A Calculational Approach to Flattening Nested Data Parallelism in Functional Languages

Gabriele Keller and Martin Simons

Technische Universität Berlin
Forschungsgruppe Softwaretechnik*

Abstract. The data-parallel programming model is currently the most successful model for programming massively parallel computers. Unfortunately, it is, in its present form, restricted to exploiting *flat* data parallelism, which is not suitable for some classes of algorithms, e.g. those operating on irregular structures. Recently, some effort has been made to implement *nested* data-parallel programs efficiently by compiling them into equivalent flat programs using a transformation called *flattening*. However, previous translations of nested into flat data-parallel programs have proved unwieldy when it comes to inventing and specifying optimizations and verifying the translation. This paper presents a new formalization of the flattening transformation in a calculational style. The formalization is easily verified and provides a good starting point for the development of new optimizations. Some optimizations invented on the basis of this new formalism are described. Furthermore, we present practical evidence obtained by experimenting with an implementation of the transformation.

Keywords: parallel programming; functional programming; nested data parallelism; flattening transformation; implementation; calculational method.

1 Introduction

Today's most successful approaches to parallel programming are based on the data-parallel programming model. There are two main reasons for this: first, such programming models are comparatively easy to use from the programmer's point of view; second, their simplicity has led to a number of efficient implementations. They are, however, in the form they are mostly used — as *flat* data parallelism — quite restricted. Languages based on this kind of parallelism, like C* [19] and HPF [12], make it difficult to fully exploit the parallelism of algorithms that work on irregular data structures. On the other hand, this restriction facilitates the generation of efficient code.

Nested data parallelism, as employed, e.g. in Nesl [8], Paralation Lisp [18], and Proteus [16], overcomes this restriction [6] while still having the potential for efficient implementation. Blelloch [4] provided the basis for such implementations by showing, for a functional language that was extended to express nested data parallelism, that such a language can be transformed mechanically into code using flat data parallelism only. The results, given in [8], show that this code can be executed efficiently on a wide range of parallel machines, from vector computers to SIMD and MIMD machines.

* TU Berlin, Forschungsgruppe Softwaretechnik (FR5-6), Franklinstr. 28/29, D-10587 Berlin. e-mail: {keller,simons}@cs.tu-berlin.de

```
Expr     :=  Var | Const                       — variables and constant expressions
          |  FunName ([ExprList])              — function application
          |  {[ExprList]}                      — vector construction
          |  let (Var = Expr)+ in Expr         — let binding
          |  if Expr then Expr else Expr       — case distinction
          |  {Expr | Generators : Expr}        — apply-to-each (source language only)

ExprList  :=  Expr                Generators  :=  Var ← Expr
           |  Expr, ExprList                   |  Var ← Expr, Generators
```

Fig. 1. Syntax of the source and the target languages

Although the basics of the transformation are clear, applying it to a concrete language easily leads to unwieldy and complicated transformation rules: their implementation becomes error-prone, and possible opportunities for optimization are hidden. In addition, extending the transformation to handle imperative features adds considerably to the difficulties encountered with functional languages [1, 9]. These problems call for a simple and compact formalism to describe the transformation. This paper introduces such a formalism, specifically designed to simplify *calculations* [2]. Moreover, we present the flattening transformation completely within this formal framework to facilitate correctness proofs and to make the calculational derivation of optimizations possible.

The paper is organized as follows. In Sect. 2, we introduce the formalism and state some simple laws. Sect. 3 presents the basic transformation rules controling the flattening process. In Sect. 4, we derive several optimizations of the basic set of rules. Sect. 5 concludes the paper by presenting some performance data obtained from an experimental implementation of the flattening transformation, as well as a brief review of related and a discussion of future work.

2 Nested Data Parallelism

Efficient implementations of nested data parallelism currently rely on the fact that the latter can be transformed mechanically into flat data parallelism. To express this *flattening* transformation formally, we introduce two functional languages: a *source language* allowing the expression of nested data parallelism, and a *target language* allowing the expression of flat data parallelism only. Both languages share the same syntax (see Fig. 1) the only difference being that the construct used to express nested data parallelism — apply-to-each — is unavailable in the target language. Flattening is specified in Sect. 3 by a set of rewrite rules translating source into equivalent target expressions.

