Array Form Transformations: Proofs of Correctness[1]

Stephen Fitzpatrick, M. Clint, P. Kilpatrick
{S.Fitzpatrick, M.Clint, P.Kilpatrick}@cs.qub.ac.uk
Department of Computer Science, The Queen's University of Belfast
Belfast BT7 1NN, Northern Ireland, UK

May 31, 1995
Technical report 1995/May-SF.MC.PLK
<URL:http://www.cs.qub.ac.uk/pub/TechReports/1995/May-SF.MC.PLK/>
<URL:ftp://ftp.cs.qub.ac.uk/pub/TechReports/1995/May-SF.MC.PLK.ps.gz>

Abstract

A number of program transformations are proved to preserve the meaning of programs. The transformations convert array operations expressed using a small number of general-purpose functions into applications of a large number of functions suited to efficient implementation on an array processor.

Keywords: proof of correctness, program transformation, functional programming, array processor

1 Introduction

In [FSCB95], a set of program transformations for converting a simple, functional form into a whole-array form suitable for execution on an array processor is discussed (some components of the whole-array form are biased towards the AMT DAP array processor [PL90], which has a 2-dimensional array of processors). The initial form operated on by the transformations uses a small number of powerful functions to express array operations; the final form, called the Array Form, uses a large number of simpler functions, each of which can be efficiently executed by an array processor. Each program transformation partially rewrites an expression given in the initial form into an equivalent expression in the Array Form — complete conversion is achieved by the exhaustive application of the transformations (performed automatically using the TAMPR transformation system [Boy70]).

An important property of program transformations is their correctness: it is vital that a transformation be *meaning-preserving* — that is, a program has the same meaning both before and after transformation. Some of the transformations used in converting to Array Form are simple enough that their correctness might, in an informal context, be taken for granted; some of the transformations are more complex and it may not be easy (or safe) to conclude by informal means that they are correct. Such a mix of simple and complex transformations is typical of many practical applications of program transformation.

In this paper, proofs of correctness of all of the more complex transformations and of a sample of the simpler transformations are presented. Combined with proofs of termination and completeness (given certain restriction on the initial form) provided in [FSCB95], the correctness proofs establish the soundness of the conversion to Array Form. It turns out that the proofs for the simple Array Form transformations are trivial, indicating that it is probably not worthwhile carrying out proofs of those simple transformations not verified in this paper (that is, the informal assessment of these simple transformations as being 'obviously correct' is probably good enough). In addition, the proofs of the more complex transformations are straightforward to carry out and require only well known proof techniques (mainly induction over sets), supporting the oft made claim that transformations, being formal entities, are amenable to formal analysis.

The structure of the paper is as follows: the context of the transformation to Array Form is discussed; the basic notation used is explained; the functions used in the initial form and in the Array Form are defined; useful lemmas relating to the functions are established; the transformations are defined; and their correctness is proved. An example application of the transformations is given in the appendix.

2 Background

The transformations discussed in this paper constitute one of several stages in a *derivation process* which converts functional specifications of numerical mathematical algorithms into efficient implementations

[1]The work reported in this paper was supported by SERC grant GR/G 57970.

tailored to the AMT DAP array processor.

$$\begin{array}{l} \text{SML} \quad \rightarrow \lambda\text{-calculus} \rightarrow \text{Unfolded} \rightarrow \text{Simplified} \\ \qquad\qquad \rightarrow \text{Array Form} \rightarrow \text{Common Sub-expressions Eliminated} \\ \qquad\qquad\quad \rightarrow \text{Fortran Plus Enhanced (DAP)} \end{array}$$

Each stage of the derivation is concerned with one aspect of the conversion from abstract specification to efficient, DAP-specific implementation; a complex task is thus divided into a sequence of simpler tasks, each of which can be tackled separately.

- A functional specification is an abstract definition of an algorithm written in a functional programming language such as SML [Wil87] or Lisp; the definition is abstract since it defines *what* is to be computed rather than *how* it is to be computed, with no concern for execution efficiency and no bias towards any particular computational architecture or computer system.
- The transformations that constitute the initial stage of the derivation are applied to convert a specification written in a specific language such as SML into a simple, generic 'functional language': the λ-calculus. Conversion into the λ-calculus simplifies later stages of the derivation by removing 'syntactic sugar' and makes most of the derivation independent of the specification language, facilitating re-use with other languages.
- The well-known technique of function unfolding [BD77] (second derivation stage) is then combined with algebraic simplification (third stage) to remove many of the inefficiencies that are inherent in an abstract style of specification.
- The Array Form stage then converts the simple, functional form into a functional form that is suited to execution on an Array Processor.
- The Array Form so produced is then optimized by combining repeated computations of complex expressions.
- Finally, the functional form is converted into an imperative form that can be compiled and executed on the AMT DAP.

The Array Form stage is the pivotal stage in the creation of a DAP implementation from a specification. If the Array Form stage is omitted, then the derivation produces a sequential Fortran77 implementation. Indeed, several of the derivation stages can be omitted, and other stages can be included — by constructing an appropriate sequence of derivation stages, a programmer can obtain implementations for a variety of computational architectures (such as sequential, vector and multi-processor architectures) and can tailor implementations in particular ways (such as optimizing implementations for sparse matrices) [CFH$^+$94, CFH$^+$93].

Each derivation stage consists of one or more *transformation sequences*; the TAMPR transformation system applies each sequence once, in turn. For each sequence, the constituent transformations are applied exhaustively to a program. The transformations considered in this paper form one such transformation sequence; this sequence is the core sequence for the Array Form derivation stage (the other transformation sequences in that stage perform minor tasks such as standardizing expressions and manipulating type information). The remainder of this paper discusses only the simple, functional form (that is the input to the Array Form stage), the Array Form and the Array Form derivation stage.

3 Basic Notation

3.1 The λ-calculus

Both the initial form and the Array Form are pure, functional forms: operations are denoted by (side-effect-free) expressions. The underlying notation used is that of the λ-calculus [Chu51, Bar84], enriched with (optional) type information and named functions. The principal construct in the λ-calculus is the *λ-abstraction*, which denotes a function expression; for example

$$\lambda\text{x:int, y:int}\cdot\text{x-y}$$

denotes a function having formal arguments x and y (both integers) and which subtracts the second argument from the first. The identifiers used for formal arguments are arbitrary and can be systematically changed — a process called *α-conversion*; for example

$$\lambda\text{x, y}\cdot\text{x-y} \quad \text{and} \quad \lambda\text{a, b}\cdot\text{a-b}$$

denote the same function.

A *λ-binding* is a λ-abstraction applied to arguments; for example

$$\lambda\text{x:int, y:int}\cdot\text{x-y (1, 2)}$$

denotes a function applied to arguments 1 and 2; argument x is bound to the value 1, argument y to 2. Evaluation of an λ-binding is performed by substituting each occurrence of each bound identifier with the value to which that identifier is bound; this substitution is called *β-reduction* and is denoted here as [*identifier* $\rightarrow$ *value*]. For example,

$$\lambda\text{x, y}\cdot\text{x-y (1, 2)} \equiv \text{(x-y)[x} \rightarrow \text{1, y} \rightarrow \text{2]} \equiv \text{1-2}$$

A λ-binding may contain another λ-binding, and there is the possibility that the two bindings use the same identifier; such a situation is called a *name clash*. For example, let E be the expression

$$\lambda\text{x}\cdot\text{1+x*(}\lambda\text{x}\cdot\text{2+x (3))}$$

The occurrence of x in the sub-expression 1+x*($\lambda\ldots$) refers to the outer (left-most) binding, while the occurrence of x in the sub-expression 2+x refers to the inner (right-most) binding; the former occurrence is said to be a *free occurrence* of x in E. Only *free* occurrences of identifiers are affected by α-conversion and β-reduction; for example, the outer abstraction in the preceding example can be α-converted as follows:

$$\lambda\text{x}\cdot\text{1+x*(}\lambda\text{x}\cdot\text{2+x (3))} \quad \rightarrow \quad \lambda\text{y}\cdot\text{1+y*(}\lambda\text{x}\cdot\text{2+x (3))}$$

Note that, in an abstraction such as λi$\cdot$e, e may contain free occurrences of i but the abstraction as a whole cannot (since i is bound by the abstraction).

