Flattening is an Improvement
(Extended Abstract)

James Riely[1] and Jan Prins[2]

[1] DePaul University
[2] University of North Carolina at Chapel Hill

Abstract. Flattening is a program transformation that eliminates nested parallel constructs, introducing flat parallel (vector) operations in their place. We define a sufficient syntactic condition for the correctness of flattening, providing a static approximation of Blelloch's "containment". This is acheived using a typing system that tracks the control flow of programs. Using a weak improvement preorder, we then show that the flattening transformations are intensionally correct for all well-typed programs.

1 Introduction

The study of program transformations has largely been concerned with functional correctness, *i.e.* whether program transformations preserve program meaning. However, if we include an execution cost-model as part of the programming language semantics, then we can ask whether program transformations additionally preserve or "improve" program performance. One program *improves* another if, for every binding of variables, it evaluates to the same answer in fewer steps. Sands has initiated a formal study of improvement for source-to-source transformation of sequential programs [30,29]. In this paper we study improvement for source-to-target transformation of parallel programs. Our source language is equipped with a natural parallel semantics, including a cost model, but lacks a direct parallel implementation. Our target is (almost) a subset of the source language that is directly implementable on parallel machines within the bounds of our cost model. We are interested in showing that a transformed program improves execution cost, *i.e.* that its performance is approximately the same as that prescribed for the source program. This gives our work a different flavor from that of Sands.

We study Blelloch and Sabot's *flattening* transformations [7], used to implement a nested data-parallel programming language in terms of a vector-based sublanguage. Nested parallelism allows the simple expression of parallel algorithms over irregular structures, such as nested lists. For examples, including many divide-and-conquer algorithms, see [3].

The flattening transformations remove instances of a second order parallel "map" functional, introducing vector operations in the process. We write parallel maps using the *iterator* construct. The syntax is similar to that normally used for list comprehensions [14], although the semantics is quite different. For example, the iterator

$$[x \Leftarrow xs, y \Leftarrow ys : \mathsf{plus}(x, \mathsf{mult}(y, 2))] \qquad (*)$$

specifies the evaluation of '$\mathsf{plus}(x, \mathsf{mult}(y, 2))$' for each binding of (x, y), drawn from $\mathsf{zip}(xs, ys)$. If xs is $\langle 1, 2\rangle$ and ys is $\langle 5, 7\rangle$, then $(*)$ evaluates to $\langle 11, 16\rangle$. The expression has a natural parallel interpretation.[1] The *step-count* of an iterator is the maximum of the step-counts of the subevaluations. The *work-count* of an iterator is the sum of the work-counts of the subevaluations. Thus ($*$) takes a constant number of steps and takes work proportional to the length of *xs* and *ys*.

Using the flattening transformations, ($*$) can be rewritten to:

$$\mathsf{let}\ \mathit{twos} \Leftarrow \mathsf{prom}(\mathit{xs}, 2)\ \mathsf{in}\ \mathsf{plus}^1(\mathit{xs}, \mathsf{mult}^1(\mathit{ys}, \mathit{twos})) \qquad (\dagger)$$

Here, prom is a primitive that "promotes" its second argument by copying it to match the length of its first argument; if *xs* is $\langle 1,2,3 \rangle$ then $\mathsf{prom}(\mathit{xs}, 2)$ evaluates to $\langle 2,2,2 \rangle$. plus^1 and mult^1 are respectively vector addition and vector multiplication. prom and the vector operators each execute in one parallel step.

Note that in translating from ($*$) to ($\dagger$), the nesting of parallel and sequential constructs has been inverted. ($*$) specifies the parallel execution of a sequential expression involving scalar addition and multiplication, whereas ($\dagger$) specifies the sequential execution of vector addition and vector multiplication. In both expressions the step-count is constant and the work-count is proportional to the length of *xs* and *ys*. In general, however, nesting inversion creates problems, particularly for conditional expressions.

