is unsafe to fuse. However, this is not quite true because the *length* call will actually ignore the obstructing *sum* calls from *fp* during fusion. Application of the deforestation algorithm to the above function results in the following fused function:

```
---lfp(nil)          ⇐ 0 ;
---lfp(cons(a,as))   ⇐ 1+lfp(as) ;
```

This example suggests that it may be too conservative to label the whole function as an unsafe producer so that its definition become e-treeless. On closer inspection, it is the obstructing function calls which are the principal cause of non e-treeless form. If we can change the annotations on these obstructing function calls, we may change the unsafe producers to safe ones. In fact, this can be very simply achieved by annotating all occurrences of obstructing calls as unsafe producers and unsafe consumers. (This treatment makes them look like constructors which cannot be consumed. Primitive functions are also treated in this way.) The idea of this re-annotation is to prevent these obstructing calls from being fused with the outer calls (the consumers) as well as the inner calls (the earlier unsafe producers). This indirectly prevents the accumulation of obstructing calls during fusion. To mark these calls uniquely, we place a small *o* subscript on each obstructing function call. With this suggestion, the earlier function *fp* can be re-annotated as follows:

```
---fp(nil)          ⇐ nil ;
---fp(cons(a,as)) ⇐ cons( sum_o(fp(as)^⊖c )^⊖p,fp(as)) ;
```

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

An intended consequence of this re-annotation is that the *fp* function is now e-treeless even when it is regarded as a safe producer. This re-annotation technique can be applied to all functions which were originally unsafe. Instead of marking these functions as unsafe, the re-annotation technique has transferred the unsafe markings to the obstructing function calls. In this way, all the original unsafe producers can now be made safe. We call these re-annotated functions (which contain obstructing calls) the *re-annotated producers.*

The re-annotation technique requires a small but fundamental change to the original annotation scheme. In our initial monovariant scheme, the producer and consumer annotations are associated with each function definition so that all calls to the same function have identical annotations. However, with this new re-annotation technique, our scheme has to be generalised to a *polyvariant* scheme that permits different annotations for distinct function calls. In particular, the obstructing calls might be annotated differently from that suggested for their function definitions.

This re-annotation technique is useful because re-annotated producers can now be safely unfolded as producers. In some cases (e.g *length(fp(as))*), more real fusion may result. In other cases, the obstructing calls may stand in the way of outer consumers but the transformation algorithm will still terminate. One possible disadvantage is that the transformed program may be larger than before, but no loss of efficiency (measured in terms of reduction steps) results.

8.5 Fusion within re-annotated producers

Re-annotated producers contain obstructing function calls. Occasionally, it may be possible to remove these obstructing calls by the fusion method itself, in order to obtain genuine safe producers. The obstructing function calls are nested with unsafe producers. We could apply the deforestation algorithm to such nested compositions in an attempt to fuse them. If the attempt succeeds, then the obstructing calls will disappear; otherwise some obstructing calls may still be present.

An example of this is the earlier function *fp* with *sum* as its obstructing call. We could make a tentative attempt to remove this obstructing call by safe fusion. If the attempt succeeds, we keep the new program. If it fails, we revert back to the original form of the re-annotated producer.

To make this fusion attempt on *fp*, we initially define a new function containing a composition of the obstructing function (without the unsafe annotations) and the re-annotated producer, as follows:

```
---nfp(as)          ⇐ sum(fp(as)) ;
```

Safe fusion is then applied to see if all obstructing calls can be removed. In the above case, the new function which results, shown below, do not contain any more obstructing calls.

```
---nfp(nil)         ⇐ 0 ;
---nfp(cons(a,as)) ⇐ nfp(as) + nfp(as) ;
```

As a result, we allow this fusion to be committed by applying a fold to the body of *fp*, as follows.