It is possible to 'reverse' the process of β-reduction, to λ-abstract occurrences of a sub-expression from an expression:

$$\text{B} \equiv \lambda\text{x}\cdot\text{(B[e} \rightarrow \text{x]) (e)}$$

That is, if x is bound to e, then e can be replaced with x. λ-abstraction is valid provided account is taken of name clashes and provided the abstracted expression e is not removed from any binding whose identifiers it contains. For example, the abstraction

$$\lambda\text{a}\cdot\text{g(f(a), f(a)) (b)} \rightarrow \lambda\text{x}\cdot\text{(}\lambda\text{a}\cdot\text{g(x, x) (b)) (f(a))}$$

in which f(a) is abstracted as x, is *incorrect* since, in the left expression, the a in f(a) is bound to b whilst, in the right expression, the a in f(a) is unbound.

An expression e is said to be *dependent on* an identifier i if e contains *free* occurrences of i; for example, i+1 is dependent on i. The terms 'is a function of' and 'contains' are synonymous with 'is dependent on'; thus, it may be said, perhaps somewhat confusingly, that the expression (x+λy$\cdot$y*y (f(x))) does *not* contain y since y is not free in the expression — the use of y as the bound identifier is entirely arbitrary.

A λ-binding that binds more than one identifier can be converted into a nest of λ-bindings, each of which binds one identifier; for example,

$$\lambda\text{i, j}\cdot\text{B (a, b)} \equiv \lambda\text{i}\cdot\text{(}\lambda\text{j}\cdot\text{B (b)) (a)} \equiv \lambda\text{j}\cdot\text{(}\lambda\text{i}\cdot\text{B (a)) (b)}$$

where a and b are such that their nesting does not introduce name clashes. This process of nesting is called Currying.

3.2 Indices and Sets

An array is considered to be a mapping from a (finite) set of indices onto some range of values of type α. For brevity, a 1-dimensional array may be referred to as a *vector* and a 2-dimensional array as a *matrix*.

An index identifies a position in an array. An index may be considered here to be a list of integers; for example, [1, 3, 5] is a 3-dimensional index, specifying position 1 in the first dimension, position 3 in the second, and position 5 in the third. The juxtaposition ij of indices i and j denotes their concatenation; for example, [1, 2, 3][4, 5] ≡ [1, 2, 3, 4, 5].

A characteristic of arrays is that their index sets are *regular* and, in particular, *rectangular*: that is, a set of multi-dimensional indices can be specified as the cartesian product of sets of 1-dimensional indices, and each set of 1-dimensional indices can be specified using a small (fixed) number of parameters. For example, the expression Shape([m, n]) denotes a set of $m \times n$ 2-dimensional indices: $\{1 \leq i \leq m, 1 \leq j \leq n : [i, j]\}$. The function Shape is a *constructor* that distinguishes lists which denote index sets from other lists, such as those denoting indices; when this distinction is implied by context, the constructor may be dropped.

Non-rectangular sets are often convenient. Such sets are constructed using standard set operations. In particular:

- S+i ≡ S ∪ {i} denotes the insertion of element i into set S (which is assumed not to contain i).
- The cartesian product of two arbitrary sets is denoted by S × T; for example [m] × [n] ≡ [m, n]. Cartesian product is not formally defined here, but the standard identities

$$\begin{aligned}
&S \times \emptyset \equiv \emptyset \times S \equiv \emptyset \\
&\{i\} \times \{j\} \equiv \{ij\} \\
&(R \cup S) \times T \equiv (R \times T) \cup (S \times T) \\
&R \times (S \cup T) \equiv (R \times S) \cup (R \times T)
\end{aligned}$$

  are assumed (the second states that the cartesian product of two singleton sets is itself a singleton set).

If a function takes as argument a list which has a known number of elements, then the argument may be written as a manifest list; for example, in the expression λ[i, j]·e the identifier i is bound to the first component of the argument, and j to the second component.

4 Array Functions

In the following sections, four *primitive* array functions and the Array Form functions are defined. The primitive functions are taken to define the semantics of arrays. Most array operations commonly encountered in numerical mathematics can be compactly expressed using these primitives. However, it is not intended that a programmer be restricted to using only these primitives: other, perhaps more convenient, functions can be defined in terms of these primitive functions. Transformations can be employed to eliminate such *derived* functions, using techniques such as unfolding and algebraic simplification.

The Array Form functions are much more restricted than the primitive functions, but are also much simpler to implement efficiently on an array processor such as the AMT DAP. In effect, the Array Form may be considered as a functional abstraction of an array processor.

4.1 Primitive Array Functions

The four primitive array functions are: shape, element, generate and reduce.

shape

Given an array, the function shape can be used to obtain its index set:

$$\text{shape}(A{:}\alpha \text{ array}) \rightarrow \text{Shape}$$

The extent of an array in a particular dimension can be obtained by specifying the dimension:

$$\text{shape(A:}\alpha\text{ array, n:int)} \rightarrow \text{int}$$

element

The element function

$$\text{element(A:}\alpha\text{ array, i:index)} \rightarrow \alpha$$

returns the value of the element of A at position i. For convenience, the infix operator @ is defined to be equivalent to element:

$$\text{A@i} \equiv \text{element(A, i)}$$

This paper tacitly switches between infix operator and prefix function according to which is the more convenient.

generate

The basic function for constructing arrays is generate:

$$\text{generate(S:Shape, g:index} \rightarrow \alpha) \rightarrow \alpha\text{ array}$$

The first argument, S, specifies the index set of the constructed array. The second argument, g, is a function, called the *generating function*, which determines the values of the elements: the value of element i is g(i).

The following are some examples of arrays constructed using generate:

- the elementwise addition of two arrays, of arbitrary dimensionality, having the same shape:

  $$\text{generate(shape(A), } \lambda\text{i}\cdot\text{A@i+B@i)}$$

- the transpose of a 2-dimensional array A of shape [m, n]:

  $$\text{generate([n, m], } \lambda\text{[i, j]}\cdot\text{A@[j, i])}$$

An argument, such as i, of a generating function is called a *generating index*. For multi-dimensional arrays, the term may also be used of the *components* of an index argument; for example, in λ[i, j]$\cdot$e, the generating indices are i and j. It should be clear from context whether a whole index or a component is being considered.

reduce

Many array operations require the elements of an array to be accumulated — or *reduced* — into a single value by the repeated application of a binary *reducing function*. For example, the sum of the elements of a numeric array is a reduction using the addition function. Reductions are denoted using the reduce function:

$$\text{reduce(r:}\alpha \times \alpha \rightarrow \alpha\text{, r0:}\alpha\text{, S:Shape, g:index} \rightarrow \alpha) \rightarrow \alpha$$

The argument r is the reducing function. The argument r0 is the *initial value* which is used to instantiate the accumulation (it is usually an identity of the reducing function and so does not alter the value of the reduction; its inclusion helps simplify the semantics of reductions by ensuring that a reduction is well defined even if an array contains only one element, or even no elements).

The arguments S and g (g is a generating function) can be used to specify the elements of the array which are to be reduced. For example,

$$\text{reduce(+, 0, shape(A), } \lambda\text{i}\cdot\text{A@i)}$$

produces the sum of the elements of array A. However, the generating function need not *necessarily* be an application of element: a reduction can involve any set of values which can be specified by applying a generating function over an index set. For example, the inner-product of two vectors U and V, of shape [n], is given by:

$$\mathrm{reduce}(+, 0, [n], \lambda i \cdot U@i*V@i)$$

The four functions shape, element, generate and reduce are the basic array functions; most common vector and matrix operations can be readily expressed using them. Some further examples are given below:

- Row i of a matrix A: generate(shape(A, 2), λ[j]·A@[i, j])
- Product of two matrices A and B (which are assumed to be conformant; i.e. the number of columns of A equals the number of rows of B):

$$\begin{array}{l}\mathrm{generate}([\mathrm{shape}(A, 1), \mathrm{shape}(B, 2)], \\ \quad \lambda[i, j]\cdot \mathrm{reduce}(+, 0, \mathrm{shape}(A, 2), \lambda[k]\cdot A@[i, k]*B@[k, j]))\end{array}$$

- Logical matrix having leading diagonal elements true, and all other elements false:

$$\mathrm{generate}([n, n], \lambda[i, j]\cdot i{=}j)$$

4.1.1 Formal Definition of generate and reduce

Definition 1: generate
The generate function is defined by the two identities

$$\begin{array}{rcll} \mathrm{shape}(\mathrm{generate}(S, \lambda i\cdot g)) & \equiv & S & \text{(G1)} \\ \forall\, i' \in S:\ \mathrm{element}(\mathrm{generate}(S, \lambda i\cdot g), i') & \equiv & \lambda i\cdot g\ (i') & \text{(G2)} \end{array}$$

□

That is, the shape of an array constructed by an application of generate is the shape specified by the shape argument; and the value of an element of the constructed array is found by applying the generating function to the element's index.

