A Case Study on Proving Transformations Correct: Data-Parallel Conversion*

Stephen Fitzpatrick†, M. Clint‡ and P. Kilpatrick‡

† Kestrel Institute
3260 Hillview Avenue
Palo Alto
California 94304
U.S.A.

‡ Department of Computer Science
The Queen's University of Belfast,
Belfast BT7 1NN
Northern Ireland

Fitzpatrick@kestrel.edu, M.Clint@qub.ac.uk, P.Kilpatrick@qub.ac.uk

Abstract

The issue of correctness in the context of a certain style of program transformation is investigated. This style is characterised by the fully automated application of large numbers of simple transformation rules to a representation of a functional program (serving as a specification) to produce an equivalent efficient imperative program. The simplicity of the transformation rules ensures that the proofs of their correctness are straightforward.

A selection of transformations appropriate for use in a particular context are shown to preserve program meaning. The transformations convert array operations expressed as the application of a small number of general-purpose functions into applications of a large number of functions which are amenable to efficient implementation on an array processor.

Keywords: Correctness proofs, program transformation, functional programming, array processor

1 Introduction

In [1, 2], and elsewhere, a style of program derivation is advocated in which a complex change is wrought on a program through a number of simpler changes brought about by the automated application of a sequence of sets of simple transformations. For example, in [3] an imperative, explicitly data-parallel program is derived from a pure, functional program (which is used as an abstract specification) in six main steps.

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

The functional program (expressed in SML [4]) is translated into the λ-calculus; definitions are unfolded; expressions are simplified using algebraic rules; array expressions are translated into explicitly data-parallel (but still pure) forms; expressions are optimized; state and imperative control constructs are introduced to produce an imperative program (expressed in a variant of Fortran for execution on the AMT DAP array processor [5]).

One of the advantages claimed for this style of derivation is that establishing that program correctness is preserved at all stages is simplified by (i) the representation of a complex change as a sequence of

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

simpler, conceptually independent changes; (ii) the simplicity of the individual transformations; and (iii) regarding transformations as abstract rewrite rules rather than concentrating on their concrete effect on given programs.

This paper presents evidence to support this claim. It contains, for the stage in the derivation that produces the explicitly data-parallel form, a proof of correctness for each of the more complex transformations and a sample of proofs of correctness for the simple transformations. This stage is conceptually and technically significant. It is the first of the two pivotal stages in the derivation, the other being the translation from functional to imperative form. Note that the proofs are of themselves not of great interest — none of them contains novel proof techniques. Indeed, the interest lies in the fact that the proofs *are* mundane.

The structure of the paper is as follows: 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 stated (their proofs are relegated to the appendix); the transformations are defined; and their correctness is proved.

2 Basic Notation

.

The basic notation used in this paper is that of the λ-calculus with optional type information and a set of primitive functions. An array is considered to be a mapping from a (finite) set of indices onto some range of values of type α. A 1-dimensional array may be referred to as a *vector* and a 2-dimensional array as a *matrix*. For a set, the operator + indicates element insertion: $S+i \equiv S \cup \{i\}$.

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.

2.1 Primitive Array Functions

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

shape(A:α array) $\rightarrow$ Shape

Given an array, the function shape can be used to obtain its index set. The extent of an array in a particular dimension can be obtained by specifying the dimension:

shape(A:α array, n:int) $\rightarrow$ int

element(A:α array, i:index) $\rightarrow \alpha$

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

A@i $\equiv$ element(A, i)

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

The basic function for constructing arrays is generate. 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:

generate(shape(A), λi·A@i+B@i)

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

generate([n, m], λ[i, j]·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]·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(r:$\alpha \times \alpha \to \alpha$, r0:$\alpha$, S:Shape, g:index $\to \alpha$) $\to \alpha$

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.

