Safe Fusion of Functional Expressions

Wei-Ngan CHIN
Dept of Information Systems & Computer Science
National University of Singapore

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, Wadler proposed a transformation algorithm, called *deforestation*, which could automatically *fuse* certain composed expressions together in order to eliminate intermediate tree-like data structures. However, his technique is only applicable to a *subset* of first-order expressions.

This paper will generalise the deforestation technique to make it applicable to **all** first-order and higher-order functional programs. Our generalisation is made possible by the adoption of a model for *safe fusion* which views each function as a producer and its parameters as consumers. Through this model, static 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 as a set of syntax-directed rewrite rules, illustrate it with examples, and provide an outline of its termination proof.

1 Introduction

Consider an expression $p(q(v_1), r(v_2, s(v_3), t(v_4)))$, call it e, where v_1, v_2, v_3, v_4 are variables and p, q, r, s, t are user defined functions. In this expression, v_1, v_2, v_3, v_4, are inputs of e, while p, q, r, s, t are simpler functions decomposed from the main task of e. This modular expression may be easier to construct, with some of the functions re-used from other programs. However, sub-terms of e, like $q(v_1)$ or $r(v_2, s(v_3), t(v_4))$ or $s(v_3)$ or $t(v_4)$, 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 [BD77] to fuse e into a piece of more tightly woven code, without 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 so that the original nesting of function calls disappears, as shown below[1]:

$$--- f_1(v_1, v_2, v_3, v_4) \Leftarrow p(q(v_1), r(v_2, s(v_3), t(v_4)))$$
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 not suitable for fusion, a simple technique, called *parameter generalisation*, can be used to abstract away the unsuitable sub-terms before fusion. For example, if the sub-term $r(v_2, s(v_3), t(v_4))$ cannot be fused with p of e, then a new function, f_2, can be defined with the unsuitable sub-term generalised out using a new parameter variable, w. This can then be transformed, as follows:

$$--- f_2(v_1, w) \Leftarrow p(q(v_1), w)$$
transforms to
$\Leftarrow$ *..equivalent expression without above nested functions..*

With the above function, the expression e is now equivalent to $f_2(v_1, r(v_2, s(v_3), t(v_4)))$, where further opportunities for fusion may be found by similar analysis and transformation of $r(v_2, s(v_3), t(v_4))$. Thus, fusion can be selectively applied, as long as sub-terms which are unsafe to fuse can be identified.

This paper grew out of Chapters 3 and 4 of the author's PhD thesis [Chi90] and has been inspired primarily from Phil Wadler's work on deforestation [Wad88]. A preliminary version of this work has appeared in [Chi91] where only first-order functional programs are considered.

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 the *producer-consumer* model of functions and propose a new annotation scheme based on safe/unsafe producers and consumers. (An earlier annotation scheme, proposed in [Chi90, Chi91], is based solely on consumers. This simpler scheme works by changing all unsafe producers to pseudo-safe. However, it is not suitable for further extension of deforestation to include the use of laws/axioms.) Section 4 presents the generalised deforestation algorithm for first-order programs, together with formal definitions of safe/unsafe producers and consumers,

[1] expressed in the Hope language with equations of the form $--- LHS \Leftarrow RHS$

Author's Address: Dept of IS & CS, National University of Singapore, Kent Ridge, Singapore 0511, e-mail: chinwn@iscs.nus.sg

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0011...$1.50

and a termination proof. Section 5 outlines our extension of deforestation to a full higher-order functional language. Section 6 shows how some potential problems, which arise from the syntactic properties used to classify safe/unsafe sub-terms, are overcome. Section 7 briefly outlines a further extension of deforestation using laws/axioms. Section 8 describes other related work. Section 9 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 pure functional expressions.