When the shape of a generation is a manifest list, axiom G1 may be used in the form

$$\mathrm{shape}(\mathrm{generate}([m, n], \lambda i\cdot g), 1) \equiv m$$

In the proofs presented in this paper, the condition $i' \in S$ in axiom G2 — that an index be a member of an array's shape — is usually ignored. It could be verified separately from the main proofs, or it could be included in the form

$$\mathrm{element}(\mathrm{generate}(S, \lambda i\cdot g), i') \equiv \text{if } (i' \in S) \text{ then } \lambda i\cdot g\ (i') \text{ else } \perp \qquad \text{(G2}'\text{)}$$

where $\perp$ is the undefined value, bottom. Context could then be used to establish that the condition $i' \in S$ is true, and so the conditional expression can be reduced into just the true limb. This technique is illustrated in one proof (of lemma 6).

Definition 2: reduce
The reduce function is defined recursively on the index set over which the reduction is to be performed:

$$\begin{array}{rcll} \mathrm{reduce}(r, r0, \emptyset, \lambda i\cdot g) & \equiv & r0 & \text{(R1)} \\ \mathrm{reduce}(r, r0, S{+}i', \lambda i\cdot g) & \equiv & r(\lambda i\cdot g\ (i'), \mathrm{reduce}(r, r0, S, \lambda i\cdot g)) & \text{(R2)} \end{array}$$

where $\emptyset$ denotes the empty set (of indices).
□

Note that no order is defined for performing reductions, so reducing functions should be associative and commutative.

4.2 Array Form Functions

The main Array Form functions are now defined. These functions are intended to capture the sorts of operations that any array processor could be expected to implement efficiently (for example, simultaneously adding all elements of two arrays). Some of the functions, and some of the transformations discussed later, are perhaps peculiar to array processors whose processors are arranged in a *two-dimensional* array (the AMT DAP is one such processor). It should be emphasized that what is under consideration in this paper is the correctness of the transformations that create the Array Form, rather than how well suited the Array Form is to a particular processor.

4.2.1 Main Array Form Functions

Probably the most important functions of the Array Form are the *mapping* functions, which apply a scalar function to each element of an array, or to corresponding elements of a pair of arrays. Mappings are supported for only certain scalar functions — for example, the basic arithmetic and logical functions.

Definition 3: map

$$\begin{aligned} &\textsf{map(A:}\alpha\textsf{ array, f:}\alpha \to \beta\textsf{)} \to \beta\textsf{ array}\\ \overset{def}{=}\ &\textsf{generate(shape(A), }\lambda\textsf{i}\cdot\textsf{f(A@i))}\end{aligned}$$

$$\begin{aligned} &\textsf{map(A:}\alpha\textsf{ array, B:}\beta\textsf{ array, f:}\alpha \times \beta \to \gamma\textsf{)} \to \gamma\textsf{ array}\\ \overset{def}{=}\ &\textsf{generate(shape(A), }\lambda\textsf{i}\cdot\textsf{f(A@i, B@i))}\end{aligned}$$

where A *and* B *have the same shape*

□

The fold function performs restricted forms of reductions, in which the values reduced are elements of an array and where the reducing function is one of a limited set (and typically evaluating sum, product, logical and, logical or, minimum or maximum). A variant of fold is also defined which reduces a matrix along its rows, thereby forming a vector of (partial) cumulative values.

Definition 4: fold

$$\begin{aligned} &\textsf{fold(r:}\alpha \times \alpha \to \alpha\textsf{, r0:}\alpha\textsf{, A:}\alpha\textsf{ array)} \to \alpha\\ \overset{def}{=}\ &\textsf{reduce(r, r0, shape(A), }\lambda\textsf{i}\cdot\textsf{A@i)}\end{aligned}$$

□

Definition 5: fold.rows

$$\begin{aligned} &\textsf{fold.rows(r:}\alpha \times \alpha \to \alpha\textsf{, R0:}\alpha\textsf{ vector, A:}\alpha\textsf{ matrix)} \to \alpha\textsf{ vector}\\ \overset{def}{=}\ &\textsf{generate([shape(A, 1)], }\lambda\textsf{[i]}\cdot\textsf{reduce(r, R0@[i], [shape(A, 2)], }\lambda\textsf{[j]}\cdot\textsf{A@[i, j]))}\end{aligned}$$

□

The join function is a data-parallel conditional. The result of applying join is an array with elements merged from two arrays according to whether the corresponding element of a *mask* array is true or false.

Definition 6: join

$$\begin{aligned} &\textsf{join(M:boolean array, T:}\alpha\textsf{ array, F:}\alpha\textsf{ array)} \to \alpha\textsf{ array}\\ \overset{def}{=}\ &\textsf{generate(shape(M), }\lambda\textsf{i}\cdot\textsf{if M@i then T@i else F@i)}\end{aligned}$$

where M, T *and* F *have the same Shape*

□

4.2.2 Miscellaneous Functions

The Array Form defines many functions for performing miscellaneous array operations such as transposing a matrix, 'shifting' the elements of a matrix in a specified direction and constructing logical matrices having true values in certain patterns (such as along their main diagonals or in their upper triangles). Here, only a few examples of such functions are considered, since the correctness of transformations involving such functions is usually trivial to establish from the function definitions.

Definition 7: matrix.transpose

$$\begin{aligned} & \text{matrix.transpose(A:}\alpha\text{ array)} \rightarrow \alpha\text{ array} \\ \overset{def}{=} & \text{ generate([shape(A, 2), shape(A, 1)], }\lambda\text{[i, j]·A@[j, i])} \end{aligned}$$

□

Definition 8: row

$$\begin{aligned} & \text{row(A:}\alpha\text{ array, i:int)} \rightarrow \alpha\text{ array} \\ \overset{def}{=} & \text{ generate(shape(A, 2), }\lambda\text{[j]·A@[i, j])} \end{aligned}$$

□

5 Preliminary Results

Some basic properties of generate and reduce are established below; in addition, some elementary identities of the λ-calculus are discussed.

In the following, if an identifier is introduced on the right of an identity, then it should be assumed that it is a 'new' identifier, i.e. one that does not occur free in the expression on the left. For example, in the identity

$$\text{B} \equiv \lambda\text{x·B[e} \rightarrow \text{x] (e)}$$

it is to be assumed that x does not occur free in B, so that no problem arises with name clashes.

Lemma 1: Identity λ-bindings
A λ-binding in which the bound identifier and the bound value are the same is redundant.

$$\lambda\text{x·B (x)} \equiv \text{B}$$

A proof of this lemma would require a consideration of the formal meaning of λ-bindings, a discussion of which is beyond the scope of this paper. However, it can be justified *informally* as follows: the right side of the identity can be β-reduced by substituting in B the bound value, x, for the bound identifier, also x; since the substitution of an identifier for itself is an identity operation, the reduced expression is equivalent to B; that is,

$$\lambda\text{x·B (x) = B[x} \rightarrow \text{x] = B}$$

□

Lemma 2: Propagation of λ-binding out of abstraction
A λ-binding can be moved out of an immediately enclosing λ-abstraction if the bound value does not depend on the identifier of abstraction.

$$\lambda\text{i·(}\lambda\text{x·B (e))} \equiv \lambda\text{x·(}\lambda\text{i·B) (e)}$$

where e *does not contain* i

(Note that x is still bound to e.)
Proof.

λi·(λx·B (e))
  = *since* e *does not contain* i, e *can be λ-abstracted using identifier* x
λx·(λi·(λx·B (x))) (e)
  = *remove identity λ-binding (lemma 1)*
λx·(λi·B) (e)

□

Lemma 3: Propagation of λ-binding through a function application
An applied λ-binding that is an argument in a function application can be moved outside the function application.

$$\mathsf{f}(\lambda \mathsf{x}\cdot \mathsf{B}\ (\mathsf{e})) \equiv \lambda \mathsf{x}\cdot \mathsf{f}(\mathsf{B})\ (\mathsf{e})$$
$$\textit{where } \mathsf{f} \textit{ does not contain } \mathsf{x}$$

Proof.