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, reduce(+, 0, shape(A), λi·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], can be expressed as reduce(+, 0, [n], λi·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):

generate([shape(A, 1), shape(B, 2)],
λ[i, j]·reduce(+, 0, shape(A, 2), λ[k]·A@[i, k]*B@[k, j]))

- Boolean matrix, of shape [n, n], having leading diagonal elements true, and all other elements false:

generate([n, n], λ[i, j]·i=j)

2.1.1 Formal Definition of generate and reduce

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

$$\text{shape}(\text{generate}(S, \lambda i{\cdot}g)) \equiv S \qquad \text{(G1)}$$
$$\forall\, i' \in S:\ \text{element}(\text{generate}(S, \lambda i{\cdot}g), i') \equiv \lambda i{\cdot}g\ (i') \qquad \text{(G2)}$$

□

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 each 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

shape(generate([m, n], λi·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

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

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 5).

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

$$\text{reduce(r, r0, } \emptyset, \lambda i \cdot g) \equiv \text{r0} \quad \text{(R1)}$$
$$\text{reduce(r, r0, S+i', } \lambda i \cdot g) \equiv \text{r}(\lambda i \cdot g\ (i'), \text{reduce(r, r0, S, } \lambda i \cdot g)) \quad \text{(R2)}$$
where i' $\notin$ S *and* $\emptyset$ *denotes the empty set (of indices)*

□

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

2.2 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 corresponding elements of two arrays). Some of the functions, and some of the transformations discussed later, are perhaps peculiar to array processors in which the processing elements 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 for use on a particular processor.

2.2.1 Main Array Form Functions

See Figure 1 for examples. 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

$$\text{map(A:}\alpha \text{ array, f:}\alpha \to \beta) \to \beta \text{ array}$$
$$\stackrel{def}{=} \text{generate(shape(A), } \lambda i \cdot \text{f(A@i))}$$

$$\text{map(A:}\alpha \text{ array, B:}\beta \text{ array, f:}\alpha \times \beta \to \gamma) \to \gamma \text{ array}$$
$$\stackrel{def}{=} \text{generate(shape(A), } \lambda i \cdot \text{f(A@i, B@i))}$$
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 — this corresponds to operations on the DAP which compute multiple vector reductions simultaneously.

Definition 4: fold

$$\text{fold(r:}\alpha \times \alpha \to \alpha, \text{ r0:}\alpha, \text{ A:}\alpha \text{ array}) \to \alpha$$
$$\stackrel{def}{=} \text{reduce(r, r0, shape(A), } \lambda i \cdot \text{A@i)}$$

□

Definition 5: fold.rows

$$\text{map}\left(\left[\begin{array}{ccc}1&2&3\\4&5&6\\7&8&9\end{array}\right],\left[\begin{array}{ccc}1&2&3\\4&5&6\\7&8&9\end{array}\right],+\right)=\left[\begin{array}{ccc}2&4&6\\8&10&12\\14&16&18\end{array}\right]$$

$$\text{fold}\left(+,0,\left[\begin{array}{ccc}1&2&3\\4&5&6\\7&8&9\end{array}\right]\right)=45$$

$$\text{fold.rows}\left(+,\left[\begin{array}{c}0\\0\\0\end{array}\right],\left[\begin{array}{ccc}1&2&3\\4&5&6\\7&8&9\end{array}\right]\right)=\left[\begin{array}{c}6\\15\\24\end{array}\right]$$

$$\text{join}\left(\left[\begin{array}{ccc}\text{T}&\text{T}&\text{F}\\\text{F}&\text{F}&\text{F}\\\text{T}&\text{F}&\text{T}\end{array}\right],\left[\begin{array}{ccc}11&12&13\\14&15&16\\17&18&19\end{array}\right],\left[\begin{array}{ccc}21&22&23\\24&25&26\\27&28&29\end{array}\right],\right)=\left[\begin{array}{ccc}11&12&23\\24&25&26\\17&28&19\end{array}\right]$$

Figure 1: Examples of Array Form functions

$$\begin{array}{l}\text{fold.rows(r:}\alpha\times\alpha\to\alpha\text{, R0:}\alpha\text{ vector, A:}\alpha\text{ matrix)}\to\alpha\text{ vector}\\ \stackrel{def}{=}\text{ generate([shape(A, 1)], }\lambda\text{[i]·reduce(r, R0@[i], [shape(A, 2)], }\lambda\text{[j]·A@[i, j]))}\end{array}$$