We say that a transformation is *correct* if, for any program, applying the transformation results in a weak improvement. *Weak improvement* allows that program P may improve Q even if P is slower by a constant factor. Weak improvement is a permissive condition; nonetheless, the flattening transformations fail to satisfy it. Although flattening does not change the results computed by an expression, it may serialize certain parallel computations, increasing the step-count drastically. This lead Blelloch [1] to define a semantic condition, known as *containment*, that identifies iterator-based programs that are suitable for implementation using only parallel vector operations. Contrary to folklore, however, containment is not sufficient to guarantee that flattening results in weak improvement; we present a counterexample in Section 5.

In order to specify a subset of programs for which flattening *does* imply weak improvement, we introduce a typing system that divides expressions into three categories. Roughly described, these are: cnst, expressions that evaluate in a constant number of parallel steps; flat, a subset of contained expressions; and exp, all expressions. Using this typing system, we are able to prove that, for flat expressions, flattening is correct. We believe that ours is the first proof of the correctness of flattening.

The paper is organized as follows: We first introduce the programming language and its semantics and the flattening transformations. In Section 5, we show that the transformations do not imply weak improvement, even for contained programs. The

[1] The cost of a parallel program is typically described using two metrics, *steps*, which are computed assuming that all available parallelism is realized, and *work*, which is computed assuming that no available parallelism is realized. Terms in our target language can be mapped to the Vector Random Acccess Machine (VRAM) [1] in a straightforward way that preserves both steps and work. The VRAM, in turn, can be related to other models of computation [8]. An expression in our target language that has step-count t and work-count w can be executed on a p-processor PRAM in $O(w/p + t \log p)$ time [1]. When $w \gg p \log p$, the PRAM running time is a good estimate of actual running times on uniform-access shared-memory machines with high-bandwidth memory systems, such as vector machines or the Tera MTA [4,23].

Table 1 Source, Intermediate and Target Expressions

A,B,C,D,E ::=	*Expressions*	*Sublanguage*
a	Value	S/I/T
x	Variable	S/I/T
p	Primitive	S/I/T
$B(A_1, .., A_\ell)$	Application	S/I/T
$\mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C$	Conditional	S/I/T
$\mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A$	Sequencing	S/I/T
$\mathsf{letrec}\ f \Leftarrow (x_1, .., x_\ell)\ D,E\ \mathsf{in}\ A$	Function definition	S/I/T
$[x_1 \Leftarrow B_1, .., x_\ell \Leftarrow B_\ell : A]$	Iterator	S/I
$\langle x_1 \Leftarrow xs_1, .., x_\ell \Leftarrow xs_\ell : A\rangle$	Evaluated iterator	I
$B^1(A_1, .., A_\ell)$	Parallel application	I/T

typing system is defined in Section 6. In the following section we sketch the correctness proof. The details are omitted for lack of space; interested readers are referred to [27]. We conclude with a discussion of related work.

2 A Nested-Sequence Language

Source Language. The language is strict, functional, and first-order. The datatypes include sequences and integer and boolean scalars. We use two notations for sequence values, angle brackets and overlines; thus, $\langle 1,2,3\rangle$ and $\overline{123}$ both represent the three-element sequence whose i^{th} element is the integer i. The empty sequence is written $\langle\rangle$ or $\bullet$.

The basic constructors for sequences are a family of primitives build_ℓ that build an ℓ-element list from ℓ arguments; for example, $\mathsf{build}_2(1,2) = \langle 1,2\rangle$. The basic destructor is the elt primitive, which selects an element from a sequence; for example, $\mathsf{elt}\big(2, \langle 5,6,7\rangle\big) = 6$. Other important primitives include rstr, which restricts a sequence based on a sequence of booleans, merge, which merges two sequences based on a sequence of booleans, flat, which "flattens" a nested sequence, and part, which partitions a sequence according to the structure of a different sequence. Let t and f be the boolean values true and false respectively, and let a through e be arbitrary values, then:

$$\mathsf{rstr}\left(\overline{\mathsf{tft}}, \overline{123}\right) = \overline{13} \qquad \mathsf{flat}\ \overline{\overline{12}\ \overline{345}} = \overline{12345}$$

$$\mathsf{merge}\left(\overline{123}, \overline{\mathsf{ftfft}}, \overline{89}\right) = \overline{18239} \qquad \mathsf{part}\left(\overline{\overline{ab}\ \overline{cde}}, \overline{12345}\right) = \overline{\overline{12}\ \overline{345}}$$