2 Wadler's Deforestation

The deforestation technique was first proposed in [Wad88] 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 to illustrate the technique contains functions of the form:

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

with its RHS term, tf, described by:

$$t ::= v \mid c(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) \mid case\ t\ in\ p_1 \Rightarrow t_1;..;p_n \Rightarrow t_n\ end$$
$$p ::= c(v_1,\ldots,v_j)$$

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 [Aug85, Wad87] to translate any expression with *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 [FW88] where each *case* construct is replaced by an equivalent g-type pattern-matching function of the form:

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

This new language helps to simplify the deforestation algorithm. Its adoption 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. A *pure treeless* function is a function whose RHS terms satisfy the grammar form:

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

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

Terms of this grammar form are known as *pure treeless* terms because they do not contain nested applications of functions. Hence, they are free of all intermediate data structures, including tree-like ones. An example of pure treeless function is:

```
---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 Wadler's algorithm to a pure treeless expression, *apptree(xs,ys,zs)*, as shown in Figure 1.

Blazed deforestation is an extension of pure deforestation to cater for functions which are not pure-treeless because of atomic-type sub-terms (e.g. integer, char). Two examples are:

```
---double(nil)          <= nil;
---double(cons(a,as)) <= cons(2*a,double(as));
---sum(nil)            <= 0;
---sum(cons(a,as))     <= a+sum(as);
```

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 $\ominus$, and each tree-type sub-term with $\oplus$. Periodically, before each fusion sequence, all sub-terms annotated as $\ominus$ 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 annotation scheme is very simple but it cannot be used to extend deforestation to all first-order expressions.

In the next section, we propose a new model for safe fusion which can differentiate *more accurately* sub-terms that can be fused, from those that 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*.

3.1 Conditions for Safe Fusion

Consider a nested application of two functions: $p(q(x))$.

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 *safe* and *effective* fusion[2]. A nested application is said to be *safely* fused if the transformation sequence which follows does not go into a loop and there is no loss of efficiency experienced by the transformed expression (when compared with the original expression). A nested application is *effectively* fused if the need for its intermediate data disappears and there is a gain in efficiency.

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

[2] My thanks to Phil Wadler for pointing out the need to differentiate between the notions of *safe*, as opposed to, *real/effective* fusion

```
append(append(xs,ys),zs)      ⇒ appthree(xs,ys,zs)                    ;fold appthree
Define & transform function, appthree
---appthree(xs,ys,zs)         ⇐ append(append(xs,ys),zs)              ;unfold append
Above unfold (with instantiation) results in two equations:
---appthree(nil,ys,zs)        ⇐ append(ys,zs)
---appthree(cons(x,xs),ys,zs) ⇐ append(cons(x,append(xs,ys)),zs) ;unfold append
                              ⇐ cons(x,append(append(xs,ys),zs)) ;fold appthree
                              ⇐ cons(x,appthree(xs,ys,zs))
```

Figure 1: Transformation Steps Applied by Pure Deforestation

amenability to safe fusion. The static properties used to determine whether a given producer or consumer is safe (or unsafe) will be given later 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 this model, the pure treeless functions from Wadler's Pure Deforestation can be viewed as functions which are **both** safe producers and safe consumers, with any expression composed from them being totally fusable. Our model is more general. This is because it can tolerate *q* as an unsafe consumer and/or *p* as an unsafe producer for *p(q(x))* to be still fusable. (Also, blazed deforestation treats all atomic-type sub-terms as unsafe, but this does not imply that all tree-like sub-terms are safe.)

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 shall 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 *static* property of the sub-terms, and can be formally described by the following annotated grammar:

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

Consumer-based annotation is a *dynamic* property of the sub-terms because it is dependent on the parameter positions that the sub-terms are presently 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 $\ominus_c$. Other sub-terms, *not* lying in the parameters of *f* or *g*-type functions (e.g. arguments of constructors), will not be provided with any consumer-based annotations.

With this scheme, each sub-term is either annotated once or twice. Sub-terms which are also 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 combined annotations:

$\oplus_p$ 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 properties - static ones determinable at compile-time - for this double annotation scheme. We shall also present the transformation algorithm for safe fusion. Initially, we consider only first-order programs. Later, we extend the transformation algorithm to all higher-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 our result as a gradual extension of Wadler's deforestation algorithm through three steps. Firstly, *unsafe consumers* are handled, then *unsafe producers* are handled, followed by *first-order functions.*

4.1 Step 1 - Handling Unsafe Consumers

Parameters of functions are classified as either safe or unsafe consumers of data. A parameter is classified as a **safe consumer** if it is *linear* and *non-accumulating*; otherwise it is classified as an **unsafe consumer.** A parameter is **linear** if its variable(s) occurs only once in each RHS term of its function. This criterion helps avoid loss of efficiency by not duplicating large non-linear arguments during deforestation. Also, roughly speaking, a parameter is **non-accumulating** if the corresponding arguments in successive recursive calls of its function will not get (syntactically) bigger during transformation. This criterion is needed because accumulating parameters may result in successively larger expressions, which can cause non-termination in transformation. Non-accumulating parameters do not cause this problem.

Consider the function, *rev_it*:

```
---rev_it(nil,w)        ⇐ w;
---rev_it(cons(a,as),w) ⇐ rev_it(as,cons(a,w));
```

which cannot be handled by either pure of blazed deforestation. The first parameter of this function is a safe consumer because it is *linear* and *non-accumulating.* (It is *non-accumulating* because the argument, *as*, of its recursive call is not bigger than the original parameter *cons(a,as).*) However, the second parameter is *accumulating* because the

recursive call, *rev_it(as,cons(a,w))*, takes a larger sub-term, *cons(a,w)*, as its argument. As a consequence, the expression, *rev_it(double(as),w)* can be fused but not *rev_it(as,double(w))*.

Formally, the *non-accumulating* criterion can be defined as follows:

> 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 definition:
> $$--- h_i(v_1, \ldots, v_j, \ldots, v_n) \Leftarrow th_i$$
> 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, $h_1, \ldots, h_k$, have the jth-argument, t_j, as a *variable* or a *constant term.*
>
> Also, all parameters of *non-recursive* functions are trivially regarded as *non-accumulating*, since there are *no* recursive calls in their definitions.

Hence, linear and non-accumulating parameters, which are regarded as safe consumers, will have their arguments (sub-terms) annotated with $\oplus_c$. Correspondingly, unsafe parameters will have their arguments annotated with $\ominus_c$. Now, if we initially assume that all functions used are *safe producers*, then whether a sub-term is safe or unsafe is dependent solely on its consumer-based annotation.

With the above assumption, the generalised transformation algorithm that is capable of handling unsafe consumers need only consists of six syntax-directed rules, as given in Figure 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 in which there may be a nesting of function calls that could be fused. We use a context notation, $\cdots C \cdots$, to identify an inner call, C, which is about to be unfolded by normal-order reduction. This inner call lies within a nesting of $\oplus$ pattern-matching arguments of g-type calls, as specified by the grammar for this context notation:

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

Four rules are used to transform expressions of the form $\cdots C \cdots$. Depending on appropriate conditions, these rules use one of four different steps to transform. They are either (i) a *direct unfold* ($\mathcal{T}3a, 4a$), (ii) a *define followed by an unfold* ($\mathcal{T}6b, 4c, 5b, 6b$), (iii) a *fold* step ($\mathcal{T}4b, 5a, 6a$), or (iv) a *skip over* step ($\mathcal{T}4d, 5c, 6c$), as shown in Figure 2.

A *fold* step is taken if a previously defined function matches the current expression. All previously defined functions are stored in a data structure, called *def_set*. A *direct unfold* step is taken if the inner call is of the form $g(c_i(t'_1, \ldots, t'_j)^{\oplus}, t_1, \ldots, t_n)$ or $f(t_1, \ldots, t_n)$ where f is non-recursive and only variables appear in unsafe consumers. (With only variables in unsafe consumers, there is no possibility of large arguments being duplicated by direct unfolding.) A *skip over* step is taken if the outermost function call contains no safe sub-terms to remove. If none of these situations are met, then a *define & unfold* step is taken. In this 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 other variables not extracted are *renamed*. Such a procedure ensures that (i) no unsafe sub-terms are fused, and (ii) the expression is linear. This procedure helps avoid successively larger expressions from being formed during deforestation. This, in turn, ensures transformation algorithm's termination.

A procedure, called $\mathcal{G}$, is used to *generalise out* unsafe sub-terms and *rename* all variable occurrences which do not lie in unsafe sub-terms. Given an expression, t, the $\mathcal{G}$ procedure will return a tuple of five items:

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