$\mathsf{f}(\lambda \mathsf{x}\cdot \mathsf{B}\ (\mathsf{e}))$
= *since* f *does not contain* x, e *can be* λ*-abstracted using identifier* x
$\lambda \mathsf{x}\cdot \mathsf{f}(\lambda \mathsf{x}\cdot \mathsf{B}\ (\mathsf{x}))\ (\mathsf{e})$
= *remove identity* λ*-binding (lemma 1)*
$\lambda \mathsf{x}\cdot \mathsf{f}(\mathsf{B})\ (\mathsf{e})$

□

This identity can be generalised for moving a λ-binding out of an argument in a function application which has more than one argument, provided all of the other arguments are free of the bound identifier. Note that this lemma can be applied in both directions, to move a λ-binding into, as well as out of, an argument position.

Lemma 4: Exchange of set union and insertion
Since set insertion is a form of set union, the commutative and associative laws that apply to union can be applied to an expression containing insertion as though it were union. In particular,

$$(\mathsf{S}+\mathsf{i}) \cup \mathsf{T} \equiv (\mathsf{S} \cup \mathsf{T})+\mathsf{i}$$

Proof.

$(\mathsf{S}+\mathsf{i}) \cup \mathsf{T}$
= *by definition of insertion*
$(\mathsf{S} \cup \{\mathsf{i}\}) \cup \mathsf{T}$
= *union is associative and commutative*
$(\mathsf{S} \cup \mathsf{T}) \cup \{\mathsf{i}\}$
= *by definition of insertion*
$(\mathsf{S} \cup \mathsf{T})+\mathsf{i}$

□

5.1 Properties of Elementwise Applications

The following lemmas pertain to elementwise applications of functions. Such applications can be denoted using the map function but, for convenience in later proofs, the operator ϵ is used to denote elementwise applications of binary functions:

Definition 9: Elementwise Operator, ϵ

$$\begin{aligned}&\epsilon(\mathsf{f}{:}\alpha \times \beta \to \gamma) \to (\alpha\ \mathsf{array} \times \beta\ \mathsf{array} \to \gamma\ \mathsf{array})\\ \overset{def}{=}\ &\lambda \mathsf{X}{:}\alpha\ \mathsf{array},\ \mathsf{Y}{:}\beta\ \mathsf{array}\cdot \mathsf{generate}(\mathsf{shape}(\mathsf{Y}),\ \lambda \mathsf{i}\cdot \mathsf{f}(\mathsf{X@i},\ \mathsf{Y@i}))\end{aligned}$$

□

For example, $\epsilon(+)$ is a function that performs elementwise addition of two arrays.

Lemma 5: Shape of an elementwise application
The shape of an application of an elementwise function to two arrays is the same as the shape of the second argument array (which is required to be of the same shape as the first argument array).

$$\mathsf{shape}(\epsilon(\mathsf{f})(\mathsf{A},\ \mathsf{B})) \equiv \mathsf{shape}(\mathsf{B})$$

Proof.

shape(ϵ(f)(A, B))
  = *definition of ϵ (definition 9)*
shape(λX, Y·generate(shape(Y), λi·f(X@i, Y@i)) (A, B))
  = *β-reduce*
shape(generate(shape(B), λi·f(A@i, B@i)))
  = *by G1*
shape(B)

□

Lemma 6: Element of an elementwise application
An element of an elementwise application of a function to two arrays is the value of that function applied to the corresponding elements of the arrays; that is, element propagates through ϵ:

$$\text{element}(\epsilon(f)(A, B), i) \equiv f(\text{element}(A, i), \text{element}(B, i)) \equiv f(A@i, B@i)$$
$$\textit{for } i \in \text{shape}(\epsilon(f)(A, B))$$

Proof.

element(ϵ(f)(A, B), i′)
  = *definition of ϵ (definition 9)*
element(λX, Y·generate(shape(Y), λi·f(X@i, Y@i)) (A, B), i′)
  = *β-reduce*
element(generate(shape(B), λi·f(A@i, B@i)), i′)
  = *by G2′*
if (i′ $\in$ shape(ϵ(f)(A, B))) then λi·f(A@i, B@i) (i′) else $\perp$
  = i′ $\in$ shape(ϵ(f)(A, B)) *is true by assumption*
λi·f(A@i, B@i) (i′)
  = *β-reduce*
f(A@i′, B@i′)

□

As discussed previously, axiom G2 rather than G2’ will be used in subsequent proofs and the requirement that the index be an element of the array’s shape will be ignored.

5.2 Properties of Reductions

If the reducing function of a reduction is an elementwise function, then the result of the reduction is an array, so it is valid to apply the shape and element functions to the result. Such a reduction is called an *ϵ-reduction*:

Definition 10: ϵ-reduction
A reduction of the form

$$\text{reduce}(\epsilon(r), R0, S, \lambda i \cdot g)$$

is called an ϵ-reduction.

□

The following two lemmas pertain to ϵ-reductions.

Lemma 7: Shape of an ϵ-reduction
The shape of the result of an ϵ-reduction is the same as the shape of the reduction’s initial value:

$$\text{shape}(\text{reduce}(\epsilon(r), R0, S, \lambda i \cdot g)) \equiv \text{shape}(R0)$$

Proof.

Proof is by induction on S.

Base Step: $\emptyset$

shape(reduce(ϵ(r), R0, $\emptyset$, λi·g))

= *by R1*

shape(R0)

Inductive Step: S+i'

Assume lemma holds for shape S. Now consider shape S+i$'$.

shape(reduce(ϵ(r), R0, S+i$'$, λi·g))

= *by R2*

shape(ϵ(r)(λi·g (i$'$), reduce(ϵ(r), R0, S, λi·g)))

= *by lemma 5*

shape(reduce(ϵ(r), R0, S, λi·g))

= *by induction hypothesis*

shape(R0)

Hence, by induction, the lemma holds for all shapes.

□

Lemma 8: Element of an ϵ-reduction

An element of a reduction using ϵ(r) is a reduction using r: that is, element can be propagated through an ϵ-reduction.

$$\begin{aligned}&\text{element(reduce(}\epsilon\text{(r), R0, S, }\lambda\text{i}\cdot\text{g), j)}\\ \equiv\ &\text{reduce(r, element(R0, j), S, }\lambda\text{i}\cdot\text{element(g, j))}\end{aligned}$$

Proof. Proof is by induction on the shape over which the reduction is performed.

Base Step: $\emptyset$

Left side:

element(reduce(ϵ(r), R0, $\emptyset$, λi·g), j)

= *by R1*

element(R0, j)

Right side:

reduce(r, element(R0, j), $\emptyset$, λi·element(g, j))

= *by R1*

element(R0, j)

Inductive Step: S+i'

Assume the lemma holds for shape S. Now consider shape S+i$'$.

Left side:

element(reduce(ϵ(r), R0, S+i$'$, λi·g), j)

= *by R2*

element(ϵ(r)(λi·g (i$'$), reduce(ϵ(r), R0, S, λi·g)), j)

= *by lemma 6*

r(element(λi·g (i$'$), j), element(reduce(ϵ(r), R0, S, λi·g), j))

= *by induction hypothesis*

r(element(λi·g (i$'$), j), reduce(r, element(R0, j), S, λi·element(g, j)))

Right side:

$\mathsf{reduce(r, element(R0, j), S{+}i', \lambda i{\cdot}element(g, j))}$

= *by R2*

$\mathsf{r(\lambda i{\cdot}element(g, j)\ (i'), reduce(r, element(R0, j), S, \lambda i{\cdot}element(g, j)))}$

= *since* j *does not contain* i, *propagate λ-binding into element (lemma 3)*

$\mathsf{r(element(\lambda i{\cdot}g\ (i'), j), reduce(r, element(R0, j), S, \lambda i{\cdot}element(g, j)))}$

Hence, by induction, the lemma holds for all shapes.

□

Lemma 9: Reduction over a union
Since no order is specified for performing reductions (and since reducing functions are required to be associative and commutative), a reduction over an index set that is the union of two sets can be split into a pair of reductions.

$$\mathsf{reduce(r, r0, S \cup T, \lambda i{\cdot}e) \equiv r(reduce(r, r0, S, \lambda i{\cdot}e), reduce(r, r0, T, \lambda i{\cdot}e))}$$

where r0 *is an identity element of* r

Proof. Proof is by induction over S.