The primitives satisfy the following equations. Let i be a natural number between 1 and ℓ. Let as be a sequence and let bs be a boolean sequence of equal length, with $\widehat{bs}$ its elementwise logical complement. Let ass be a sequence of sequences.

$$\mathsf{elt}\big(i, \mathsf{build}_\ell(a_1, .., a_\ell)\big) = a_i$$
$$\mathsf{merge}\big(\mathsf{rstr}(\widehat{bs}, as), bs, \mathsf{rstr}(bs, as)\big) = as$$
$$\mathsf{part}\big(ass, \mathsf{flat}\ ass\big) = ass$$

The syntax is parameterized with respect to sets *Prim*, of *primitive names*, ranged over by p, and *Var*, of *variable names*, ranged over by f, x, y, z, xs, xss, etc. Let h through n range over integers, bv over booleans, and a, b, as, bs, etc. over arbitrary values.

The syntax of *expressions*, or *terms*, A, B, etc. is given in Table 1. A term is *source* term if it contains no evaluated iterators or parallel applications. A term is *target* term if it contains no iterators or evaluated iterators. We sometimes refer to arbitrary terms as *intermediate* terms.

Parallelism. Most of the constructs of the language are sequential; thus step-count and work-count are computed the same way. For example, the step-count of '$\mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A$' is the sum of the step-counts for the subexpressions A and B; the work-count is the sum of the work-counts for A and B.

Parallelism is expressed in the source language using reduction primitives and the iterator construct. For example, the key step in the parallel *quicksort* of a sequence *xs* (with no duplicate values) can be written, with some syntactic sugar, as follows [3]:

$$\begin{aligned}
&\mathsf{let}\ \mathit{les} = [x \Leftarrow \mathit{xs} \mid x < \mathsf{elt}(1,\mathit{xs}) \colon x] \\
&\qquad \mathit{gtr} = [x \Leftarrow \mathit{xs} \mid x \geq \mathsf{elt}(1,\mathit{xs}) \colon x] \\
&\mathsf{in}\ \mathsf{flat}\ [\mathit{ys} \Leftarrow \mathsf{build}_2(\mathit{les},\mathit{gtr}) \colon \mathit{quicksort}(\mathit{ys})]
\end{aligned}$$

If n is the length of *xs*, then the expected step-count is $O(\log n)$ and the expected work is $O(n \log n)$. Like all other primitives, reductions are assigned a constant number of steps. Thus '$\mathsf{sum}\ \langle 1,2,3,4,5\rangle$' evaluates to 15 with step-count 1. We formalize the notions of step and work complexity in Section 4.

Execution of the nested data-parallelism expressed in this simple algorithm is quite challenging, as the subproblems created by recursive invocations vary in size, and the quicksort call tree varies in depth. The correctness of the flattening transformations established in this paper guarantee that the flattened quicksort combines all these separate pieces of work in the form of an expected $O(\log n)$ vector operations of size $O(n)$.

Intermediate Constructs. The flattening transformations eliminate iterators. To simplify the expression of the transformation rules, we introduce an intermediate construct, called the *evaluated iterator* or *e-iterator*. Semantically, e-iterators are similar to iterators.