where $t^{\triangle}$ is the generalised 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 are not part of 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 *rev_it(double(as)$^{\oplus}$,double(w)$^{\ominus}$)*. This expression can be transformed by the new algorithm, as shown in Figure 3. (Notice that primitive functions, like $*$, are simply regarded as unsafe producers with unsafe consumers.)

An important point to note is that the above algorithm removes (either eliminates or transfers away) *all* safe sub-terms. This can be verified by showing that expressions which result from the algorithm will have a form, known as *extended-treeless*, or *e-treeless*, form satisfying the grammar below:

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

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 its definition's RHS terms are *e-treeless*.

The above e-treeless form requires that functions used are also e-treeless. This can be achieved because we use only safe producers and these are e-treeless. In fact, the e-treeless grammar form will be used to help distinguish between safe and unsafe producers in the next sub-section. A closer look at the e-treeless form reveals that only nested constructor terms are produced but never nested function calls (unless they are in unsafe consumers). Hence, when such functions are unfolded, their intermediate constructor terms can always be safely consumed by outer g-type function calls. This helps to prevent successively larger expressions (which cause non-termination) from being formed during fusion. Thus, if only e-treeless functions are used in the expression to be transformed, it can be proved that the above transformation algorithm terminates (see Section 4.4 for a proof outline).

4.2 Step 2 - Handling Unsafe Producers

The definition of safe/unsafe producers is based on the e-treeless form. However, there are some subtle issues involved because whether an expression is e-treeless or not, is itself dependent on how its functions are classified.

Formally, a function[3] p is a **safe producer** if its RHS term(s) satisfies the e-treeless form, when p has been *regarded* as a safe producer.

[3] We actually consider each set of mutually recursive functions, which may occassionally consists of a single function

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

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

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

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

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

DEFINE & ADD TO *def_set*

$f\_new(vn_1,\ldots,vn_k,ve_1,\ldots,ve_s) \Leftarrow \cdots g(c_i(t'^{\triangle}_1,\ldots,t'^{\triangle}_j),t^{\triangle}_1,\ldots,t^{\triangle}_n)\cdots^{\triangle}$

UNFOLD

$\Leftarrow \mathcal{T}[\cdots tg_i[t'^{\triangle}_1/v'_1,\ldots,t'^{\triangle}_j/v'_j,t^{\triangle}_1/v_1,\ldots,t^{\triangle}_n/v_n]\cdots^{\triangle}]$

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

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

a) IF f is not recursive and $\forall a \in 1\ldots n, {t_a}^{\ominus}$ IS A VARIABLE (*direct unfold*)
$\Rightarrow \mathcal{T}[\cdots tf[t_1/v_1,\ldots,t_n/v_n]\cdots]$

b) IF $f\_old(vn_1,\ldots,vn_k,ve_1,\ldots,ve_s) \Leftarrow \cdots f(t^{\triangle}_1,\ldots,t^{\triangle}_n)\cdots^{\triangle} \in def\_set$ (*fold*)
$\Rightarrow f\_old(vo_1,\ldots,vo_k,\mathcal{T}[te_1],\ldots,\mathcal{T}[te_s])$

WHERE $(\cdots f(t^{\triangle}_1,\ldots,t^{\triangle}_n)\cdots^{\triangle}, te_1,\ldots,te_s,ve_1,\ldots,ve_s, vo_1,\ldots,vo_k,vn_1,\ldots,vn_k) == \mathcal{G}[\cdots f(t_1,\ldots,t_n)\cdots]$

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

d) IF $\cdots f(t_1,\ldots,t_n)\cdots = f(t_1,\ldots,t_n)$ and $\forall a \in 1\ldots n, {t_a}^{\oplus}$ IS A VARIABLE (*skip over*)
$\Rightarrow f(\mathcal{T}[t_1],\ldots,\mathcal{T}[t_n])$

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

a) SIMILAR TO (*fold*) of $\mathcal{T}4b$

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

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

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

a) SIMILAR TO (*fold*) OF $\mathcal{T}4b$

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

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

Figure 2: Transformation Algorithm for Safe Fusion of First-Order Expressions