2.1 The Source Language

The source language is a strongly typed, strict, first-order functional language. It has one class of type constructors: vector constructors $\{\cdot\}$ of arbitrary but finite arity. Vectors, like lists in Haskell or ML, are linearly ordered, homogeneously typed sets of arbitrary but finite size, and can also contain vectors of the same type as elements. They play a central role, since parallelism can only be expressed by operations on vectors. The language offers a set of primitive parallel operations on vectors, some of which are described in Tab. 1. General parallel computations are specified by the

$\# :: \{\alpha\} \to \text{Int}$	length of a vector
$! :: \{\alpha\} \times \text{Int} \to \alpha$	$xs!n$ returns the nth element of vector xs.
$+\!\!+ :: \{\alpha\} \times \{\alpha\} \to \{\alpha\}$	concatenation of two vectors
$\text{dist} :: \alpha \times \text{Int} \to \{\alpha\}$	$\text{dist}(x, n)$ creates a vector of length n with xs as its elements; we denote $\text{dist}(x, \#xs)$ by $x \bullet xs$
$\text{permute} :: \{\alpha\} \times \{\text{Int}\} \to \{\alpha\}$	constructs the permutation of a given vector according to a given permutation vector
$\text{pack} :: \{\alpha\} \times \{\text{Bool}\} \to \{\alpha\}$	constructs a vector by taking from a given vector all elements corresponding to a true value in a given flag vector
$\oplus\text{\_scan} :: \{\alpha\} \to \{\beta\}$	for each monoid operator $\oplus$ with identity $1_\oplus$ (e.g. $+, *, \min, \max$), $\oplus\text{\_scan}(\{x_1, \ldots, x_n\})$ computes the prefix sum $\{1_\oplus, x_1, \ldots, x_1 \oplus \cdots \oplus x_{n-1}\}$
$\oplus\text{\_reduce} :: \{\alpha\} \to \beta$	$\oplus\text{\_reduce}(\{x_1, \ldots, x_n\})$ computes the generalized sum of the argument vector $x_1 \oplus \cdots \oplus x_n$.
$\mathcal{F} :: \{\{\alpha\}\} \to \{\alpha\}$	discards the top-level structure of a nested vector
$\mathcal{P} :: \{\{\beta\}\} \times \{\alpha\} \to \{\{\alpha\}\}$	given a nested vector xs and a vector ys such that $\#ys = \#\mathcal{F}(xs)$, $\mathcal{P}_{xs}(ys)$ denotes the partition of ys induced by the top-level structure of xs

Tab. 1. Some primitive parallel operations

"apply-to-each" construct whose notation is derived from list-comprehension: the expression $\{f\,x \mid x \leftarrow xs : p\,x\}$ denotes the vector that results from evaluating *in parallel* the body $f\,x$ for all those elements x taken from the vector xs for which the Boolean expression $p\,x$ evaluates to true. For instance, the expression $\{2 * x \mid x \leftarrow \{1, 2, 5, 8\} : \text{even}(x)\}$ evaluates to $\{4, 16\}$. An apply-to-each can have multiple generators, as in $\{x + y \mid x \leftarrow \{1, 2\}, y \leftarrow \{5, 8\}\}$, in which case the bindings are evaluated in a *lockstep* fashion provided that the generating vectors have the same length. Thus, the last expression evaluates to $\{6, 10\}$. This distinguishes "vector-comprehension" from list-comprehension, where bindings are evaluated in a combinatory fashion.

The only way to express *nested* parallel computations is to make the body of an apply-to-each itself a parallel computation. Let us illustrate the use of nested data parallelism by a small example. Quicksort, a classical divide-and-conquer algorithm, contains two sources of parallelism. First, the division of the problem into subproblems can be solved in parallel, i.e. it can be decided in parallel for each element of a vector whether it is less than, equal to, or greater than a fixed pivot element. This kind of parallelism within a function is called *intrafunction parallelism*.

```
$qsort(xs) =$
    if $\#xs \leq 1$ then $xs$
    else let $m = xs[\#xs/2]$
             $s = \{x \mid x \leftarrow xs : x < m\}$
             $e = \{x \mid x \leftarrow xs : x = m\}$
             $g = \{x \mid x \leftarrow xs : x > m\}$
             $sorted = \{qsort(ys) \mid ys \leftarrow \{s, g\}\}$
         in $sorted[0] +\!\!+ e +\!\!+ sorted[1]$
```

Second, the subproblems themselves can be solved in parallel by recursively applying Quicksort to them. This kind of parallelism between function calls is called *interfunction parallelism*. In a flat data parallel language, one would — without a substantial amount of additional coding effort — only be able to exploit one of the sources of parallelism in this algorithm, because exploiting both requires parallel computations within parallel computations. In our source language, on the other hand, the subproblems are first computed in parallel by three flat apply-to-each expressions, and then recursively solved in parallel by means of a nested apply-to-each.