□

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

$$\begin{array}{l}\text{join(M:boolean array, T:}\alpha\text{ array, F:}\alpha\text{ array)}\to\alpha\text{ array}\\ \stackrel{def}{=}\text{ generate(shape(M), }\lambda\text{i·if M@i then T@i else F@i)}\\ \quad\textit{where}\text{ M, T }\textit{and}\text{ F }\textit{have the same Shape}\end{array}$$

□

2.2.2 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

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

□

Definition 8: row

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

□

3 Preliminary Results

Some properties of generate and reduce are presented below — proofs of these properties are presented in the appendix. In addition, some elementary identities of the λ-calculus are noted (without proof).

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

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

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.

$$\lambda \mathsf{x} \cdot \mathsf{B}\ (\mathsf{x}) \equiv \mathsf{B}$$

□

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 the abstraction.

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

(Note that x is still bound to 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.

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

□

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.

3.1 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 promote a binary function to a binary function on arrays. For example, ϵ(+) is a function that performs elementwise addition of two arrays.

Definition 9: Elementwise Operator, ϵ

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

□

Lemma 4: 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).

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

□

Lemma 5: 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:

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

□

3.2 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

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

is called an ϵ-reduction.
□

The following two lemmas pertain to ϵ-reductions.

Lemma 6: Shape of an ϵ-reduction
The result of an ϵ-reduction has the same shape as the reduction's initial value:

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

□

Lemma 7: 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.

$$\text{element}(\text{reduce}(\epsilon(\text{r}), \text{R0}, \text{S}, \lambda\text{i}{\cdot}\text{g}), \text{j})$$
$$\equiv \text{reduce}(\text{r}, \text{element}(\text{R0}, \text{j}), \text{S}, \lambda\text{i}{\cdot}\text{element}(\text{g}, \text{j}))$$
$$\textit{where } \text{S} \textit{ is independent of } \text{j}$$

□

Lemma 8: 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.

$$\text{reduce}(\text{r}, \text{r0}, \text{S} \cup \text{T}, \lambda\text{i}{\cdot}\text{e}) \equiv \text{r}(\text{reduce}(\text{r}, \text{r0}, \text{S}, \lambda\text{i}{\cdot}\text{e}), \text{reduce}(\text{r}, \text{r0}, \text{T}, \lambda\text{i}{\cdot}\text{e}))$$
$$\textit{where } \text{r0} \textit{ is an identity element of } \text{r}, \textit{ and } \text{S} \textit{ and } \text{T} \textit{ are disjoint}$$

□

Lemma 9: 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 $\text{–i}'' \times \text{S}$).

$$\text{reduce}(\text{r}, \text{r0}, \text{–i}'' \times \text{S}, \lambda\text{ij}{\cdot}\text{e}) \equiv \text{reduce}(\text{r}, \text{r0}, \text{S}, \lambda\text{j}{\cdot}(\lambda\text{i}{\cdot}\text{e}\ (\text{i}')))$$
$$\textit{where } \text{r0} \textit{ is an identity element of } \text{r}$$
$$\textit{and where } \text{i}' \textit{ and } \text{i} \textit{ have the same dimensionality}$$

□

4 The Transformations

In this section, the main transformations (actually, identities) 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 [3]. In addition, several transformations optimize combinations of operations to make best use of the DAP hardware.