```
T[rev_it(double(as),double(w))]                                  ; T5b define revdb
                    ⇒ revdb(as,T[double(w)])                     ; T5c 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)]                          ; T3a unfold rev_it
                    ⇐ T[ws]                                      ; T1 skip
                    ⇐ ws
---revdb(cons(a,as),ws) ⇐ T[rev_it(cons(2*a,double(as)),ws)]     ; T3a unfold rev_it
                    ⇐ T[rev_it(double(as),cons(2*a,ws))]         ; T5a fold revdb
                    ⇐ revdb(T[as],T[cons(2*a,ws)])               ; T1,T2 skip
                    ⇐ revdb(as,cons(T[2*a],T[ws]))               ; T4d,T1 skip
                    ⇐ revdb(as,cons(T[2]*T[a],ws))               ; T2,T1 skip
                    ⇐ revdb(as,cons(2*a,ws))
```

Figure 3: An Application of Generalised Deforestation Algorithm $\mathcal{T}$

Not all functions are safe producers. An example is the function, *rev_flatten*:

$$\begin{array}{l} \text{---}rev\_flatten(nil) \Leftarrow nil; \\ \text{---}rev\_flatten(cons(as,ass)) \Leftarrow append(rev\_flatten(ass)^{\oplus_c}, as^{\oplus_c}); \end{array}$$

This function is *not* e-treeless when it is regarded as a safe producer. This is because *rev_flatten(ass)*, its recursive call, is presently lying in a safe consumer of the *append* function. In contrast, e-treeless form has only variables (or unsafe producers) in the positions of safe consumers. However, if *rev_flatten* is regarded as an unsafe producer, then its RHS terms would be e-treeless. Note that this only happens after *rev_flatten* has been regarded as an unsafe producer.

Through this circular definition, unsafe producers can also be made e-treeless, and thus be handled by our generalised deforestation algorithm. The way to handle unsafe producers is *never* to unfold them, as producers, during fusion. The reason is that unsafe producers can be seen as *producing* or *generating* function terms, in addition to constructor terms. For example, the *rev_flatten* function can be viewed as a recursive function which produces *append* function calls. These generated functions cannot be readily consumed by the pattern-matching equations of outer g-type functions.

In the combined annotation scheme, all unsafe producers in both safe or unsafe consumers are generalised out prior to each fusion sequence. This (conservatively) prevents all unsafe producers from being unfolded, as producers. Notice that our algorithm does not prevent unsafe producers (like *rev_flatten*) from being unfolded in fusion as *safe consumers*.

4.3 Step 3 - Handling Each Function

Presently, the transformation algorithm is formulated to transform *expressions* which are composed from e-treeless 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 each RHS term of first-order functions, but two additional issues need to be considered.

Firstly, our algorithm must be applied in a **bottoms-up order** where 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. Bottoms-up order ensures that child functions are always converted to e-treeless form before their parent functions (as required by $\mathcal{T}$).

Secondly, each set of sibling (mutually recursive) functions must be simultaneously transformed and regarded as *potentially* unsafe producers and unsafe consumers. As potentially unsafe functions, the sibling calls will **not** be unfolded during their functions' transformation. After the set of sibling functions have been transformed to e-treeless form, we can use the static analyses of Section 4.1 and 4.2 to determine if the transformed functions are safe or unsafe producers, and their parameters are safe or unsafe consumers. It is better to apply static analyses after transformations because syntactic properties often change (from unsafe to safe) during transformation.

4.4 Outline of Termination Proof

In this section, we outline the termination proof of the generalised deforestation algorithm for first-order programs. A more detailed proof is available in [Chi90] for the interested reader. There are two steps in our proof.

Firstly, we need to show that the number of *define steps* by $\mathcal{T}4c, 5b, 6b$ in any application of our algorithm is finite. This can be proved by showing that there exists an upper bound on the size of expressions used for defining new functions. The presence of an upper bound indicates that there could only be finitely many different new functions (formed from a fix set of function and constructor symbols) that could be introduced. As a result, the number of *define steps* will be finite because expressions which re-occur will be folded, rather than result in another new function definition.

Secondly, we need to show that the number of the other steps between each pair of *define steps* is finite. This can be proved by showing that there is a well-founded decreasing measure among the *non-define steps*. These non-define steps include $\mathcal{T}2, 3, 4a, 4b, 4d, 5a, 5c, 6a, 6c$.

With these two proof steps, the proposed algorithm is terminating because a finite number of *non-define steps*, between a finite number of *define steps*, implies that the total number of steps needed to transform each expression is also finite.

5 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 modular programs 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** [Chi90, CD92], which 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 than the full higher-order expression form, with the residual features easily handled.

Consider the following extended grammar for higher-order expressions:

$$\begin{array}{lll} t & ::= v \mid c(t_1, \ldots, t_n) & \mid t(t_1, \ldots, t_n) \mid f \mid g \\ & & \mid lambda\ (v_1, \ldots, v_n) \Rightarrow t\ end \end{array}$$

Compared to the grammar for first-order expressions, some new (higher-order) features which need to be taken care of includes, 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* which are non-accumulating.

Curried applications are all those 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 $\mathcal{R}$ [Chi90], which has been proved terminating, as long as well-typed higher-order programs are used.

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

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

WHERE $v_{m+1}^{\oplus_h},\ldots,v_n^{\oplus_h}$ are the non-accumulating function-type arguments annotated with $\oplus_h$.

Some residual higher-order sub-terms may remain in expressions of the above restricted form. However, they are easier to consider (for extending the deforestation algorithm) than the full higher-order form.

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 have either function-type sub-terms or variable applications as arguments. As these higher-order arguments are not required to be removed by deforestation, they can be marked as either unsafe consumers or unsafe producers.

Secondly, four additional $\mathcal{T}$ rules (shown in Figure 4 as $\mathcal{T}7-10$) are needed to directly handle the residual higher-order features. These rules simply skip over the residual higher-order features.

Lastly, during deforestation, it is possible for new higher-order expressions to re-appear. These occur as side-effects of eliminating constructor terms with function-type arguments. The rule for eliminating new higher-order expressions which re-appear is $\mathcal{T}11$ of Figure 4.

The above set of new rules have been shown in [Chi90] not to affect the termination property of $\mathcal{T}$. As an illustration of this extended set of $\mathcal{T}$ rules, the following higher-order program:

```
---main(x,y)              <= map_ho(add_five(x),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));
```

can be transformed to its equivalent first-order program:

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

Wadler has also considered higher-order extension for deforestation in [Wad88]. However, his solution is not general, as it relies on a restricted higher-order facility - called *higher-order macro* - whose use can always be converted to first-order equivalent before deforestation. Higher-order macros essentially correspond to higher-order functions with *fixed* function-type parameters (i.e. do not change across recursion). They can neither return function-type results, nor support constructor terms with function-type arguments. The advantage of the higher-order macro scheme is its simplicity, but it requires the user to adopt a restricted higher-order language.

6 Pseudo Safe/Unsafe Consumers and Producers

Our generalisation of the deforestation technique 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!

In fact, in earlier work by the author [Chi90, Chi91], 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. Similarly, *let* constructs can also be used to (trivially) convert non-linear and/or accumulating parameters to linear and non-accumulating parameters to obtain pseudo-safe consumers. These pseudo-safe producers and consumers do not result in real fusion, as their corresponding pseudo-safe sub-terms are merely *transferred* to the *let* constructs, rather than eliminated, during transformation. They are harmless because they do not affect termination, nor result in loss of efficiency. However, a minor problem is that they tend to result in larger transformed programs.

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, $---id(x) \Leftarrow x$, to *wrap* around appropriate sub-terms; so that safe producers become pseudo-unsafe, or non-accumulating parameters become accumulating. Fortunately, our generalised 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, using g-type functions, such as:

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

can cause problem to our algorithm. This is because our present algorithm do not perform direct unfolds on g-type functions. Given the rather contrived technique used to deceive the algorithm, this is a shortcoming we could tolerate.

7 Further Improvements

Further improvements to our generalised deforestation are possible. These improvements can help remove more intermediate data structures from user programs. Two possible improvements are briefly proposed below.

Firstly, some of the non-linear pattern-matching parameters could be *linearised* with the help of the *let* construct. This can be used to convert unsafe consumers to equivalent safe ones. Consider the function:

```
---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 which can be used to make such a pattern-matching parameter safe (or linear) is to abstract out the non-linear auxiliary variable, as follows:

```
---square((cons(x,xs)^⊕c) <= let v=x in cons(v*v,square(xs));
```

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

Figure 4: Additional $\mathcal{T}$ rules for fusing higher-order programs

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.

A second possible way to improve deforestation 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 *unsafe producers* to be successfully fused as producers. Consider the program:

```
---sizet(t^⊕c)              ⇐ length(flatten(t));
---length(nil^⊕c)           ⇐ 0;
---length(cons(x,xs)^⊕c)    ⇐ 1+length(xs));
---flatten(leaf(a)^⊕c)      ⇐ cons(a,nil);
---flatten(node(lt,rt)^⊕c)  ⇐ append(flatten(lt),flatten(rt);
```

Presently, the expression *length(flatten(t))* cannot be fused because the inner function *flatten* is an unsafe producer. In fact, the main reason why *flatten* is considered as an unsafe producer is that it produces *append* function calls (whilst safe producers produces only constructors). These produced function calls cannot be safely consumed using the pattern-matching equations of *length*. Note that pattern-matching equations can consume *constructors* (from safe producers) but not *functions* (from unsafe producers). 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 viewed as an equation of *length* whose linear and non-accumulating parameter, *append(xs,ys)*, is a safe consumer of *append* function calls. Consequently. it can be used to successfully fuse expressions which contain unsafe producers of *append* calls. In particular, this law can be used to help transform function *sizet* to the following function (without its unnecessary intermediate data structure):

```
---sizet(leaf(a))      ⇐ 1+0;
---sizet(node(lt,rt)) ⇐ sizet(lt)+sizet(rt);
```

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 [Chi90] for a method to synthesize distributive laws). Given that these laws can be made available, there is still a need to integrate their use into the generalised deforestation algorithm. Further work is in progress to provide this integration. In this paper, we shall briefly suggest the main changes needed.

Initially, the double annotation scheme has to be enhanced to indicate what functions (if any) can be consumed by parameters (through laws), and what functions are produced by the unsafe producers. With this enhancement, all those unsafe producers which lie in consumers, which can consume their produced functions, will be classified as safe sub-terms. These new safe sub-terms are fused with the help of laws. The transformation algorithm must be accordingly modified to apply the appropriate laws when fusing these new category of safe sub-terms.

8 Other Related Work

Over the years, there have been a number of different proposals for techniques which can remove unnecessary intermediate sub-terms 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 and Darlington [BD77] where *loop combination* (fusion) of programs was illustrated 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 the time) are largely handcrafted. Subsequently, Martin Feather [Fea82] build a system, called ZAP, which was able to derive low-level unfold/fold transformation sequences from higher-level pattern-directed transformation 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 the transformations needed by the *tupling*, *generalisation* and *composition* (fusion) tactics. One large example illustrated was the transformation of a multi-pass compiler into a two-pass compiler for a toy language. However, pattern-directed transformations have to be individually specified. Our work is based on the same unfold/fold framework but we have now developed a transformation algorithm for the fusion tactic.

The predecessor of Wadler's deforestation is the listless transformer [Wad84, Wad85]. The initial listless transformer [Wad84] is a semi-decision procedure which could convert recursive programs with *bounded evaluation*[4] property to equivalent listless machines (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 [Wad85], 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

[4] needs bounded internal storage to perform computation

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 generalised deforestation algorithm presented in this paper is also a decision procedure. It is able to eliminate data structures apart from lists (e.g. trees) and selectively apply generalisation to avoid subterms which are unsafe to fuse. 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 in deforestation. This is not necessary a bad thing if one considers the advantages of modularisation for program transformation. In [Chi90], 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. However, further work is still needed to develop a more general framework for combining tactics.

Another work closely related to the listless transformer is Turchin's *supercompiler* [Tur86]. 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 can then be 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 data structure similar to the s-expression of Lisp. While we 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 generalisation.

Recently, Richard Waters proposed a new transformation technique [Wat91] 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 orginating 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 *tupled* function. The pre-order restriction is more limiting that the safe-unsafe criteria of our generalised deforestation, but the on-line cycle restriction is something new. In our case, this can only be achieved by combining fusion with the tupling tactic.