In the target language, parallelism is expressed using parallel implementations of the primitives. Thus '$[x \Leftarrow \mathit{xs} \colon \mathsf{plus}(x,x)]$' in the source language becomes '$\mathsf{plus}^1(\mathit{xs},\mathit{xs})$' in the target. We require that each primitive p have a parallel implementation p^1. The target language also allows for parallel application of user-defined functions. Thus '$[x \Leftarrow \mathit{xs} \colon f\,x]$' in the source language becomes '$f^1\,\mathit{xs}$' in the target. Here, however, the body of f^1 must be provided explicitly. In the expression '$\mathsf{letrec}\ f \Leftarrow \widetilde{x} D{,}E\ \mathsf{in}\ A$', the expressions D and E give definitions for f. Essentially D gives the sequential implementation of f, whereas E gives the parallel implementation of f^1. In practice, only the sequential definition need be provided by a programmer, the parallel definition can be derived automatically, as $E \overset{\mathit{def}}{=} \langle \widetilde{y} \Leftarrow \widetilde{x} \colon D\{\!|\widetilde{y}/\widetilde{x}|\!\}\rangle$. In examples, we usually write function declarations simply as '$\mathsf{letrec}\ f \Leftarrow \widetilde{x} D\ \mathsf{in}\ A$' or equivalently '$\mathsf{letrec}\ f \widetilde{x} \Leftarrow D\ \mathsf{in}\ A$'.

Notation. The notation for iterators is sometimes cumbersome. We often write '$[x_1 \Leftarrow B_1, .., x_\ell \Leftarrow B_\ell \colon A]$' as '$\left[\widetilde{x} \Leftarrow \widetilde{B} \colon A\right]$'. In examples, we also use a notation for *filters*, which can be coded using the rstr primitive. For example, '$[x \Leftarrow \langle 1,2,3,4,5,6\rangle \mid \mathsf{odd}\,x \colon \mathsf{square}\,x]$' evaluates to the sequence $\langle 1,9,25\rangle$. Here, '$\mathsf{odd}\,x$' is an expression that filters the values over which the iterator is applied.

Table 2 Transformations: Context and Let Rules

(X-CTXT$_A$)	$B\widetilde{A}$	$\leadsto$	$B'\widetilde{A}$	if $B \leadsto B'$
(X-CTXT$_{L1}$)	$\text{let } x \Leftarrow B \text{ in } A$	$\leadsto$	$\text{let } x \Leftarrow B' \text{ in } A$	if $B \leadsto B'$
(X-CTXT$_{L2}$)	$\text{let } x \Leftarrow B \text{ in } A$	$\leadsto$	$\text{let } x \Leftarrow B \text{ in } A'$	if $A \leadsto A'$
(X-CTXT$_{C1}$)	$\text{if } B \text{ then } A \text{ else } C$	$\leadsto$	$\text{if } B \text{ then } A' \text{ else } C$	if $A \leadsto A'$
(X-CTXT$_{C2}$)	$\text{if } B \text{ then } A \text{ else } C$	$\leadsto$	$\text{if } B \text{ then } A \text{ else } C'$	if $C \leadsto C'$
(X-CTXT$_{R1}$)	$\text{letrec } f \Leftarrow \widetilde{x} D, E \text{ in } A$	$\leadsto$	$\text{letrec } f \Leftarrow \widetilde{x} D', E \text{ in } A$	if $D \leadsto D'$
(X-CTXT$_{R2}$)	$\text{letrec } f \Leftarrow \widetilde{x} D, E \text{ in } A$	$\leadsto$	$\text{letrec } f \Leftarrow \widetilde{x} D, E' \text{ in } A$	if $E \leadsto E'$
(X-CTXT$_{R3}$)	$\text{letrec } f \Leftarrow \widetilde{x} D, E \text{ in } A$	$\leadsto$	$\text{letrec } f \Leftarrow \widetilde{x} D, E \text{ in } A'$	if $A \leadsto A'$
(X-ELET)	$\text{let } x \Leftarrow y \text{ in } A$	$\leadsto$	$A\{\!\!\{ y/x \}\!\!\}$	
(X-ILET$_A$)	$B\ (A_1,..,A_i,..,A_\ell)$	$\leadsto$	$\text{let } x \Leftarrow A_i \text{ in } B\ (A_1,..,x,..,A_\ell)$	if $A_i \notin \mathit{Var}$
(X-ILET$_P$)	$B^1(A_1,..,A_i,..,A_\ell)$	$\leadsto$	$\text{let } x \Leftarrow A_i \text{ in } B^1(A_1,..,x,..,A_\ell)$	if $A_i \notin \mathit{Var}$
(X-ILET$_C$)	$\text{if } B \text{ then } A \text{ else } C$	$\leadsto$	$\text{let } x \Leftarrow B \text{ in if } x \text{ then } A \text{ else } C$	if $B \notin \mathit{Var}$
(X-ILET$_I$)	$[x_1 \Leftarrow B_1,..,x_i \Leftarrow B_i,..,x_\ell \Leftarrow B_\ell : A]$			if $B_i \notin \mathit{Var}$
		$\leadsto$	$\text{let } xs_i \Leftarrow B_i \text{ in } [x_1 \Leftarrow B_1,..,x_i \Leftarrow xs_i,..,x_\ell \Leftarrow B_\ell : A]$	