4.1 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

$$\text{generate}(S, \lambda i{\cdot}f(a)) \equiv \text{map}(\text{generate}(S, \lambda i{\cdot}a), f)$$
$$\text{generate}(S, \lambda i{\cdot}f(a, b)) \equiv \text{map}(\text{generate}(S, \lambda i{\cdot}a), \text{generate}(S, \lambda i{\cdot}b), f)$$
where f *is a scalar function for which elementwise application is supported by* map

□

(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

$$\text{generate}(S, \lambda i{\cdot}\text{if } p \text{ then } t \text{ else } f)$$
$$\equiv \text{join}(\text{generate}(S, \lambda i{\cdot}p), \text{generate}(S, \lambda i{\cdot}t), \text{generate}(S, \lambda i{\cdot}f))$$

□

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

Transformation 3: matrix.transpose

$$\text{generate}([m, n], \lambda[i, j]{\cdot}A@[j, i]) \equiv \text{matrix.transpose}(A)$$
where A *has shape* [n, m]

□

Transformation 4: row

$$\text{generate}([m], \lambda[j]{\cdot}A@[r, j]) \equiv \text{row}(A, r)$$
where A *has shape* [n, m] *and* A *and* r *are independent of* j

□

4.2 Propagation through λ-expressions

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

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

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:

$\forall$ i $\in$ S: evaluate e;
$\forall$ i $\in$ 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 applies 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

generate(S, λi·λx·B (e)) $\equiv$ λx·generate(S, λi·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

generate(S, λi·(λx·B (e)))
$\equiv$ λX·generate(S, λi·(λx·B (X@[i]))) (generate(S, λi·e))
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

generate([l, m], λ[i, j]·(λx·B (generate([n], λ[k]·e))))
$\equiv$ λX·generate([l, m], λ[i, j]·(λx·B (generate([n], λ[k]·X@[i, k]))))
(generate([l, n], λ[i, k]·e))
where e *is a function of* i *and* k *but not of* j, *and is not an application of* element

□

As it stands, 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.[1] 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.

[1] 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 8: Binding is a function application

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

□

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.

4.3 Transformations for Reductions

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

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

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.) [2]

The following two transformations enforce these preferences.

Transformation 9: generate-reduce combination

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

□

Transformation 10: generate-reduce swap

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

□

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.

[2] 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).

Transformation 11: reduce-reduce combination

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

where r0 *is an identity element of* r, *and* r *and* T *are independent of* i

□

5 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.

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

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

$$\text{generate(S, } \lambda\text{j}\cdot\text{f(}\lambda\text{i}\cdot\text{a (j), } \lambda\text{i}\cdot\text{b (j)))}$$

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

$$\text{generate(S, } \lambda\text{j}\cdot(\lambda\text{i}\cdot\text{f(a, b) (j)))}$$

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

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

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

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

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

□

5.1 Proofs of Transformations for λ-bindings

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

Proof 5: Invariant binding

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

$$\text{generate}(S, \lambda i\cdot(\lambda x\cdot B\ (e)))$$
$$\equiv \lambda X\cdot\text{generate}(S, \lambda i\cdot(\lambda x\cdot B\ (X@[i]))\ (\text{generate}(S, \lambda i\cdot e))$$
where e *is a scalar*

Proof follows by β-reducing X:

λX·generate(S, λi·(λx·B (X@[i]))) (generate(S, λi·e))
= *β-reduce* X*; only occurrence of* X *is that shown*
generate(S, λi·(λx·B (element(generate(S, λi·e), i))))
= *by G2*
generate(S, λi·(λx·B (λi·e (i))))
= *remove identity binding (lemma 1)*
generate(S, λi·(λx·B (e)))

□

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

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

where e *is depednent on* i *and* k*, but not on* j

Proof follows by β-reducing X:

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

□

Proof 8: Binding is a function application

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

Proof follows by β-reducing x′:

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

□

5.2 Proofs of Transformations for Reductions

Proof 9: generate-reduce combination

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

where n *and* r *are independent of* i

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

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

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 6*
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 7)*

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′, where i′ $\notin$ S.

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 $\times$ 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 8)*

r(reduce(r, r0, S × T, λij·e), reduce(r, r0, –i′″ × T, λij·e))

= *collapse singleton dimensions in reduction (lemma 9);* 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.

□

6 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 Proofs of Preliminary Lemmas

Some basic properties of generate and reduce are established below.

A.1 Properties of Elementwise Applications

Proof of Lemma 4: Shape of an elementwise application

shape(ϵ(f)(A, B)) ≡ shape(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)

□

Proof of Lemma 5: Element of an elementwise application

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

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′)

□

A.2 Properties of Reductions

Proof of Lemma 6: Shape of an ϵ-reduction

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

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′, where i′ $\notin$ S.