Base Step: ∅

Left side:

$\mathsf{reduce(r, r0, \emptyset \cup T, \lambda i{\cdot}e)}$

=

$\mathsf{reduce(r, r0, T, \lambda i{\cdot}e)}$

Right side:

$\mathsf{r(reduce(r, r0, \emptyset, \lambda i{\cdot}e), reduce(r, r0, T, \lambda i{\cdot}e))}$

= *by R1*

$\mathsf{r(r0, reduce(r, r0, T, \lambda i{\cdot}e))}$

= *since* r0 *is an identity of* r

$\mathsf{reduce(r, r0, T, \lambda i{\cdot}e)}$

Inductive Step: S+i'

Assume lemma holds for shape S. Now consider shape S+i′.

Left side:

$\mathsf{reduce(r, r0, (S{+}i') \cup T, \lambda i{\cdot}e)}$

= *interchanging set insertion and union (lemma 4)*

$\mathsf{reduce(r, r0, (S \cup T){+}i', \lambda i{\cdot}e)}$

= *by R2*

$\mathsf{r(\lambda i{\cdot}e\ (i'), reduce(r, r0, S \cup T, \lambda i{\cdot}e))}$

= *by induction hypothesis*

$\mathsf{r(\lambda i{\cdot}e\ (i'), r(reduce(r, r0, S, \lambda i{\cdot}e), reduce(r, r0, T, \lambda i{\cdot}e)))}$

Right side:

$\mathsf{r(reduce(r, r0, S{+}i', \lambda i{\cdot}e), reduce(r, r0, T, \lambda i{\cdot}e))}$

= *by R2*

$\mathsf{r(r(\lambda i{\cdot}e\ (i'), reduce(r, r0, S, \lambda i{\cdot}e)), reduce(r, r0, T, \lambda i{\cdot}e))}$

= *since* r *is associative*

$\mathsf{r(\lambda i{\cdot}e\ (i'), r(reduce(r, r0, S, \lambda i{\cdot}e), reduce(r, r0, T, \lambda i{\cdot}e)))}$

Hence, by induction, the lemma holds for all shapes.

□

Lemma 10: Collapsing singleton dimensions
If one of the dimensions of a multi-dimensional reduction contains only a single member, that dimension can be collapsed — that is, removed from the reduction's index set. For this paper, collapsing is required for only leading dimensions (i is a leading dimension in $\{\mathsf{i}\} \times \mathsf{S}$).

$$\mathsf{reduce(r, r0, \{i'\} \times S, \lambda ij \cdot e) \equiv reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i')))}$$

where r0 *is an identity element of* r *and where* i′ *and* i *have the same dimensionality*

Proof. Proof is by induction over S.

Base Step: $\emptyset$

Left side:

$\mathsf{reduce(r, r0, \{i'\} \times \emptyset, \lambda ij \cdot e)}$
 =
$\mathsf{reduce(r, r0, \emptyset, \lambda ij \cdot e)}$
 = *by R1*
$\mathsf{r0}$

Right side:

$\mathsf{reduce(r, r0, \emptyset, \lambda j \cdot (\lambda i \cdot e\ (i')))}$
 = *by R1*
$\mathsf{r0}$

Inductive Step: S+j'

Assume lemma holds for shape S. Now consider shape S+j′.

Left side:

$\mathsf{reduce(r, r0, \{i'\} \times (S{+}j'), \lambda ij \cdot e)}$
 = *by definition of set insertion*
$\mathsf{reduce(r, r0, \{i'\} \times (S \cup \{j'\}), \lambda ij \cdot e)}$
 =
$\mathsf{reduce(r, r0, (\{i'\} \times S) \cup (\{i'\} \times \{j'\}), \lambda ij \cdot e)}$
 = *split index set (lemma 9)*
$\mathsf{r(reduce(r, r0, \{i'\} \times S, \lambda ij \cdot e), reduce(r, r0, \{i'\} \times \{j'\}, \lambda ij \cdot e))}$
 = *induction hypothesis; cartesian product of singleton sets is a singleton set*
$\mathsf{r(reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i'))), reduce(r, r0, \{i'j'\}, \lambda ij \cdot e))}$
 = *by R2*
$\mathsf{r(reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i'))), r(\lambda ij \cdot e\ (i'j'), reduce(r, r0, \emptyset, \lambda ij \cdot e)))}$
 = *by R1 and by definition of index concatenation*
$\mathsf{r(reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i'))), r(\lambda i, j \cdot e\ (i', j'), r0))}$
 = r0 *is an identity of* r*; Currying*
$\mathsf{r(reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i'))), \lambda j \cdot (\lambda i \cdot e\ (i'))\ (j'))}$

Right side:

$\mathsf{reduce(r, r0, S{+}j', \lambda j \cdot (\lambda i \cdot e\ (i')))}$
 = *by R2*
$\mathsf{r(\lambda j \cdot (\lambda i \cdot e\ (i'))\ (j'), reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i'))))}$
 = *since* r *is commutative*
$\mathsf{r(reduce(r, r0, S, \lambda j \cdot (\lambda i \cdot e\ (i'))), \lambda j \cdot (\lambda i \cdot e\ (i'))\ (j'))}$

Hence, by induction, the lemma holds for all shapes.

□

6 The Transformations

In this section, the main transformations for converting from expressions that use the basic array functions (generate, reduce, etc.) into expressions that use Array Form functions are listed; their correctness is established in the following section. The basic strategy for converting into Array Form is to propagate applications of generate into generating functions; for example, a generation of a suitable scalar function becomes an application of map; and a generation of a conditional expression becomes an application of join — this strategy is discussed at length in [FSCB95]. In addition, several transformations optimize combinations of operations to make best use of the DAP hardware.

In general, a transformation is a unidirectional rewrite rule — it rewrites expressions matching some specified pattern into a form specified by a replacement. However, all of the transformations used in converting to Array Form are based upon algebraic identities: they could be applied in both the forward and reverse direction, though in practice they are only applied in the forward direction. That is, an identity $\mathsf{x} \equiv \mathsf{y}$ is used as a rewrite rule $\mathsf{x} \to \mathsf{y}$. To establish the correctness of the transformations, it is sufficient to show that the identities are valid; consequently, it is the identities that are considered here, though they are referred to as transformations.

6.1 General Transformations

Constructing an array by applying a scalar function to the elements of an array (or to corresponding elements of a pair of arrays) is equivalent to applying the elementwise version of the function (expressed using map).

Transformation 1: Propagation through scalar functions

$$\mathsf{generate(S, \lambda i \cdot f(a)) \equiv map(generate(S, \lambda i \cdot a), f)}$$
$$\mathsf{generate(S, \lambda i \cdot f(a, b)) \equiv map(generate(S, \lambda i \cdot a), generate(S, \lambda i \cdot b), f)}$$

where f *is a scalar function for which elementwise application is supported*

□

(The function f is necessarily independent of i.)

Constructing an array by evaluating a conditional expression for each element is equivalent to forming a data-parallel conditional, in which arrays are constructed from the true and false limbs of the conditional and are merged according to a mask generated from the predicate.

Transformation 2: Propagation through conditional

$$\begin{aligned} &\mathsf{generate(S, \lambda i \cdot if\ p\ then\ t\ else\ f)} \\ \equiv\ &\mathsf{join(generate(S, \lambda i \cdot p), generate(S, \lambda i \cdot t), generate(S, \lambda i \cdot f))} \end{aligned}$$

□

As mentioned previously, only a few examples of miscellaneous Array Form functions will be considered.

Transformation 3: matrix.transpose

$$\mathsf{generate([m, n], \lambda[i, j] \cdot A@[j, i]) \equiv matrix.transpose(A)}$$

where A *has shape* [n, m]

□

Transformation 4: row

$$\mathsf{generate([m], \lambda[j] \cdot A@[r, j]) \equiv row(A, r)}$$

where A *has shape* [n, m] *and* A *and* r *are independent of* j

□

6.2 Propagation through λ-expressions

Consider a generation in which the body of the generating function is a λ-binding:

$$\text{generate(S, } \lambda\text{i}\cdot(\lambda\text{x}\cdot\text{B (e)))}$$

The task of the Array Form transformations is to convert this generation into a form that can be efficiently implemented on an array processor: in the general case, all of the bindings are evaluated in parallel, then all of the body expressions are evaluated in parallel:

$$\forall\ \text{i} \in \text{S: evaluate e;}$$
$$\forall\ \text{i} \in \text{S: evaluate B}$$

If parallelism of unlimited dimensionality were permitted, it would be a simple matter to create this parallel form. However, because the DAP is limited to 2-dimensional parallelism, it is incapable of efficiently implementing the general case in the above manner. For example, if S were 2-dimensional and e 1-dimensional, then the above scheme would require the creation of a 2-dimensional array of 1-dimensional arrays, a structure which the DAP can manipulate, *but not in a completely parallel manner*.