The variable x is bound in '$\text{let } x \Leftarrow B \text{ in } A$', the scope is A. The variable f is bound in the definition '$\text{letrec } f \Leftarrow \widetilde{x} D,E \text{ in } A$', the scope is D, E and A; the variables x_i are also bound in the definition '$\text{letrec } f \Leftarrow \widetilde{x} D,E \text{ in } A$', the scope is D and E. The variables x_i are bound in the iterator '$[\widetilde{x} \Leftarrow \widetilde{B} : A]$', the scope is A. The variables x_i are bound in the e-iterator '$\langle \widetilde{x} \Leftarrow \widetilde{xs} : A \rangle$', the scope is A. Let $fv(A)$ be the set of free variables occuring in A. We identify expressions up to renaming of bound variables. In every binding construct, the variables x_i must be unique. In every e-iterator $\langle \widetilde{x} \Leftarrow \widetilde{xs} : A \rangle$, A must be a source term.

3 The Transformations

Flattening was introduced in [7] and is an important implementation strategy for NESL [6] and Proteus [24,20]. Blelloch and Sabot described flattening as a set of transformations. A typical rule is the following rule for let-expressions. Given that variable zs does not occur free in A, '$[x \Leftarrow xs : \text{let } z \Leftarrow B \text{ in } A]$' rewrites to:

$$\text{let } zs \Leftarrow [x \Leftarrow xs : B] \text{ in } [x \Leftarrow xs, z \Leftarrow zs : A]$$

As the example implies, the basic strategy is to "push" the iterator expressions through the abstract syntax until it can be replaced, either by a variable or a promoted constant. The elimination rules allow '$[x \Leftarrow xs : x]$' to be rewritten simply as 'xs' and '$[x \Leftarrow xs : A]$' to be rewritten as '$\text{prom}(xs, A)$' as long as x does not appear free in A. The transformation of conditionals specifies that if z does not appear free in A or C, then '$[z \Leftarrow zs, x \Leftarrow xs : \text{if } z \text{ then } A \text{ else } C]$' rewrites to:

$$\text{merge}\big([x \Leftarrow \text{rstr}(zs, xs) : A],\ \text{not}^1\, zs,\ [x \Leftarrow \text{rstr}(\text{not}^1\, zs, xs) : C]\big)$$

We formalize the flattening transformations as a relation $A \leadsto A'$ on expressions. The relation is defined in two tables. The context rules and the transformations for let introduction and elimination are given in Table 2. The main rules are in Table 3. In all of the rules, variables introduced on the right-hand-side of the transformation must be