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 4*
shape(reduce(ϵ(r), R0, S, λi·g))
= *by induction hypothesis*
shape(R0)

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

□

Proof of Lemma 7: Element of an ϵ-reduction

$$\text{element}(\text{reduce}(\epsilon(\text{r}), \text{R0}, \text{S}, \lambda\text{i}\cdot\text{g}), \text{j}) \equiv \text{reduce}(\text{r}, \text{element}(\text{R0}, \text{j}), \text{S}, \lambda\text{i}\cdot\text{element}(\text{g}, \text{j}))$$
$$\textit{where } \text{S} \textit{ is independent of } \text{j}$$

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′, where i′ $\notin$ S.

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 5*

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:

reduce(r, element(R0, j), S+i′, λi·element(g, j))

= *by R2*

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

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

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

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

□

Proof of Lemma 8: Reduction over a union

reduce(r, r0, S $\cup$ T, λi·e) $\equiv$ r(reduce(r, r0, S, λi·e), reduce(r, r0, T, λi·e))

where r0 *is an identity element of* r, *and* S *and* T *are disjoint*

Proof. Proof is by induction over S.

Base Step: $\emptyset$

Left side:

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

=

reduce(r, r0, T, λi·e)

Right side:

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

= *by R1*

r(r0, reduce(r, r0, T, λi·e))

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

reduce(r, r0, T, λi·e)

Inductive Step: S+i'

Assume lemma holds for shape S. Now consider shape S+i′, where $i' \notin S \cup T$.

Left side:

reduce(r, r0, (S+i′) ∪ T, λi·e)

= *interchanging set insertion and union*

reduce(r, r0, (S ∪ T)+i′, λi·e)

= *by R2*

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

= *by induction hypothesis*

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

Right side:

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

= *by R2*

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

= *since* r *is associative*

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

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

□

Proof of Lemma 9: Collapsing singleton dimensions

reduce(r, r0, –i′″ × S, λij·e) ≡ reduce(r, r0, S, λj·(λi·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: ∅

Left side:

reduce(r, r0, –i′″ × ∅, λij·e)

=

reduce(r, r0, ∅, λij·e)

= *by R1*

r0

Right side:

reduce(r, r0, ∅, λj·(λi·e (i′)))

= *by R1*

r0

Inductive Step: S+j'

Assume lemma holds for shape S. Now consider shape S+j′, where $j' \notin S$.

Left side:

reduce(r, r0, –i′″ × (S+j′), λij·e)

= *by definition of set insertion*

reduce(r, r0, –i′″ × (S ∪ –j′″), λij·e)

=

reduce(r, r0, (–i′″ × S) ∪ (–i′″ × –j′″), λij·e)

= *split index set (lemma 8)*

r(reduce(r, r0, –i′″ × S, λij·e), reduce(r, r0, –i′″ × –j′″, λij·e))

= *induction hypothesis; cartesian product of singleton sets is a singleton set*

r(reduce(r, r0, S, λj·(λi·e (i′))), reduce(r, r0, –i′j′″, λij·e))

= *by R2*

r(reduce(r, r0, S, λj·(λi·e (i′))), r(λij·e (i′j′), reduce(r, r0, ∅, λij·e)))

= *by R1 and by definition of index concatenation*

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

= r0 *is an identity of* r*; Currying*

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

Right side:

reduce(r, r0, S+j′, λj·(λi·e (i′)))

= *by R2*

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

= *since* r *is commutative*

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

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

□

References

[1] 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.

[2] 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/¿.

[3] Stephen Fitzpatrick, Terence J. Harmer, Alan Stewart, Maurice Clint, and James M. Boyle. The automated transformation of abstract specifications of numerical algorithms into efficient array processor implementations. *The Science Of Computer Programming*, 28(1), January 1997.

[4] A. Wikström. *Functional Programming using Standard ML*. Prentice Hall, 1987.

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