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

$$\begin{aligned} \text{---}fp(cons(a,as)) &\Leftarrow cons(sum(fp(as)),fp(as)) ; \\ &\Leftarrow cons(nfp(as),fp(as)) ; \end{aligned}$$

This technique of fusing the obstructing calls with the re-annotated producers may succeed or fail. Successful fusion results in new producers without the obstructing calls. Such safe producers could help facilitate more fusion. An important characteristic of this technique is that it is *decidable* and *terminating*. It is decidable because success or failure is solely determined by the absence or presence of obstructing function calls in the final transformed program. This can be syntactically determined. It is terminating because it is based on the extended deforestation algorithm which has already been shown to terminate.

8.6 Laws for re-annotated producers

Another way to improve the fusion method is to make use of *laws*, in addition to the equations of user-defined functions. Laws can help improve fusion by allowing some of the (remaining) re-annotated producers to be successfully fused as producers. Consider the program:

$$\begin{aligned} &\textbf{data } tree(A) = leaf(A) +\!\!+ node(tree(A),tree(A)); \\ &\textbf{dec } sizet: tree(A) \rightarrow int; \\ &\textbf{dec } flatten: tree(A) \rightarrow list(A); \\ &\text{---}sizet(t^{\oplus_c}) \Leftarrow length(flatten(t)); \\ &\text{---}flatten(leaf(a)^{\oplus_c}) \Leftarrow cons(a,nil); \\ &\text{---}flatten(node(lt,rt)^{\oplus_c}) \Leftarrow append_O(flatten(lt),flatten(rt)); \end{aligned}$$

Presently, the expression *length(flatten(t))* cannot be effectively fused. The reason is that obstructing $append_o$ function calls (produced by *flatten*) cannot be consumed by the pattern-matching equations of the outer *length* call. Applying the deforestation algorithm to the RHS of *sizet* could only result in the following program:

$$\begin{aligned} &\text{---}sizet(leaf(a)^{\oplus_c}) \Leftarrow 1+0; \\ &\text{---}sizet(node(lt,rt)^{\oplus_c}) \Leftarrow length(append_O(flatten(lt),flatten(rt)); \end{aligned}$$

Notice that pattern-matching equations can consume constructors but not obstructing function calls. However, laws do not have this inhibition. In fact, most laws (on user-defined functions) can be viewed as rewrite rules which happily consume functions! An example is the following distributive law of *length*:

$$length(append(xs,ys)) = length(xs)+length(ys)$$

This law can be used as a rewrite rule to replace an expression matching the LHS by its RHS. Used in this manner, it can be viewed as an equation of *length* whose linear and non-accumulating parameter, *append(xs,ys)*, is a safe consumer of *append* calls. Consequently, it can be used to successfully fuse function *length* with any producers with obstructing *append* calls. In particular, this law can be used to transform *sizet* to the following more efficient function:

$$\begin{aligned} &\text{---}sizet(leaf(a)) \Leftarrow 1+0; \\ &\text{---}sizet(node(lt,rt)) \Leftarrow sizet(lt)+sizet(rt); \end{aligned}$$

Laws on user-defined functions can either be provided by users (in the same way as equations are provided) and/or be derived via some synthesis techniques (see Chin, 1992b, for a method to synthesize distributive laws). Given that these laws

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

can be made available, we simply have to add them to the set of equations used to perform unfolding. In particular, these laws will be used by rule $\mathscr{T}3$, with the obstructing function calls taking the place of constructors. The termination property of our transformation algorithm is not compromised, as long as we ensure that the RHS of laws are e-treeless (with constant) in form. This can be assured by applying the deforestation algorithm and appropriate annotations to the laws themselves!

This improvement technique using laws may not work if the pseudo-safe *let* construct (illustrated in section 7) was used to capture the re-annotated (unsafe) producers. This is so because folding transformation step might be hindered by *let* construct.

8.7 Re-annotating unsafe consumers

The re-annotation and associated techniques for handling unsafe producers (sections 8.4, 8.5, 8.6) could also be applied to unsafe accumulating parameters, in order to make them safe. In particular, one of the reasons for marking a parameter as unsafe is to avoid the possibility of accumulating sub-terms during transformation. However, this annotation can be transferred to the accumulator (*c.f.* obstructing call) rather than the parameter to stop the accumulation.