Table 3 Transformations: Iterator Rules

(X-IIT)	$[\widetilde{x} \Leftarrow \widetilde{xs} : A]$ $\boxed{\begin{array}{l} \mathit{fv}(A) \setminus \widetilde{x} = \{y_1, .., y_\ell\} \\ A \text{ is a source term} \end{array}}$	$\rightsquigarrow$	$\begin{array}{l} \mathsf{if\ empty}\ xs_h\ \mathsf{then}\ \langle\rangle \\ \mathsf{else\ let}\ ys_1 \Leftarrow \mathsf{prom}(xs_h, y_1) \\ \qquad \vdots \\ \quad \mathsf{let}\ ys_\ell \Leftarrow \mathsf{prom}(xs_h, y_\ell) \\ \quad \mathsf{in}\ \ \langle \widetilde{x} \Leftarrow \widetilde{xs}, \widetilde{y} \Leftarrow \widetilde{ys} : A \rangle \end{array}$	
(X-EIT)	$\langle \widetilde{y} \Leftarrow \widetilde{ys}, x \Leftarrow xs, \widetilde{z} \Leftarrow \widetilde{zs} : A \rangle$	$\rightsquigarrow$	$\langle \widetilde{y} \Leftarrow \widetilde{ys}, \widetilde{z} \Leftarrow \widetilde{zs} : A \rangle$	$\boxed{x \notin \mathit{fv}(A)}$
(X-CONST)	$\langle x \Leftarrow xs : A \rangle$	$\rightsquigarrow$	$\mathsf{prom}(xs, A)$	$\boxed{x \notin \mathit{fv}(A)}$
(X-VAR)	$\langle x \Leftarrow xs : x \rangle$	$\rightsquigarrow$	xs	
(X-APP)	$\langle \widetilde{x} \Leftarrow \widetilde{xs} : B(x_{i_1}, .., x_{i_\ell}) \rangle$	$\rightsquigarrow$	$B^1(xs_{i_1}, .., xs_{i_\ell})$	
(X-LETREC)	$\langle \widetilde{x} \Leftarrow \widetilde{xs} : \mathsf{letrec}\ f \Leftarrow \widetilde{y} D, E\ \mathsf{in}\ A \rangle$	$\rightsquigarrow$	$\mathsf{letrec}\ f \Leftarrow \widetilde{y} D, E\ \mathsf{in}\ \langle \widetilde{x} \Leftarrow \widetilde{xs} : A \rangle$	
(X-LET)	$\langle \widetilde{x} \Leftarrow \widetilde{xs} : \mathsf{let}\ z \Leftarrow B\ \mathsf{in}\ A \rangle$	$\rightsquigarrow$	$\mathsf{let}\ zs \Leftarrow \langle \widetilde{x} \Leftarrow \widetilde{xs} : B \rangle\ \mathsf{in}\ \langle \widetilde{x} \Leftarrow \widetilde{xs}, z \Leftarrow zs : A \rangle$	
(X-IF)	$\langle \widetilde{x} \Leftarrow \widetilde{xs} : \mathsf{if}\ x_h\ \mathsf{then}\ A\ \mathsf{else}\ C \rangle$ $\boxed{\begin{array}{l} \mathit{fv}(A) = \{x_{i_1}, .., x_{i_\ell}\} \neq \emptyset \\ \mathit{fv}(C) = \{x_{j_1}, .., x_{j_k}\} \neq \emptyset \end{array}}$	$\rightsquigarrow$	$\begin{array}{l} \mathsf{if\ all}\ xs_h\ \mathsf{then}\ \langle \widetilde{x} \Leftarrow \widetilde{xs} : A \rangle \\ \mathsf{else\ if\ not\ some}\ xs_h\ \mathsf{then}\ \langle \widetilde{x} \Leftarrow \widetilde{xs} : C \rangle \\ \mathsf{else\ let}\ ys_{i_1} \Leftarrow \mathsf{rstr}(xs_h, xs_{i_1}) \\ \qquad \vdots \\ \quad \mathsf{let}\ ys_{i_\ell} \Leftarrow \mathsf{rstr}(xs_h, ys_{i_\ell}) \\ \quad \mathsf{let}\ zs_{j_1} \Leftarrow \mathsf{rstr}(\mathsf{not}^1 xs_h, xs_{j_1}) \\ \qquad \vdots \\ \quad \mathsf{let}\ zs_{j_k} \Leftarrow \mathsf{rstr}(\mathsf{not}^1 xs_h, xs_{j_k}) \\ \quad \mathsf{in}\ \ \mathsf{merge}(\langle \widetilde{y} \Leftarrow \widetilde{ys} : A \rangle, \mathsf{not}^1 xs_h, \langle \widetilde{z} \Leftarrow \widetilde{zs} : C \rangle) \end{array}$	
(X-IT)	$\langle \widetilde{x} \Leftarrow \widetilde{xs} : [y_1 \Leftarrow x_{i_1}, .., y_{m'} \Leftarrow x_{i_{m'}} : A] \rangle$ $\boxed{\begin{array}{l} \mathit{fv}(A) = \{x_{k_1}, .., x_{k_p}\} \\ \qquad \cup \{y_{k'_1}, .., y_{k'_q}\} \\ \qquad \neq \emptyset \end{array}}$	$\rightsquigarrow$	$\begin{array}{l} \mathsf{if\ all\ empty}^1 xs_h\ \mathsf{then}\ \mathsf{prom}(xs_h, \langle\rangle)) \\ \mathsf{else\ let}\ xs'_{k_1} \Leftarrow \mathsf{flat}(\mathsf{prom}^1(xs_{i_{h'}}, xs_{k_1})) \\ \qquad \vdots \\ \quad \mathsf{let}\ xs'_{k_p} \Leftarrow \mathsf{flat}(\mathsf{prom}^1(xs_{i_{h'}}, xs_{k_p})) \\ \quad \mathsf{let}\ ys'_{k'_1} \Leftarrow \mathsf{flat}(xs_{i_{k'_1}}) \\ \qquad \vdots \\ \quad \mathsf{let}\ ys'_{k'_q} \Leftarrow \mathsf{flat}(xs_{i_{k'_q}}) \\ \quad \mathsf{in}\ \ \mathsf{part}(xs_{i_{h'}}, \langle \widetilde{x} \Leftarrow \widetilde{xs}', \widetilde{y} \Leftarrow \widetilde{ys}' : A \rangle) \end{array}$	