2.2 The Target Language

Omitting the apply-to-each construct in the target language means that we lose the ability to express nested parallel computations. However, the apply-to-each also allowed us to express general flat parallel evaluation of expressions. In order to recover this ability, we introduce a functional called *lifting*: lifting maps a function $f :: \alpha \to \beta$ to the function $f^\uparrow :: \{\alpha\} \to \{\beta\}$ such that semantically and operationally $f^\uparrow(xs) = \{f(x) \mid x \leftarrow xs\}$. In other words, $f^\uparrow$ denotes the computation that applies f in parallel to all the elements of the argument vector; f is *lifted* to the level of vectors. This highlights the fact that in the target language all parallel computation is expressed at the level of vectors; the individual elements of the vectors are not visible. This, then, appears to be the right level of abstraction for implementing data parallelism.

2.3 Simple Laws

A subtle difference exists between lifting and the map functional $*$ known from functional languages, or the list calculus [3]: in accordance with the lockstep evaluation of multiple generator bindings, lifting maps an n-ary function $f :: \alpha_1 \times \cdots \times \alpha_n \to \beta$ to $f^\uparrow$ of type $\{\alpha_1\} \times \cdots \times \{\alpha_n\} \to \{\beta\}$, which is a partial function that is only defined if all argument vectors have the same length. Consequently, $f^\uparrow$ is, in general, not equal to $f* :: \{\alpha_1 \times \cdots \times \alpha_n\} \to \{\beta\}$. Another partial function is the vector constructor, which accepts only tuples of homogeneously typed arguments. Calculation with partial functions is addressed by Gries and Schneider [11], and we follow their approach. However, when stating laws, we tacitly drop hypotheses requiring equal length or homogeneity of types of the arguments whenever these constraints can be easily derived from the context. Below we denote multiple lifting of a function f by $f^{(n\uparrow)} = (f^{(n-1\uparrow)})^\uparrow$. This should be distinguished from the repeated composition of f, denoted by $f^n = f \circ f^{n-1}$.

The target language supplies for each primitive operation its lifted version. It also provides lifted versions of the vector constructor which we denote by $\{\cdot\}^\uparrow$. Moreover, a lifted variant of the case distinction of type $\{\text{Bool}\} \times \{\alpha\} \times \{\alpha\} \to \{\alpha\}$ exists. It constructs a result vector by making a selection from either its second or third argument vector, depending on the corresponding Boolean value in its first argument vector.[1]

For our subsequent calculations, we now introduce two meta level operations. First,

$$\pi_i^n :: \alpha_1 \times \cdots \times \alpha_n \to \alpha_i, \quad 1 \leq i \leq n$$

denotes the projection along the ith component of an n-ary product, i.e. it is defined by $\pi_i^n = \lambda(x_1, \ldots, x_n).x_i$. Second,

$$\text{zip}^n :: \{\alpha_1\} \times \cdots \times \{\alpha_n\} \to \{\alpha_1 \times \cdots \times \alpha_n\}$$

denotes the lifted version of the n-ary Cartesian product, i.e., given n vectors of equal length, zip^n constructs the corresponding vector of products. Note that zip^1 is the identity. By abuse of notation, we subsequently drop the index n and calculate with π_i and zip only. We can now specify the relationship between lifting and map:

[1] Strictly speaking, this lifted operation is not primitive, since it is no data-parallel computation. It can be expressed in terms of two primitive data-parallel operations—pack and combine—which, without loss of generality and for the sake of brevity, we do not consider in this paper.

(Meta-1) *Lifting-map*: $f^{\uparrow} = f* \circ \text{zip}$

and, since $\pi_i * \circ \text{zip} = \pi_i$:

(Meta-2) *Lifted projection*: $\pi_i{}^{\uparrow} = \pi_i$

Furthermore, we can establish the following relationships:

(Meta-3) *Selection-projection*: $(!i) \circ \{\cdot\} = \pi_i$

(Meta-4) *Selection-map*: $(!i) \circ f* = f \circ (!i)$

Lifting plays an important part in the transformation rules of Sect. 3, where it is used to describe how the body expression of an apply-to-each is "lifted" from its pointwise form to a corresponding vectorwise form. An equally important part is played by the two primitive operations flatten $\mathcal{F}$ and partition $\mathcal{P}$ (see Tab. 1). Recall that $\mathcal{F}$ takes a vector of vectors and concatenates the subvectors. Thus, $\mathcal{F} :: \{\{\alpha\}\} \rightarrow \{\alpha\}$ *removes* the top-level nesting structure of a nested vector. Conversely, partition $\mathcal{P} :: \{\{\beta\}\} \times \{\alpha\} \rightarrow \{\{\alpha\}\}$ *imprints* the topmost nesting structure of a given nested vector on a second vector. Partition is partial, as described in Tab. 1. Removing the top n nesting levels is denoted by $\mathcal{F}^n$. Similarly, $\mathcal{P}^n_{xs}$ denotes the partitioning according to the top n levels of a nested vector xs, i.e. $\mathcal{P}^n_{xs} = \mathcal{P}_{xs} \circ \mathcal{P}^{n-1}_{\mathcal{F}(xs)}$. By iterating the condition on $\mathcal{P}$ we find that $\mathcal{P}^n_{xs}(ys)$ is defined if $\mathcal{P}^{n-1}_{\mathcal{F}(xs)}(ys)$ is defined and $\#\mathcal{P}^{n-1}_{\mathcal{F}(xs)}(ys) = \#\mathcal{F}(xs)$. The next two laws state an inverse relationship between $\mathcal{F}$ and $\mathcal{P}$:

(Flatten-1) *Left inverse*: $(\mathcal{P}^n_{xs} \circ \mathcal{F}^n)\ xs = xs$

(Flatten-2) *Right inverse*: $(\mathcal{F}^n \circ \mathcal{P}^n_{xs})\ ys = ys$

The application of a k-fold lifted m-ary function to an m-tuple $\overline{xs}$ of nested vectors does not alter the nesting structure up to level k. In particular, we may partition with respect to the structure of any component $\overline{xs}_i$ because they are equal:

(Flatten-3) *Partition*: $\mathcal{P}^n_{f^{(k\uparrow)}(\overline{xs})} = \mathcal{P}^n_{\overline{xs}_i}, \quad k \geq n$

Counterparts to the map-reduce-promotion and map-promotion laws of the list calculus are the following two laws; let k denote the arity of f:

(Flatten-4) *Lifting-flatten promotion*: $\mathcal{F} \circ f^{(n+1\uparrow)} = f^{(n\uparrow)} \circ \underbrace{(\mathcal{F} \times \cdots \times \mathcal{F})}_{k \text{ times}}, \quad n \geq 1$

(Flatten-5) *Lifting promotion*: $f^{\uparrow} \circ g^{\uparrow} = (f \circ g)^{\uparrow}$

To prove the latter rule, note that typing constraints force f to be unary.

3 Basic Flattening Transformation

Translating nested data-parallel programs into flat programs requires the transformation of a "pointwise" apply-to-each, possibly expressing nested parallelism, into an equivalent "vectorwise" function of the target language. It is crucial that this transformation, known as "flattening nested data parallelism" [4], preserves the degree of parallelism specified in the original program [15]. We now present a basic set of transformation rules for flattening nested parallelism in terms of the algebraic framework of the previous section. The rules control three separate tasks: elimination of the apply-to-each; lifting user-defined functions; and reduction of multiply lifted to singly lifted functions.

Eliminating the Apply-to-Each Construct. Without loss of generality, we assume that apply-to-each constructs are of the form $\{e \mid \bar{x} \leftarrow xs\}$, where $\bar{x}$ may be a variable of product type (with explicitly named components), in which case xs is a vector of products. This normal form can be generated by the following set of rules:

$$\begin{aligned} \{e \mid x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n : p\} &= \{e \mid (x_1, \ldots, x_n) \leftarrow \text{zip}(e_1, \ldots, e_n) : p\} \\ \{e \mid \bar{x} \leftarrow e' : p\} &= \{e \mid \bar{x} \leftarrow \text{pack}(e', (\lambda\bar{x}.p) * e')\} \\ \{e \mid \bar{x} \leftarrow e'\} &= \textbf{let } xs = e' \textbf{ in } \{e \mid \bar{x} \leftarrow xs\} \end{aligned}$$

The remaining apply-to-each expressions are replaced by a meta-level map expression:

$$\{e \mid \bar{x} \leftarrow xs\} = (\lambda\bar{x}.e) * xs$$

By this process, we end up with an expression from which all apply-to-each constructs are removed, but at the cost of introducing meta-level map constructs and lambda terms.