As an example, the parameter of the following *fusc* function from Dijkstra (expressed using even/odd view of integer) is currently an accumulating parameter:

```
data bint = zero ++ one ++ even(bint) ++ odd(bint);
dec fusc : binint → int;
dec succ : binint → binint;
---fusc(zero)      ⇐ 0;
---fusc(one)       ⇐ 1;
---fusc(even(n))   ⇐ fusc(n);
---fusc(odd(n))    ⇐ fusc(succ(n))+fusc(n);
---succ(zero)      ⇐ one;
---succ(one)       ⇐ even(one);
---succ(even(n))   ⇐ odd(n);
---succ(odd(n))    ⇐ even(succ(n));
```

Ignoring the linearity criterion for the moment (which does not affect non-termination), this parameter can be made safe by marking the accumulator, *succ*, as an unsafe function call, as shown below. (A non-linear parameter may duplicate sub-terms during transformation. However, if the duplicated sub-terms are eventually eliminated, no loss of efficiency results in the final program.)

$$\text{---}fusc(odd(n)^{\oplus}) \Leftarrow fusc(succ_O(n^{\ominus_c})^{\ominus_p})+fusc(n);$$

Such a re-annotation helps to localise the unsafe marking to the accumulator rather than the parameter. Notice that the above function remains e-treeless even after the accumulating parameter is considered as safe.

A good thing about the re-annotated consumer is that we can now attempt to remove the accumulator by safe fusion! In a similar technique to section 8.5, we could define a new function, called *sfusc*, where *fusc* is nested with *succ*:

```
---sfusc(n)      ⇐ fusc(succ(n));
```

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

Applying the fusion procedure to this new function results in a transformed function which no longer has the accumulator, as follows.

```
---sfusc(zero)      <= 1;
---sfusc(one)       <= 1;
---sfusc(even(n))   <= sfusc(n)+fusc(n);
---sfusc(odd(n))    <= sfusc(n);
```

The original accumulator in *fusc* could now be removed by folding with *sfusc*. The transformed function's parameter is now non-accumulating and could be fused with any safe producer. Similar to the technique for re-annotated producers, this removal of accumulators works for only a sub-class of functions but it is decidable.

Laws can also be used to help remove accumulators. This could result in either total elimination of accumulators or the replacement of one accumulator by another (hopefully cheaper one). Consider a function *f* where there is an accumulator, called *acc*, in one of its parameters.

```
---f(..,v,..)        <= ..f(..,acc(v),..)..;
```

This accumulator can be absorbed by safe fusion transformation if laws of the form, shown below, exist. Examples of such laws include *acc(acc(x))=acc(x)* for idempotent functions.

$$acc^n(v) = acc^m(v) \text{ where } n > m$$

Alternatively, if this function is used in an expression, like *f(..,g(v),..)*, and laws of the form shown below exist, then we could successfully fuse the nested *f* and *g* function calls.

$$acc(g(v)) = g(acc2(v))$$

In the process, a new accumulator *acc2* may be introduced which is hopefully cheaper than the original accumulator, *acc*. Some automatic complexity analysis methods for functional programs (such as Sands, 1990) could be used to help decide if it is advantageous to replace an old accumulator with a new one.

9 Related work

Over the years, there have been a number of different proposals for techniques which can remove unnecessary intermediate data structures from user programs. These proposals differ in name, scope, sophistication and the extent of their automation. Some of these techniques are briefly described and compared below.

One of the earliest proposal is given in the seminal paper by Burstall & Darlington (1977), where *loop combination* (fusion) of programs was shown as a transformation encompassed under the unfold/fold framework for optimising functional programs. The unfold/fold framework is very general but the transformation examples given (at that time) are largely handcrafted. Subsequently, Martin Feather (1982) built a system, called ZAP, which was able to derive low-level unfold/fold transformation sequences from higher-level pattern-directed transformations given by the users. The pattern-directed transformation contains a number of ways for expressing the desired target program form. They can be used to express certain transformations covered by the *tupling*, *generalisation* and *composition* (fusion) tactics. One large

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

example illustrated was the transformation of a multi-pass compiler into a two-pass compiler for a toy language. However, these pattern-directed transformations have to be manually specified. Our work is based on the same unfold/fold framework but we have now developed an automatic transformation algorithm for the fusion (with generalisation) tactic.