fresh, that is, they may not appear free in any subexpression given anywhere in the rule. We write $\overset{\star}{\rightsquigarrow}$ for the reflexive and transitive closure of $\rightsquigarrow$.

The general transformation strategy is as follows. The context and let introduction rules are used to isolate an iterator expression. Once an iterator expression is found, the let introduction rule (X-ILET$_\mathrm{I}$) is applied until the iteration space of the iterator is described entirely by variables. Note that '$\widetilde{x} \Leftarrow \widetilde{xs}$' is shorthand for '$x_1 \Leftarrow xs_1$, .., $x_h \Leftarrow xs_h$, .., $x_m \Leftarrow xs_m$'; thus, on the right-hand side of the rule, h can be bound to any integer between 1 and m.

At this point (X-IIT) is used to remove the iterator construct, replacing it with an e-iterator. The remaining rules of Table 3 are then used to "push" the e-iterator through the syntax until it can be removed using (X-CONST), (X-VAR) or (X-APP). The rules (X-ELET)and (X-EIT) allow for the elimination of useless let and e-iterator binders.

The rule (X-IIT) enforces two properties of e-iterators. First, it guarantees that e-iterators are only invoked dynamically on non-empty sequences. Second, it guarantees that e-iterators have no free variables. All free variables in an iterator are explicitly bound before the iterator is replaced with an e-iterator. The rules for conditionals and iterators are designed to preserve these properties. The transformation rules $(\textsc{x-if})$ and $(\textsc{x-it})$ require that A and C contain at least one free variable. Variants of these rules must be used in the case that $fv(A)$ or $fv(C)$ are empty; the variants are straightforward and have been elided. The soundness of $(\textsc{x-letrec})$ is ensured by the typing rules, presented in Section 6.