Another related area is partial evaluation [Con90, BEJ88, JSS89]. The primary mechanism used in partial evaluation is the specialisation of function calls which have some or all of its arguments *known*[5] (or *partially known*). 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* [Jon88] has been used to analyse (recursive) functions to find out which of the arguments are known or unknown (also known as *static* vs *dynamic*). However, this analysis cannot be used to guarantee the termination of the partial evaluation process itself. Lately, Holst has proposed an additional analysis, called *finiteness analysis* [Hol91], 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 applicable to strict, first-order functional languages. Whilst partial evaluation specialises known or partially known arguments, our deforestation technique appears to be more general as it also specialises *symbolic* arguments which are unknown. However, partial evaluation also employs reduction and simplification techniques which are currently ignored by deforestation.

[5] grounded term without free variables

9 Conclusion

The basic idea of annotating sub-terms, to distinguish between those 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 together with the discovery that it is possible to characterise, using syntactic properties, functions and their parameters into either safe or unsafe producers and consumers. This discovery led us to the use of a double annotation scheme to identify safe sub-terms which could be fused. It allowed us to extend 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 linearing certain pattern-matching parameters and handling certain unsafe producers with the help of laws.

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

10 Acknowlegement

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 paper have inspired much of the work done here. Thanks also to the National University of Singapore for financially supporting my PhD through their Graduate Overseas Scholarship. Lastly, thanks to the reviewers for their useful comments.

References

[Aug85] Leonard Augustsson. Compiling pattern-matching. In *Conference on Functional Programming and Computer Architecture (LNCS 201, ed Jouannaud)*, pages 368–381, Nancy, France, 1985.

[BD77] RM Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of Association for Computing Machinery*, 24(1):44–67, January 1977.

[BEJ88] D. Bjorner, AP. Ershov, and ND Jones. *Workshop on Partial Evaluation and Mixed Computations*. Gl Avarnes, Denmark, North-Holland, 1988.

[CD92] Wei-Ngan Chin and John Darlington. Higher-order removal transformation technique for functional programs. In *15th Australian Computer Science Conference, Australian CS Comm Vol 14, No 1*, pages 181–194, Hobart, Tasmania, January 1992.

[Chi90] Wei-Ngan Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, March 1990.

[Chi91] Wei-Ngan Chin. Generalising deforestation for all first-order functional programs. In *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages, BIGRE 74*, pages 173–181, Bordeaux,France, October 1991.

[Con90] Charles Consel. Binding time analysis for higher-order untyped functional languages. *6th ACM Conference on Lisp and Functional Programming*, pages 264–272, June 1990.

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM Transaction on Programming Languages and Systems*, 4(1):1–20, January 1982.

[FW88] AB Ferguson and Phil Wadler. When will deforestation stop? In *Proc of 1988 Glasgow Workshop on Functional Programming (as Research Report 89/R4 of Glasgow University)*, pages 39–56, Rothesay, Isle of Bute, August 1988.

[Hol91] Carsten Kehler Holst. Finiteness analysis. In *5th ACM Conference on Functional programming Languages and Computer Architecture*, pages 473–495, Cambridge, Massachusetts, August 1991.

[Jon88] Neil J. Jones. Automatic program specialisation: A re-examination from basic principles. In *Workshop on Partial Evaluation and Mixed Computations*, pages 225–282, Gl Avarnes, Denmark, North-Holland, 1988.

[JSS89] ND Jones, P Sestoft, and H Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. *Journal of LISP and Symbolic Computation*, 2(1):9–50, 1989.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transaction on Programming Languages and Systems*, 8(3):90–121, July 1986.

[Wad84] Phil Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *ACM Symposium on Lisp and Functional Programming*, pages 45–52, Austin, Texas, August 1984.

[Wad85] Phil Wadler. Listlessness is better than laziness II: Composing listless functions. In *Workshop on Programs as Data Objects*, pages 282–305, Springer-Verlag, New York, 1985.

[Wad87] Phil Wadler. Efficient compilation of pattern-matching. In *The Implementation of Functional Programming Languages (by Simon Peyton-Jones)*, chapter 5. Prentice-Hall International, 1987.

[Wad88] Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358, Nancy, France, March 1988.

[Wat91] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transaction on Programming Languages and Systems*, 13(1):52–98, January 1991.