The following set of transformation rules successively replace these newly introduced terms by constructs of our target language. We start with two simple cases: mapping the identity or a constant expression over a vector ($c \bullet xs = \text{dist}(c, \#xs)$):

(Trafo-1) *Identity*: $(\lambda\bar{x}.\bar{x}) * xs = xs$

(Trafo-2) *Constant*: $(\lambda\bar{x}.c) * xs = c \bullet xs$

Consider next the case $(\lambda x.f(\bar{e})) * xs$, where f is either primitive or user-defined. We can lift f and factor it out of the abstraction. This also holds if f is already lifted; we then merely lift it once more. Thus, by writing $f^{(0\uparrow)}$ for f, we arrive at the rule:

(Trafo-3) *Application*:

$$(\lambda\bar{x}.f^{(n\uparrow)}(e_1, \ldots, e_n)) * xs = f^{(n+1\uparrow)}((\lambda\bar{x}.e_1) * xs, \ldots, (\lambda\bar{x}.e_n) * xs)$$

Note that multiply lifted functions are not part of the target language. They need to be further transformed, which is the job of the last set of rules given below.

Finally, we consider let bindings and case distinctions in the body of an abstraction:

(Trafo-4) *Let binding*: $(\lambda\bar{x}.\textbf{let } a = b \textbf{ in } e) * xs = \textbf{let } as = (\lambda\bar{x}.b) * xs$ $\textbf{in } (\lambda(\bar{x}, a).e) * \text{zip}(xs, as)$

(Trafo-5) *Case distinction*:

$$(\lambda\bar{x}.\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2) * xs = \textbf{if}^{\uparrow}\ (\lambda\bar{x}.b) * xs\ \textbf{then}^{\uparrow}\ (\lambda\bar{x}.e_1) * xs\ \textbf{else}^{\uparrow}\ (\lambda\bar{x}.e_2) * xs$$

Lifting Defined Functions. Lifting defined functions is expressed in terms of lifted primitive operations, which are part of the target language. For each function definition

$$f :: \alpha_1 \times \cdots \times \alpha_n \to \beta \qquad f(x_1, \ldots, x_n) = e$$

we construct the lifting $f^{\uparrow} :: \{\alpha_1\} \times \cdots \times \{\alpha_n\} \to \{\beta\}$ by applying the rule:

(Trafo-6) *Defined functions*: $f^{\uparrow}(\overline{xs}) = (\lambda(x_1, \ldots, x_n).e) * \text{zip}(\overline{xs})$

The right-hand side is then further transformed by the other rules.

Eliminating Multiply Lifted Functions. As we have seen, the rule Trafo-3 may introduce multiply lifted functions, but, by using $\mathcal{F}$ and $\mathcal{P}$, multiple lifting can be expressed in terms of single lifting. (Here, $\overline{\mathcal{F}}$ denotes the appropriate product of $\mathcal{F}$s.)

$$\begin{aligned} f^{(n\uparrow)}(\overline{xs}) &= (\mathcal{P}^{n-1}_{f^{(n\uparrow)}(\overline{xs})} \circ \mathcal{F}^{n-1} \circ f^{(n\uparrow)})(\overline{xs}) && \{\text{Flatten-1}\} \\ &= (\mathcal{P}^{n-1}_{\overline{xs}_1} \circ f^{\uparrow} \circ \overline{\mathcal{F}^{n-1}})(\overline{xs}) && \{\text{Flatten-3, Flatten-4 repeatedly}\} \end{aligned}$$

Hence, multiply lifted functions can be eliminated by the repeated application of $\mathcal{F}$ and $\mathcal{P}$. Operationally, they merely alter the structure, no computation being involved.

(Trafo-7) *Multiply lifted functions*: $f^{(n\uparrow)}(\overline{xs}) = (\mathcal{P}^{n-1}_{\overline{xs}_1} \circ f^{\uparrow} \circ \overline{\mathcal{F}^{n-1}})(\overline{xs})$

Note that this transformation preserves the degree of parallelism of the left-hand side: f is still applied to all elements of the argument vector in a single parallel step.

4 Improved Transformation Rules

This section presents improvements on the transformation rules given in the previous section. We prove the correctness of these optimizations but only give indications as to why they improve the original transformation.

Lifting defined functions. For any f, we have the identity $f^{\uparrow}(\overline{xs}) = \textbf{if } \overline{xs}_1 == \{\} \textbf{ then } \{\} \textbf{ else } f^{\uparrow}(\overline{xs})$. By unfolding Trafo-6 with this identity, we obtain:

(Opt-1) *Defined functions*:

$$f^{\uparrow}(\overline{xs}) = \textbf{if } \overline{xs}_1 = \{\} \textbf{ then } \{\} \textbf{ else } (\lambda(x_1, \ldots, x_n).e) * \text{zip}(\overline{xs})$$

This rule prevents the trivial vector from being passed down the expression tree until the primitive lifted functions have a chance to recognize it.