The predecessor of Wadler's (1984, 1985) deforestation is the listless transformer (1984; 1985). The first listless transformer (Wadler, 1984) is a semi-decision procedure which could convert each recursive program with *bounded evaluation* property (needs bounded internal storage to perform computation) to an equivalent listless machine (c.f. flowchart schemata with finite number of states). This transformer was able to eliminate intermediate lists (including list of lists) and achieve the effect of tupling transformation to eliminate multiple traversals of lists (from non-linear parameters). A subsequent modification to obtain a decision procedure (Wadler, 1985), requires programs to be also *pre-order* (single traversal of inputs and production of outputs in a left-to-right manner). Given two pre-order listless functions g and f, the new listless transformer is able to automatically generate a new pre-order listless function for their composition, $g \circ f$ where $\circ$ is the function composition operator. The pre-order requirement rules out certain programs which return more than one lists. As a result, tupling transformation (possible in the earlier listless transformer) is now prevented from happening. The extended deforestation algorithm is also a decision procedure. It is able to eliminate data structures apart from lists (e.g. trees) and selectively apply generalisation to avoid fusing subterms that are unsafe. In addition, the transformed program is in the source language and can thus be more easily subjected to further transformation. However, tupling capability (which requires non-linear parameters to be handled) is not present. This is not necessarily a bad thing if one considers the advantages of modularisation for program transformation. In particular, the tupling tactic can be formulated separately. In Chin (1990), we presented a range of transformation tactics (e.g higher-order removal, tupling and fusion) which are more convenient to specify individually. Some of these tactics have been appropriately combined (e.g. fusion and higher-order removal) to achieve better transformation. Other combinations of tactics (e.g. fusion and tupling) are still under investigation.

The predecessor of listless transformer is Turchin's *supercompiler* (1986). Here, *driving* (unfold using normal-order strategy) and *generalisation* techniques are used to obtain finite graphs of configurations (or states) from the symbolic evaluation of user programs. The graphs of configurations obtained are then used to compile more efficient programs. Turchin's supercompiler is basically a program specialiser which can perform both fusion and partial evaluation transformations. It is based on the REFAL language which is first-order and uses a special list data structure that could be accessed from both ends. While we have relied on a simple *off-line* generalisation technique (using an annotation scheme which is able to identify unsafe sub-terms), Turchin made use of sophisticated techniques which look back at the history of configurations in order to perform on-the-fly (or *on-line*) generalisation. The off-line strategy is simpler to implement, but the on-line strategy could potentially discover more opportunities for fusion. In particular, the on-line strategy does not require

renaming to be performed during fusion (unless generalisation is forced). As a result, it could sometimes perform both fusion and the elimination of multiple traversals of common variables. As an example, consider the program:

```
---main1(xs)                  <= zip(double(xs),double(xs));
---zip(cons(x,xs),cons(y,ys)) <= cons((x,y),zip(xs,ys));
---zip(xs,ys)                 <= nil;
```

The supercompiler could transform the *main1* function to the following equivalent where both intermediate data structures and multiple traversal of the variable *xs* are eliminated:

```
---main1(cons(x,xs))          <= cons((2*x,2*x),main1(xs));
---main1(xs)                  <= nil;
```