Nevertheless, the above scheme can be used for certain cases where the *effective* or useful parallelism is at most 2-dimensional: for example, if S is 1- or 2-dimensional and e is a scalar value; or if S is 2-dimensional and e is a vector which is independent of one of the dimensions of S. The transformations below pertain to such cases; if none of these transformations apply to a given binding, then, as a last resort, the binding can be β-reduced in the hope that the resulting expression can be parallelised. In the following, it is assumed that all shapes in generations and reductions are at most 2-dimensional.

If the bound value is independent of the generating index, then the generation can be propagated into the binding.

Transformation 5: Invariant binding

$$\text{generate(S, } \lambda\text{i}\cdot\lambda\text{x}\cdot\text{B (e))} \equiv \lambda\text{x}\cdot\text{generate(S, } \lambda\text{i}\cdot\text{B) (e)}$$

where e *is independent of* i *and* S *is independent of* x

□

If the bound value is a scalar, then the generation can be propagated into the binding by creating an array of bound values.

Transformation 6: Scalar binding

$$\begin{aligned} &\text{generate(S, } \lambda\text{i}\cdot(\lambda\text{x}\cdot\text{B (e)))} \\ \equiv\ &\lambda\text{X}\cdot\text{generate(S, } \lambda\text{i}\cdot(\lambda\text{x}\cdot\text{B (X@[i]))) (generate(S, } \lambda\text{i}\cdot\text{e))} \end{aligned}$$

where e *is a scalar expression*

□

The binding for x is removed by β-reduction after application of this transformation.

Suppose that a 2-dimensional array of shape [l, m] is being generated over indices i and j, and that for each element a λ-binding is formed in which the bound value is a vector generation over index k. In the general case, $l \times m$ vectors must be created. However, if the bound vector is independent of j, then only l vectors need be created (one for each value of i). These l vectors can be created simultaneously as the rows of a matrix generation over indices i and k.

Transformation 7: Matrix generation, vector binding independent of one generating index

$$\begin{aligned} &\text{generate([l, m], } \lambda\text{[i, j]}\cdot(\lambda\text{x}\cdot\text{B (generate([n], } \lambda\text{[k]}\cdot\text{e))))} \\ \equiv\ &\lambda\text{X}\cdot\text{generate([l, m], } \lambda\text{[i, j]}\cdot(\lambda\text{x}\cdot\text{B (generate([n], } \lambda\text{[k]}\cdot\text{X@[i, k]))))} \\ &\quad\text{(generate([l, m], } \lambda\text{[i, k]}\cdot\text{e))} \end{aligned}$$

where e *is a function of* i *and* k *but not of* j*, and is not an application of* element

□

As is, the expression produced by this transformation does not appear to be an improvement over the initial expression, as the generation still contains a vector binding. However, it is usually the case that

B requires only individual elements of x, and not the vector as a whole.[2] Then the binding for x can be reduced and parallelisation of the resulting expression can proceed.

A similar transformation can be applied when e is independent of i rather than j. (It is assumed that the case of e independent of k is optimized by converting the vector binding into a scalar binding.)

If the array being constructed is a matrix and the bound value is a vector which is dependent on both matrix indices, and if the generating function of the vector is a function application, then the function application can be moved outside the vector. This transformation is expressed below for arrays of arbitrary dimensionality, but it is applied in practice only to a matrix generation/vector binding combination.

Transformation 8: Binding is a function application

$$\begin{aligned} & \mathsf{generate(S, \lambda i{\cdot}(\lambda x{\cdot}B\ (generate(T, \lambda j{\cdot}f(a)))))} \\ \equiv\ & \mathsf{generate(S, \lambda i{\cdot}(\lambda x'{\cdot}(\lambda x{\cdot}B\ (generate(T, \lambda j{\cdot}f(x'@j)))))\ \ (generate(T, \lambda j{\cdot}a)))} \end{aligned}$$

□

As discussed above, it may be possible to remove the binding for x if only individual elements of x are required, and not the entire array. The repeated application of this transformation may reduce the binding into a form that can be parallelised by one of the preceding transformations. A similar transformation can be used if f is a binary function.

6.3 Transformations for Reductions

Consider a generation having a generating function that is a reduction:

$$\mathsf{generate(S, \lambda i{\cdot}reduce(r, r0, T, \lambda j{\cdot}g))}$$

There are several ways that this expression could be converted into Array Form:

- Each of the reductions can be parallelised: that is, i is iterated over sequentially, and for each i, a parallel reduction is performed.
- The generation can be parallelised by exchanging the generate and the reduce; then j is iterated over sequentially, while for each j, the generation is evaluated in parallel.
- The generation and the reduction can be combined into a partial reduction (such as fold.rows), so that both are evaluated in parallel.

The third option, combination, is generally preferable when it is feasible, since it makes maximum use of parallelism. However, on a computer such as the DAP, which is limited to 2-dimensional parallelism, combination is possible only when both S and T are 1-dimensional.

Failing the third option, the second option is preferable, since generations generally make better use of parallelism than reductions. (For example, two arrays of arbitrary size can, theoretically, be added in a single step, whereas the reduction of an array of size [n] requires $\log_2(n)$ steps.) Exchanging a matrix generation and a vector reduction is equivalent to the well known optimization for matrix product, in which the 'ijk' order (k parallel) is converted into the 'kij' order (ij parallel).

The following two transformations enforce these preferences.

Transformation 9: generate-reduce combination

$$\begin{aligned} & \mathsf{generate([m], \lambda[i]{\cdot}reduce(r, r0, [n], \lambda[j]{\cdot}e))} \\ \equiv\ & \mathsf{fold.rows(r, generate([m], \lambda[i]{\cdot}r0), generate([m, n], \lambda[i, j]{\cdot}e))} \\ & \quad \textit{where } \mathsf{n} \textit{ and } \mathsf{r} \textit{ are independent of } \mathsf{i} \end{aligned}$$

□

[2]Function unfolding and algebraic simplification normally result in array expressions occurring solely as arguments of element. The exceptions are arrays which occur as arguments in or results of applications of recursive functions.

Transformation 10: generate-reduce swap

generate(S, λi·reduce(r, r0, T, λj·e))
$\equiv$ reduce(ϵ(r), generate(S, λi·r0), T, λj·generate(S, λi·e))
where r, r0 *and* T *are independent of* i *and* S *is independent of* j

□

This transformation can be applied for arrays of arbitrary dimensionality; for the DAP, however, it is used only for matrix generation and vector reduction.

If each component of a reduction is itself a reduction (which uses the same reducing function), then coalescing the reductions into a single reduction increases parallelism.

Transformation 11: reduce-reduce combination

reduce(r, r0, S, λi·reduce(r, r0, T, λj·e)) $\equiv$ reduce(r, r0, S $\times$ T, λij·e)
where r0 *is an identity element of* r *and* r *and* T *are independent of* i

□

7 Proofs of Correctness

The correctness of the transformations discussed in the preceding section is established below. The proofs are presented in the same order as the transformations.

Proof 1: Propagation through scalar functions
Consider the case of a binary function.

generate(S, λi·f(a, b)) $\equiv$ map(generate(S, λi·a), generate(S, λi·b), f)

Proof follows directly from the definition of map:

map(generate(S, λi·a), generate(S, λi·b), f)
= *definition 3*
generate(shape(generate(S, λi·a)),
λi·f(element(generate(S, λi·a), i), element(generate(S, λi·b), i)))
= *by G1 and G2*
generate(S, λi·f(λi·a (i), λi·b (i)))
= *remove identity bindings (lemma 1)*
generate(S, λi·f(a, b))

A similar proof applies for unary functions.

□

Note that the proof is simplified by choosing the appropriate generating index when the generation is introduced. Any other generating index, say j, could be used and would lead to an expression such as

generate(S, λj·f(λi·a (j), λi·b (j)))

which, since f is independent of j, is equivalent to

generate(S, λj·(λi·f(a, b) (j)))

which is one way of expressing the process of α-converting a λ-abstraction from using identifier i to using identifier j; that is, this expression is equivalent under α-conversion to generate(S, λi·f(a, b)).