Distributing constant values over nested vectors. Whenever a function that depends on a constant is applied to each element of a vector, the constant has to be distributed over the vector. Consider the following identity where the left-hand side expresses the addition of a constant c to each element of a nested vector X.

$$\{\{c + x \mid x \leftarrow xs\} \mid xs \leftarrow X\} = \mathcal{P}_X(\mathcal{F}((c \bullet X) \bullet^{\uparrow} X) +^{\uparrow} \mathcal{F}(X))$$

The evaluation of $c \bullet X$ is realized by simply broadcasting c, but $\bullet^{\uparrow}$ requires an expensive general communication operation, which in this case can be avoided:

$$\begin{aligned}
(c \bullet X) \bullet^{\uparrow} X &= (\lambda x.c) * X \ \bullet^{\uparrow} \ (\lambda x.x) * X && \{\text{Trafo-2,Trafo-1}\}\\
&= (\lambda y.c \bullet y) * X && \{\text{Trafo-3}\}\\
&= (\lambda x.c) * * X && \{\text{Trafo-2, } \eta\text{-conversion}\}\\
&= \mathcal{P}_X(\mathcal{F}((\lambda x.c) * * X)) && \{\text{Flatten-1}\}\\
&= \mathcal{P}_X(c \bullet \mathcal{F}(X)) && \{\text{map reduce promotion, Trafo-2}\}
\end{aligned}$$

Thus, lifted distributes can be replaced by a simple broadcast:

(Opt-2) *Distributing constants over nested vectors*:

$$(c \bullet X) \bullet^{\uparrow} X = \mathcal{P}_X(c \bullet \mathcal{F}(X))$$

This rule also covers nesting depth greater than 2. For instance, let X be of nesting depth three, then the resulting expression can be further transformed:

$$\begin{aligned}
\mathcal{P}_X(\mathcal{F}((c \bullet X) \bullet^{\uparrow} X) \bullet^{\uparrow} \mathcal{F}(X)) &= \mathcal{P}_X(\mathcal{F}(\mathcal{P}_X(c \bullet \mathcal{F}(X))) \bullet^{\uparrow} \mathcal{F}(X)) && \{\text{Opt-2}\}\\
&= \mathcal{P}_X(c \bullet \mathcal{F}(X) \bullet^{\uparrow} \mathcal{F}(X)) && \{\text{Flatten-2}\}\\
&= \mathcal{P}^2_X(c \bullet \mathcal{F}^2(X)) && \{\text{Opt-2}\}
\end{aligned}$$

Indexing. Transforming the expression $\{X\,!\,i \mid i \leftarrow I\}$ yields $(X \bullet I)\,!^{\uparrow}\,I$. This causes X to be distributed first over the length of the index vector I, before a lifted indexing selects individual elements from the distributed copies. Hence, the intuitive work of the order $O(\#I)$, suggested by the original expression, increases to $O(\#X \cdot \#I)$. But the intended functionality is already provided by the permutation primitive:

(Opt-3) *Indexing*: $(X \bullet I)\,!^{\uparrow}\,I = \text{permute}(X, I)$

Distributing constant expressions. The transformation rule Trafo-2 only allows constant values to be distributed. By extending this rule to constant expressions

(Opt-4) *Constant expressions*: $(\lambda\bar{x}.e) * xs = e \bullet xs, \quad \bar{x}$ not free in e

we avoid inefficiencies that result from transforming nested apply-to-each expressions, where a generator merely serves as a replicator. Consider the expression $\{\{c + x \mid y \leftarrow ys\} \mid x \leftarrow xs\}$ and its transformation $((c \bullet xs) \bullet^{\uparrow} (ys \bullet xs)) +^{(2\uparrow)} (xs \bullet^{\uparrow} (ys \bullet xs))$. The vectors $c \bullet xs$ and xs are first distributed over the same structure $ys \bullet xs$, and then the elements of the resulting nested vectors are added. Distributing the constant expression yields $((c \bullet xs) +^{\uparrow} xs) \bullet^{\uparrow} (ys \bullet xs)$, adding first involving less work and distributing later.