Our fusion method is *not* able to perform such a combined transformation because it is used *solely* for the elimination of intermediate data structures. (A different transformation method, called tupling (Chin, 1993) will be needed to eliminate multiple traversals of data structures.) However, the power of the on-line strategy is dependent on the generalisation technique used. The on-line generalisation proposed by Turchin (1988) maintains a sequence function calls encountered for each step of supercompilation. This sequence of encountered calls, say $g_1 g_2 ... g_n$, corresponds to the nesting of g-type calls of the form, $g_1(g_2(\cdots g_n(c,..)\cdots,..),..)$, where c is a call that is about to be unfolded by normal-order evaluation. Turchin showed that supercompilation will terminate if generalisation is performed whenever the current sequence of g-type calls encountered is a sub-sequence of some previous configuration. However, this strategy may generalise pre-maturely for certain types of programs. In particular, functions with swapping parameters may not be successfully fused by the on-line strategy. As an example, consider the program:

```
---main2(xs,ys,zs)            <= fswap(xs,p(ys),q(zs));
---fswap(nil,ys,zs)           <= ..ys..zs...;
---fswap(cons(x,xs),ys,zs)    <= cons(x,fswap(xs,zs,ys));
```

The on-line generalisation strategy proposed in Turchin, (1988) is not able to fuse the RHS of *main2*. The reason for this is that a pre-mature generalisation has occured because the sub-sequence of encountered calls does not include p and q. In contrast, our off-line strategy will detect that the two swapping parameters are non-accumulating and could therefore contain arguments that are safe to fuse. Hence, Turchin's on-line generalisation strategy is better in some aspect but worse in others, when compared to our off-line generalisation strategy.

Recently, Richard Waters (1991) has proposed a transformation technique for fusing expressions using *series* (various sequences, e.g. vectors, lists, which may be unbounded) so that unnecessary intermediate series data structures could be eliminated. He identified a sub-class of expressions which could be transformed, namely those which are *statically analyzable*, *pre-order* and *on-line cyclic*. Water's technique cannot handle tree-like structures (including sequence inside sequence). However, the on-line cycle restriction allows fusion of functions which take multiple inputs originating from common variables (thus, forming cycles) with the on-line characteristic (lockstep production of one output for every input consumed). This has the same effect as fusing *multiple-inputs* functions composed with a set of

https://doi.org/10.1017/S0956796800001179 Published online by Cambridge University Press

synchronizable producer calls. The pre-order restriction is more limiting that the safe-unsafe criteria of our generalised deforestation, but the on-line cyclic restriction is something extra. An example of this is the expression *zip(double(xs),double(xs))* of function *main1* above where all the three function calls used are on-line (i.e. produces one output for every input consumed) and the common *xs* forms a cycle. Direct fusion of such expressions has the same effect as combining the tupling and fusion transformation together (see Chin & Khoo, 1993).

Another related area is partial evaluation (Consel, 1990; Bjorner *et al.*, 1988; Jones *et al.*, 1989). The primary mechanism used in partial evaluation is the specialisation of function calls which have some or all of their arguments *known* (or *partially known*). (A known argument is a grounded term without any free variables.) Such calls can be transformed to equivalent but more efficient functions which exploit the context of their known arguments. Traditionally, an analysis technique called *binding-time analysis* (Jones, 1988) has been used to analyse functions to find out which of the arguments are known or unknown (also called *static* vs. *dynamic*). However, this analysis cannot be used to guarantee the termination of the partial evaluation process itself. Lately, Holst (1991) has proposed an additional analysis, called *finiteness analysis* to determine which known arguments can preserve the termination property of partial evaluation. This analysis is used to identify *in-situ non-increasing* parameters which can be viewed as a *semantic* derivative of our syntactic non-accumulating criterion. Presently, Holst's analysis is only applicable to strict, first-order functional languages. In contrast, our safe fusion scheme (using syntactic analysis) is applicable to lazy, higher-order languages. It will be interesting to see if similar semantic analysis can be formulated for lazy and/or higher-order languages. Whilst partial evaluation specialises the known or partially known arguments, the deforestation technique appears to be more general as it also specialises *symbolic* arguments which are unknown. (Symbolic arguments refer to arguments of non-trivial expressions which may have free variables.) However, partial evaluation also employs reduction and simplification techniques which are currently ignored by deforestation.