Proof 2: Propagation through conditional

generate(S, λi·if p then t else f)
$\equiv$ join(generate(S, λi·p), generate(S, λi·t), generate(S, λi·f))

Proof follows directly from the definition of join:

join(generate(S, λi·p), generate(S, λi·t), generate(S, λi·f))
  = *definition 6*
generate(shape(generate(S, λi·p)),
  λi·if element(generate(S, λi·p), i)
  then element(generate(S, λi·t), i)
  else element(generate(S, λi·f), i))
  = *by G1 and G2*
generate(S, λi·if λi·p (i) then λi·t (i) else λi·f (i))
  = *remove identity bindings (lemma 1)*
generate(S, λi·if p then t else f)

□

Proof 3: matrix.transpose

generate([m, n], λ[i, j]·A@[j, i]) ≡ matrix.transpose(A)
where A *has shape* [n, m]

Proof follows directly from the definition of matrix.transpose:

matrix.transpose(A)
  = *definition 7*
generate([shape(A, 2), shape(A, 1)], λ[i, j]·A@[j, i])
  = *substituting for the shape of* A
generate([m, n], λ[i, j]·A@[j, i])

□

Proof 4: row

generate([m], λ[j]·A@[r, j]) ≡ row(A, r)
where A *and* r *are independent of* i *and* A *has shape* [n, m]

Proof follows directly from the definition of row:

row(A, r)
  = *definition 8*
generate(shape(A, 2), λ[j]·A@[r, j])
  = *substituting for the shape of* A
generate([m], λ[j]·A@[r, j])

□

7.1 Proofs of Transformations for λ-bindings

The following proofs pertain to transformations for generating functions whose bodies are λ-bindings.

Proof 5: Invariant binding

generate(S, λi·(λx·B (e))) ≡ λx·generate(S, λi·B) (e)
where e *is independent of* i *and* S *is independent of* x

Proof involves only elementary properties of the λ-calculus:

generate(S, λi·(λx·B (e)))
  = *move binding of* x *out of abstraction of* i *(lemma 2)*
generate(S, λx·(λi·B) (e))
  = *move binding out of application of* generate *(lemma 3)*
λx·generate(S, λi·B) (e)

□

Proof 6: Scalar binding

$$\begin{aligned}&\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (e)))\\ \equiv\ &\lambda X\cdot \mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (X@[i])))\ (\mathsf{generate}(S,\ \lambda i\cdot e))\end{aligned}$$

where e *is a scalar*

Proof follows by β-reducing X:

$\lambda X\cdot \mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (X@[i])))\ \ (\mathsf{generate}(S,\ \lambda i\cdot e))$
  = *β-reduce* X*; only occurrence of* X *is that shown*
$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (\mathsf{element}(\mathsf{generate}(S,\ \lambda i\cdot e),\ i))))$
  = *by G2*
$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (\lambda i\cdot e\ (i))))$
  = *remove identity binding (lemma 1)*
$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (e)))$

□

Proof 7: Matrix generation, vector binding independent of one generating index

$$\begin{aligned}&\mathsf{generate}([l,\ m],\ \lambda[i,\ j]\cdot(\lambda x\cdot B\ (\mathsf{generate}([n],\ \lambda[k]\cdot e))))\\ \equiv\ &\lambda X\cdot \mathsf{generate}([l,\ m],\ \lambda[i,\ j]\cdot(\lambda x\cdot B\ (\mathsf{generate}([n],\ \lambda[k]\cdot X@[i,\ k]))))\\ &\ (\mathsf{generate}([l,\ m],\ \lambda[i,\ k]\cdot e))\end{aligned}$$

where e *is a function of* i *and* k*, but not of* j

Proof follows by β-reducing X:

$\lambda X\cdot \mathsf{generate}([l,\ m],\ \lambda[i,\ j]\cdot(\lambda x\cdot B\ (\mathsf{generate}([n],\ \lambda[k]\cdot X@[i,\ k]))))$
 $(\mathsf{generate}([l,\ m],\ \lambda[i,\ k]\cdot e))$
  = *β-reduce* X*; only occurrence of* X *is that shown*
$\mathsf{generate}([l,\ m],$
 $\lambda[i,\ j]\cdot(\lambda x\cdot B\ (\mathsf{generate}([n],\ \lambda[k]\cdot \mathsf{element}(\mathsf{generate}([l,\ m],\ \lambda[i,\ k]\cdot e),\ [i,\ k])))))$
  = *by G2*
$\mathsf{generate}([l,\ m],\ \lambda[i,\ j]\cdot(\lambda x\cdot B\ (\mathsf{generate}([n],\ \lambda[k]\cdot(\lambda[i,\ k]\cdot e\ ([i,\ k]))))))$
  = *remove identity binding (lemma 1)*
$\mathsf{generate}([l,\ m],\ \lambda[i,\ j]\cdot(\lambda x\cdot B\ (\mathsf{generate}([n],\ \lambda[k]\cdot e))))$

□

Proof 8: Binding is a function application

$$\begin{aligned}&\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (\mathsf{generate}(T,\ \lambda j\cdot f(a)))))\\ \equiv\ &\mathsf{generate}(S,\ \lambda i\cdot(\lambda x'\cdot(\lambda x\cdot B\ (\mathsf{generate}(T,\ \lambda j\cdot f(x'@j)))))\ \ (\mathsf{generate}(T,\ \lambda j\cdot a)))\end{aligned}$$

Proof follows by β-reducing x':

$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x'\cdot(\lambda x\cdot B\ (\mathsf{generate}(T,\ \lambda j\cdot f(x'@j)))))\ \ (\mathsf{generate}(T,\ \lambda j\cdot a)))$
  = *β-reduce* x'*: only instance of* x' *is that shown*
$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (\mathsf{generate}(T,\ \lambda j\cdot f(\mathsf{element}(\mathsf{generate}(T,\ \lambda j\cdot a),\ j))))))$
  = *by G2*
$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (\mathsf{generate}(T,\ \lambda j\cdot f(\lambda j\cdot a\ (j))))))$
  = *remove identity binding (lemma 1)*
$\mathsf{generate}(S,\ \lambda i\cdot(\lambda x\cdot B\ (\mathsf{generate}(T,\ \lambda j\cdot f(a)))))$

□

7.2 Proofs of Transformations for Reductions

Proof 9: generate-reduce combination

$$\begin{aligned}&\text{generate}([m], \lambda[i]\cdot\text{reduce}(r, r0, [n], \lambda[j]\cdot e))\\ \equiv\ &\text{fold.rows}(r, \text{generate}([m], \lambda[i]\cdot r0), \text{generate}([m, n], \lambda[i, j]\cdot e))\\ &\quad \textit{where } n \textit{ and } r \textit{ are independent of } i\end{aligned}$$

Proof follows directly from the definition of fold.rows:

fold.rows(r, generate([m], λ[i]·r0), generate([m, n], λ[i, j]·e))
  = *by definition 5*
generate([shape(generate([m, n], λ[i, j]·e), 1)],
  λ[i]·reduce(r,
    element(generate([m], λ[i]·r0), [i]),
    [shape(generate([m, n], λ[i, j]·e), 2)],
    λ[j]·element(generate([m, n], λ[i, j]·e), [i, j])))
  = *by G1 and G2*
generate([m], λ[i]·reduce(r, λ[i]·r0 ([i]), [n], λ[j]·λ[i, j]·e ([i, j])))
  = *remove identity bindings (lemma 1)*
generate([m], λ[i]·reduce(r, r0, [n], λ[j]·e))

□

Proof 10: generate-reduce swap

$$\begin{aligned}&\text{generate}(S, \lambda i\cdot\text{reduce}(r, r0, T, \lambda j\cdot e))\\ \equiv\ &\text{reduce}(\epsilon(r), \text{generate}(S, \lambda i\cdot r0), T, \lambda j\cdot\text{generate}(S, \lambda i\cdot e))\\ &\quad \textit{where } r, r0 \textit{ and } T \textit{ are independent of } i \textit{ and } S \textit{ is independent of } j\end{aligned}$$

Since both sides of this identity evaluate to arrays, proof of this identity requires proof that the two arrays have the same shape and the same elements.

Same Shapes

Left side:

shape(generate(S, λi·reduce(r, r0, T, λj·e)))
  = *by G1*
S

Right side:

shape(reduce(ϵ(r), generate(S, λi·r0), T, λj·generate(S, λi·e)))
  = *by lemma 7*
shape(generate(S, λi·r0))
  = *by G1*
S

Same Elements

Consider an arbitrary element i'.