The notation $x \bullet xs = \text{dist}(x, \#xs)$ hides a further opportunity for optimizing lifted distributes. Consider the expression $xs \bullet^{\uparrow} (ys \bullet zs)$. Using the definition of $\bullet$, we transform it into $\text{dist}^{\uparrow}(xs, \#^{\uparrow}(\text{dist}(ys, \#zs)))$: ys is first distributed over the length of zs, and then the length of each copy is computed. Alternatively, we first compute the length of ys, and then distribute it over zs: $\text{dist}^{\uparrow}(xs, \text{dist}(\#ys, \#zs))$. This is summarized by the identity $f * \circ \text{dist} = \text{dist} \circ (f \times \text{id})$

Extract. Frequently, vectors are constructed merely to apply an operation in parallel on its elements, and deconstructed right after this by selection operations. The Quicksort algorithm is an example of such a situation. During the transformation of expressions of this nature, subexpressions of the form $(!i) * \circ f ** \circ \{\cdot\}^{\uparrow}$ arise. If we think of a tuple of vectors of equal length as a tuple of rows, i.e. a matrix, then this expression means that the matrix is first transposed, then a function is applied to all its elements, and finally the ith column is extracted. Instead, we could simply apply f to all elements of the matrix and extract the ith row. This observation is justified by calculation:

$$\begin{aligned}
(!i) * \circ f ** \circ \{\cdot\}^{\uparrow} &= f * \circ (!i) * \circ \{\cdot\}^{\uparrow} && \{\text{map promotion, Meta-4}\}\\
&= f * \circ (!i) * \circ \{\cdot\} * \circ \text{zip} && \{\text{Meta-1}\}\\
&= f * \circ \pi_i * \circ \text{zip} && \{\text{map promotion, Meta-3}\}\\
&= f * \circ \pi_i && \{\text{Meta-1, Meta-2}\}\\
&= f * \circ (!i) \circ \{\cdot\} && \{\text{Meta-3}\}\\
&= (!i) \circ f ** \circ \{\cdot\} && \{\text{Meta-4}\}
\end{aligned}$$

We have derived the following rule:

(Opt-5) *Extract*: $(!i) * \circ f ** \circ \{\cdot\}^{\uparrow} = (!i) \circ f ** \circ \{\cdot\}$

5 Discussion

Experiments. To support our claim that the transformation rules in Sect. 4 are indeed optimizations, we performed several experiments. We describe here briefly the results of these benchmarks, executed on a Sun4/50 and a Cray J932 vector processor. Since the complete compiler that uses the optimization rules is not yet finished, the programs were manually transformed into C plus calls to the CVL. CVL (C Vector Library [7]) implements those primitives of the target language that operate on vectors; it was originally designed and implemented for Nesl [5]. Our manual transformation was executed mechanically to ensure that all steps can be easily applied in the compiler.

The amount of time saved through the optimization induced by Opt-1 — when applied to Quicksort — was, as expected, logarithmic to the problem size, since it depends on the number of function calls. To sort a sequence of 10 000 numbers, it was two percent of the overall running time. The time savings through Opt-2 were linear to the problem size, about a factor 1.5. For Opt-3, no benchmarks were performed, because the work-complexity of the nonoptimized computation is higher, and the outcome is therefore clear. Depending on the work-complexity needed to compute the expression in Opt-4, the application of the rule also decreases the work complexity. We therefore executed for this rule only those benchmarks, for which the work complexity did not change: $\mathrm{dist}(\#ys, xs)$ vs. $\#^{\uparrow}(\mathrm{dist}(ys, xs))$ and $(a \bullet xs) +^{\uparrow} (b \bullet ys)$ vs. $(a + b) \bullet xs$. In both cases, the speedups obtained by the optimizations were about a factor of four. The benchmarks for Opt-5 yielded a speedup by a factor of 6, lifted concatenation as well as lifted extraction on vectors being far more time consuming than the plain versions.

Related work. Palmer et al. [15] proceed much along the same lines as we do. They introduce equations on vector expressions that are used to implement flattening. Although their transformations and our formalism share common features, there are important differences. Most important is our emphasis on a calculational approach that interprets apply-to-each constructs as map operations. As a result, our transformation rules follow from the algebraic properties stated in Sect. 2, instead of being introduced ad hoc. The optimizations achieved by Palmer et al. are partly the same as those in Sect. 4, but, given our calculational approach, we are able to simplify and generalize them. In particular, we are able to avoid introducing new primitives when dealing with the problems of distributing constant values and parallel indexing. Also, our formalism (rule Opt-4) picks up optimizations that the expression hoisting of Palmer et al. fails to consider (cf. the dist problem). Other optimizations like Flatten-2 and Opt-5 are, to the best of our knowledge, unique to our approach. Prins et al. [17] formally demonstrate that the flattening transformation is work-efficient. A similar statement for our formalism still remains to be verified, and we hope to be able to adapt their reasoning.