Recently, Proietti & Pettorossi (1991) formulated a transformation procedure, called the elimination procedure (in short), for eliminating unnecessary variables from logic programs. This procedure is based on an iteration of unfold, define followed by fold steps and uses a syntactic criterion for clauses, called *non-ascending criterion.* This criterion is very similar to our *linear*, *non-accumulating* parameter criterion but was independently discovered at about the same time (for different languages). The *non-ascending* criterion is applied to the whole clause rather than to individual parameters. Those clauses which do not satisfy this criterion could be made to do so by parameter extraction/generalisation.

One interesting aspect of the elimination procedure is that it does not need a separate analysis for producers. This is so because in logic programs, parameters of relations could be used for both input and output. However, though relations are more general than functions, certain transformations are not possible without functional mode analysis. For example, the procedure to eliminate redundant calls (for tupling transformation) in Proietti & Pettorossi (1991) requires this analysis.

10 Conclusion

The basic idea of annotating terms, in order to distinguish between sub-terms which can be eliminated from those sub-terms which cannot, is essentially similar to Wadler's blazing technique. Our main contribution is the extension of deforestation so that it is applicable to a much wider range of expressions. This extension is made possible by the adoption of the producer-consumer view of functions and the discovery that syntactic properties could be used to classify functions and their parameters into either safe or unsafe producers and consumers. This discovery led to the use of a double annotation scheme to identify safe sub-terms which could be fused. It allowed us to apply deforestation to all first-order programs. In addition, with the help of another transformation technique, called higher-order removal, we are able to extend deforestation to all well-typed higher-order programs, and thus have more intermediate terms eliminated. Furthermore, we are also able to improve deforestation even further by augmenting the syntactic analysis with additional improvement techniques. These techniques help us obtain more cases of safe consumers and safe producers.

Like Wadler's original deforestation algorithms, our extension remains fully automatic and is guaranteed to terminate. As a result, it is very suitable for use in the optimisation phase of any purely functional language compiler.

Much further work remains to be done in the area of safe fusion. One immediate task, currently being undertaken, is to implement the above generalised deforestation algorithm into a functional language compiler. When completed, we shall be able to measure the improvements provided by this optimisation in terms of average reductions in processor and storage utilization. We shall also be measuring the cost of this optimization, in terms of time taken to perform the transformation together with new code sizes of transformed programs. Code size is of particular concern because it is theoretically possible to get exponential increase with certain types of programs. However, we suspect that this would only happen with contrived programs. More formal analysis to identify speed-up achievable together with time complexity of the transformation algorithm could also be done.

Another possible avenue for future work is to enhance the safe fusion method even further. Two immediate areas which could be looked at are better analysis techniques (perhaps using semantic-based ones) together with methods for combining fusion with tupling. These improvements are likely to make safe fusion even more attractive.

Acknowledgements

This work owed much to John Darlington who supervised my PhD at Imperial College. Also, thanks to Phil Wadler whose extremely clear, simple but purposeful papers have inspired much of the work done here. Khoo Siau Cheng and Morten Sorgensen read this draft and provided some very helpful comments. Thanks also due to the referees whose sharp comments have helped to improve both the format and contents of this paper.

References

Augustsson, L (1985) Compiling pattern-matching. In: *Conference on Functional Programming and Computer Architecture (LNCS 201, ed Jouannaud)*, pp. 368–381, Nancy, France.

Burstall, R. R & Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1): 44–67, January.

Bjorner, D, Ershov, A. P. & Jones, N. D. (1988) *Workshop on Partial Evaluation and Mixed Computations.* Gl Avarnes, Denmark. North-Holland.

Chin, W.-N. & Darlington, J. (1992) Higher-order removal transformation technique for functional programs. In: *15th Australian Computer Science Conf.*, Hobart, Tasmania. *Australian CS Comm.*, **14**1:181–194.)

Chin, W.-N. (1990) *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, University of London, March.

Chin, W.-N. (1991) Generalising deforestation for all first-order functional programs. In: *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages, BIGRE 74*, pp. 173–181, Bordeaux, France, October.