Left side:

element(generate(S, λi·reduce(r, r0, T, λj·e)), i′)

= *by G2*

λi·reduce(r, r0, T, λj·e) (i′)

= *since* r, r0 *and* T *are independent of* i, *move binding into reduction (lemma 3)*

reduce(r, r0, T, λi·(λj·e) (i′))

= *move binding of* i *into abstraction of* j *(lemma 2)*

reduce(r, r0, T, λj·(λi·e (i′)))

Right side:

element(reduce(ϵ(r), generate(S, λi·r0), T, λj·generate(S, λi·e)), i′)

= *move* element *into reduction (lemma 8)*

reduce(r, element(generate(S, λi·r0), i′), T,
  λj·element(generate(S, λi·e), i′))

= *by G2*

reduce(r, λi·r0 (i′), T, λj·(λi·e (i′)))

= *since* r0 *is independent of* i

reduce(r, r0, T, λj·(λi·e (i′)))

Hence, the left and right sides have the same shapes and the same elements; thus they are equivalent.
□

Proof 11: reduce-reduce combination

reduce(r, r0, S, λi·reduce(r, r0, T, λj·e)) $\equiv$ reduce(r, r0, S $\times$ T, λij·e)
  where r0 *is an identity element of* r *and* r, r0 *and* T *are independent of* i

Proof is by induction over S.

Base Step: $\emptyset$

Left side:

reduce(r, r0, $\emptyset$, λi·reduce(r, r0, T, λj·e))

= *by R1*

r0

Right side:

reduce(r, r0, $\emptyset$ $\times$ T, λij·e)

=

reduce(r, r0, $\emptyset$, λij·e)

= *by R1*

r0

Inductive Step: S+i’

Assume identity holds for shape S. Now consider shape S+i′.

Left side:

reduce(r, r0, S+i′, λi·reduce(r, r0, T, λj·e))
  = *by R2*
r(λi·reduce(r, r0, T, λj·e) (i′), reduce(r, r0, S, λi·reduce(r, r0, T, λj·e)))
  = *since* r, r0 *and* T *are independent of* i,
  move binding for i *into* reduce *(lemma 2)*
r(reduce(r, r0, T, λi·(λj·e) (i′)), reduce(r, r0, S, λi·reduce(r, r0, T, λj·e)))
  = *induction hypothesis; move binding for* i *into abstraction of* j *(lemma 3)*
r(reduce(r, r0, T, λj·(λi·e (i′))), reduce(r, r0, S × T, λij·e))

Right side:

reduce(r, r0, (S+i′) × T, λij·e)
  = *by definition of set insertion*
reduce(r, r0, (S ∪ {i′}) × T, λij·e)
  =
reduce(r, r0, (S × T) ∪ ({i′} × T), λij·e)
  = *split reduction (lemma 9)*
r(reduce(r, r0, S × T, λij·e), reduce(r, r0, {i′} × T, λij·e))
  = *collapse singleton dimensions in reduction (lemma 10);* r *commutes*
r(reduce(r, r0, T, λj·(λi·e (i′))), reduce(r, r0, S × T, λij·e))

Hence, by induction, the identity holds for all shapes.

□

8 Conclusions

The transformations required for converting basic array expressions into whole-array form have been shown to preserve the meaning of expressions. The majority of the proofs of correctness are simple and many are trivial; even the more complex proofs are straightforward to carry out, and use only well-known techniques (primarily induction over sets).

The simplicity of the proofs is due in large measure to the decomposition of a derivation into independent stages and to the decomposition of each stage into a sequence of simple transformations — the effect of each transformational step is small and is consequently readily amenable to formal analysis. In addition, the postponement of consideration of imperative details until very late in a derivation allows most of the transformational steps to be made within a purely functional framework; indeed, in this paper, it was not necessary to consider imperative details at all, even though the motive for applying the transformations is to tailor a program to the peculiarities of a particular type of imperative system.

Transformation sequences provide a means of characterising the special features of particular parallel architectures and/or problems domains (e.g. array processors or sparse matrix problems). The transformations relating to array processors presented in this paper may be reused in the derivation of efficient implementations of algorithms for the solution of a range of problems. Further, they are applied automatically by a tool. For these reasons, it is particularly important that the transformation sequence be *proved* to be meaning preserving.

A Example Application of Transformations

To illustrate the working of the transformations, they are here applied to matrix product. Assume a program contains an expression A*B where A and B are matrices of shape [n, n] and where * denotes matrix product. After translation into the λ-calculus, function unfolding and algebraic simplification, this expression becomes

```
generate([n, n], λ[i, j]·reduce(+, 0, [n], λ[k]·times(A@[i, k], B@[k, j]))
```

where times denotes multiplication of real numbers. This expression is the input to the Array Form transformations; their application proceeds as follows.

The generation and reduction are swapped (transformation 10).

```
→ reduce(ϵ(+), generate([n, n], λ[i, j]·0), [n],
  λ[k]·generate([n, n], λ[i, j]·times(A@[i, k], B@[k, j]))
```

The first generation creates a matrix in which all of the elements have the same value (0); such a generation can be expressed in the Array Form function as expand([n, n], 0) (the expand function corresponds to the 'broadcast' of a value to all of the processing elements in the processor array). In the second generation, the body of the generating function is an application of a scalar function for which elementwise application is supported; this generation is thus converted into an application of map (transformation 1).

```
→ reduce(ϵ(+), expand([n, n], 0), [n],
  λ[k]·map(
    generate([n,n], λ[i, j]·A@[i, k]),
    generate([n,n], λ[i, j]·B@[k, j]), times))
```

The body of the generating function of the first generation is independent of one of the generating indices, j; this generation can be expressed in the Array Form using the expand.rows function, which creates a matrix by duplicating a vector row-wise. (This function corresponds to a row-wise broadcast to the processor array). Similarly, the second generation can be expressed as a column-wise expansion since the body of its generating function is independent of i.

```
→ reduce(ϵ(+), expand([n, n], 0), [n],
  λ[k]·map(
    expand.cols([n], generate([n], λ[i]·A@[i, k])),
    expand.rows([n], generate([n], λ[j]·B@[k, j])),
    times))
```

The second generation corresponds to extracting a row of B (transformation 4); similarly, the first generation corresponds to extracting a column of A.

```
→ reduce(ϵ(+), expand([n, n], 0), [n],
  λ[k]·map(
    expand.cols([n], col(A, k)),
    expand.rows([n], row(B, k)),
    times))
```

The expression above is the output from the Array Form transformations; it may be further processed by subsequent derivation stages to produce an implementation for the DAP.

```
   real A(*n, *n), B(*n, *n)
   real P(*n, *n)
   integer k
   ⋮
   P = 0
   DO 10 k=1, n
   P = P+matc(A( , k), n)*matr(B(k, ), n)
10 CONTINUE
   ⋮
```

where

- P stores the product matrix;
- matr and matc are the DAP functions for row- and column-wise expansions;
- + and * denote elementwise addition and multiplication of two matrices;
- M(, k) and M(k,) denote, respectively, column k and row k of matrix M.

References

[Bar84] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. North-Holland, revised edition, 1984.

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

[Boy70] James M. Boyle. A transformational component for programming languages grammar. Technical Report ANL-7690, Argonne National Laboratory, July 1970.

[CFH$^+$93] M. Clint, Stephen Fitzpatrick, T.J. Harmer, P. Kilpatrick, and J.M. Boyle. A family of intermediate forms. Technical Report 1993/Nov-MC.SF.TJH.PLK.JMB, Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland, UK, November 1993.
<URL:http://www.cs.qub.ac.uk/pub/TechReports/1993/Nov-MC.SF.TJH.PLK.JMB/>.

[CFH$^+$94] M Clint, Stephen Fitzpatrick, T J Harmer, P L Kilpatrick, and J M Boyle. A family of data-parallel derivations. In Wolfgang Gentzsch and Uwe Harms, editors, *Proceedings of High Performance Computing and Networking, Volume II*, volume 797 of *Lecture Notes in Computer Science*, pages 457–462. Springer-Verlag, April 1994.

[Chu51] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951.

[FSCB95] Stephen Fitzpatrick, A. Stewart, M. Clint, and J.M. Boyle. The automated transformation of abstract specifications of numerical algorithms into efficient array processor implementations. Technical Report 1995/Jun-SF.AS.MC.JMB, Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland, UK, June 1995.
<URL:http://www.cs.qub.ac.uk/pub/TechReports/1995/Jun-SF.AS.MC.JMB/>.

[PL90] Dennis Parkinson and John Litt, editors. *Massively Parallel Computing with the DAP*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990.

[Wil87] A. Wilström. *Functional Programming using Standard ML*. Prentice Hall, 1987.