Conclusion. We presented a formal framework for expressing the flattening transformation using a set of transformation rules that translate nested data-parallel functional programs into flat data-parallel programs. Our aim was to obtain a precise understanding of the flattening process, and to express it in the form of a concise set of clear rules in an equational style, facilitating the development of new optimizations and the verification of the translation. While we failed to provide a completely formal verification of the rules, our experiments did provide evidence that the new rules result in a number of useful optimizations that are an improvement on previous implementations. The calculational style supported by the formalism is well-suited for compiler design and rigorous reasoning. There is still room for improvement of the calculus. The way we treat meta tuples — in a rather backstage manner — is unsatisfactory. Moreover, we have not yet exploited all the algebraic properties of our language's constructs. For instance, nesting generators corresponds to taking the tensor product of the data space, whereas parallel generators correspond to taking the Cartesian product. Here we plan to investigate whether we can suitably adopt the semantic notion of "shapes" [13]. There are a number of questions requiring further research in the area of flattening nested data parallelism. Current implementations of nested data-parallel languages are based on the CVL vector library [7]. During development of the new transformation rules and from experience gained in the implementation of V [9], it became clear that a different set of operations from those supported by the CVL would allow the generation of more efficient code. A new vector library that improves on these observed deficiencies should be carefully designed and implemented. Another research task would be to extend the presented rules to allow the translation of imperative languages like V. One line we plan to pursue here is the integration of imperative constructs by means of monads [14, 20]. This would allow the controlled introduction of imperative features such as mutable variables.

Acknowledgments. We wish to thank M. Chakravarty for discussing numerous points with us. Our thanks also go to Y. Guo, R. Joshi, M. Köhler, H. Lock, and W. Pfannenstiel for helpful comments on previous versions of this paper, to the members of the V project, and to P. Bacon for polishing up the English. The work of the first-named author is being supported by a scholarship from the German Research Foundation (DFG).

References

1. P. K. T. Au, M. M. T. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, G. Keller, M. Köhler, M. Simons, and W. Pfannenstiel. Enlarging the scope of vector-based computations: extending Fortran 90 with nested data parallelism. In W. Giloi, ed., *Intl. Conf. on Advances in Parallel and Distributed Computing*. IEEE Computer Society, 1997.
2. R. Backhouse. The calulational method. *Inf. Process. Lett.*, 53, 1995.
3. R. Bird. An introduction to the theory of lists. In M. Broy, ed., *Logic of Programming and Calculi of Discrete Design*, pp. 3–42. Springer, 1986.
4. G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.
5. G. E. Blelloch. Nesl: A nested data-parallel language. TR CMU-CS-95-170, CMU, 1995.
6. G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.
7. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library. TR CMU-CS-93-114, CMU, 1993.
8. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *J. Par. Distr. Comput.*, 21(1):4–14, 1994.
9. M. M. T. Chakravarty, F.-W. Schröer, and M. Simons. V—Nested parallelism in C. In Giloi et al. [10], pp. 167–174.
10. W. K. Giloi, S. Jähnichen, and B. D. Shriver, eds. *Programming Models for Massively Parallel Computers*. IEEE Computer Society, 1995.
11. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, ed., *Computer Science Today*, LNCS 1000, pp. 366–373. Springer, 1996.
12. HPF Forum. HPF language specification (Version 1.0). Tech. rep., Rice University, 1993.
13. C. B. Jay. A semantics for shape. *Sci. Comput. Programming*, 25:251–283, 1995.
14. S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In H. R. Nielson, ed., *Europ. Symp. on Programming (ESOP'96)*, LNCS 1058. Springer, 1996.
15. D. Palmer, J. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *5th Symp. on the Front. of Massively Parallel Processing*. IEEE Computer Society, 1995.
16. J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *4th ACM Symp. on Princ. and Pract. of Parall. Programming*, pp. 119–128, 1993.
17. J. W. Riely, J. Prins, and S. P. Iyer. Provably correct vectorization of nested-parallel programs. In Giloi et al. [10], pp. 213–222.
18. G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, 1988.
19. Thinking Machines Corporation. *C* Language Reference Manual*, 1991.
20. P. Wadler. Comprehending monads. *Math. Struct. in Comp. Sci.*, 2:461–493, 1992.