Chin, W.-N. (1992a) Safe fusion of functional expressions. In: *7th ACM Lisp and Functional Programming Conf.*, pp. 11–20, San Francisco, CA, June.

Chin, W.-N. (1992b) Synthesizing parallel lemma. In: *Proc. JSPS Seminar on Parallel Programming Systems*, World Scientific Publishing, pp. 201–217, Tokyo, Japan, May.

Chin, W.-N. (1993) Towards an automated tupling strategy. In: *3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, pp. 119–132, Copenhagen, Denmark, June.

Chin, W.-N. & Khoo, S.-C. (1993) Tupling functions with multiple recursion parameters. In: *3rd Int. Workshop on Static Analysis: LNCS 724*, Springer-Verlag, pp. 124–140, Padova, Italy, September.

Consel, C (1990) Binding time analysis for higher-order untyped functional languages. In: *6th ACM Conf. on Lisp and Functional Programming*, pp. 264–272, June.

Feather, M. S. (1982) A system for assisting program transformation. *ACM Trans. Programming Languages & Systems*, **4**(1): 1–20, January.

Ferguson, A. B. &Wadler, P. (1988) When will deforestation stop? In: *Proc. Glasgow Workshop on Functional Programming* (available as Research Report 89/R4, Glasgow University), pp. 39–56, Rothesay, Isle of Bute, August.

Holst, C. K. (1991) Finiteness analysis. In: *5th ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 473–495, Cambridge, MA, August.

Jones, N. D. (1988) Automatic program specialisation: A re-examination from basic principles. In: *Workshop on Partial Evaluation and Mixed Computations*, pp. 225–282, Gl Avarnes, Denmark. North-Holland.

Jones, N. D., Sestoft, P. & Sondergaard, H. (1989) An experiment in partial evaluation: the generation of a compiler generator. *J. LISP and Symbolic Computation*, **2**(1): 9–50.

Milner, R. (1978) A theory of type polymorphism. *J. Comp. & Syst. Sci.*, 348–375.

Proietti, M. & Pettorossi, A. (1991) Unfolding - definition - folding, in this order for avoiding unnecessary variables in logic programs. In: *Proc. PLILP'91; LNCS 528*, pp. 347–358, Passau, Germany, August.

Runciman, C, Firth, M. & Jagger, N. (1989) Transformation in a non-strict language: An approach to instantiation. In: *Glasgow Functional Programming Workshop*, August.

Sands, D. (1990) Complexity analysis for a lazy higher-order language. In: *3rd Euro. Symposium on Programming: LNCS 432*, pp. 361–376, Copenhagen, Denmark, May.

Sestoft, P. (1988) Automatic call unfolding in a partial evaluator. In: *Workshop on Partial Evaluation and Mixed Computations*, pp. 485–506, Gl Avarnes, Denmark. North-Holland.

Turchin, V. F. (1986) The concept of a supercompiler. *ACM Trans. Programming Languages & Systems*, **8**(3): 90–121, July.

Turchin, V. F. (1988) The algorithm of generalisation in the supercompiler. In: *Workshop on Partial Evaluation and Mixed Computations*, pp. 531–549, Gl Avarnes, Denmark. North-Holland.

Wadler, P. (1984) Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In: *ACM Symposium on Lisp and Functional Programming*, pp. 45–52, Austin, TX, August.

Wadler, P. (1985) Listlessness is better than laziness II: Composing listless functions. In: *Workshop on Programs as Data Objects*, pp. 282–305, Springer-Verlag, New York.

Wadler, P. (1987) Efficient compilation of pattern-matching. In: S. Peyton-Jones (ed.), *The Implementation of Functional Programming Languages*. Prentice-Hall.

Wadler, P. (1988) Deforestation: Transforming programs to eliminate trees. In: *Euro. Symposium on Programming*, pp 344–358, Nancy, France, March.

Waters, R. C. (1991) Automatic transformation of series expressions into loops. *ACM Trans. Programming Languages & Systems*, **13**(1): 52–98, January.