4 A Reference-Based Semantics

We present the semantics of the intermediate language, and thus also the source and target languages. The semantics gives a formal defintion of the steps and work used in the evaluation of an expression. We sketch a reference-based implementation of the target language that meets the constraints imposed by the semantics and discuss other alternatives.

The semantics of expressions is defined in Table 4 using judgments of the form '$\sigma \vdash A \xrightarrow[w]{t} a$', which is read, "given environment σ, expression A evaluates to a with t steps and w work." We occasionally drop the annotations t and w when they are not of interest. Here σ is a runtime environment which maps variables to values and function definitions; formally,

$$\sigma ::= \emptyset \mid f \Leftarrow \widetilde{x} D{,}E \mid x \Leftarrow a \mid \sigma_1, \sigma_2$$

where σ_1 and σ_2 have disjoint domains. Intuitively, the evaluation of an expression is an operation on a computer store. Given a store σ, the evaluation '$\sigma \vdash A \xrightarrow[w]{t} a$' models the execution of A to produce a value a stored in memory. In particular, I/O costs are not taken into account. This leads us to the axiom '$\sigma \vdash x \xrightarrow[0]{0} \sigma(x)$', which states that variable x can be evaluated with no computation whatsoever; the value of x is already in σ and therefore need not be computed.

The evaluation of a value takes time proportional to the cost of copying the value into the store. Copying a value a takes steps proportional to its *depth* $(\mathcal{D}\ a)$ and work proportional to its *size* $(\mathcal{S}\ a)$. For example, the depth of $\langle\langle 1\rangle, \langle 2,3,4\rangle, \langle 5,6\rangle\rangle$, is 2; its size is 10.

Explicit sequencing, via the let construct, incurs no cost. This ensures the validity of the let-introduction rules given in Table 2. For example, the semantics validates the equation '$f\,A \;=\; \mathsf{let}\ x \Leftarrow A\ \mathsf{in}\ f\,x$'. In order to compute $f\,A$, one must first compute A. In $\mathsf{let}\ x \Leftarrow A\ \mathsf{in}\ f\,x$ the sequence of events is simply made explicit, it is not changed.

Function declaration also incurs no cost. This interpretation is justified by the typing rules given in the next chapter. Roughly, functions must be fully parameterized; therefore, function declarations can be processed statically, with no runtime cost.

The rules $(\textsc{e-it})$ and $(\textsc{e-eit})$ formalize the interpretation of iterators outlined in the Introduction. In $[\widetilde{x} \Leftarrow \widetilde{B} \colon A]$, the expressions B_i are evaluated sequentially to produce sequences $\langle b_{ji}\rangle_{j=1}^{n}$, then A is evaluated in parallel for each of the n bindings of b_{ji} to x_i. The work of an iterator includes a constant charge for each parallel subevaluation; this ensures, *e.g.*, that $[x \Leftarrow xs \colon y]$ has work proportional to the length of xs.

The rule $(\textsc{e-app}_{\textsc{p}})$ appeals to an evaluation relation for primitives. The judgment '$p\,\widetilde{a} \xrightarrow[w]{t} d$' states that given parameters a_i, p evaluates to d with t steps and w work. We elide the definition for lack of space; a few examples are given at the end of this section.

Table 4 The Evaluation Relation

(E-VAR)
$$\frac{}{\sigma \vdash x \xrightarrow[0]{0} \sigma(x)}$$

(E-VAL)
$$\frac{}{\sigma \vdash a \xrightarrow[\mathcal{S}\,a]{\mathcal{D}\,a} a}$$

(E-LET)
$$\frac{\sigma \vdash B \xrightarrow[w_B]{t_B} b \qquad \sigma, x \Leftarrow b \vdash A \xrightarrow[w_A]{t_A} a}{\sigma \vdash \mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A \xrightarrow[w_B+w_A]{t_B+t_A} a}$$

(E-LETREC)
$$\frac{\sigma, f \Leftarrow \widetilde{x}D,E \vdash A \xrightarrow[w_A]{t_A} a}{\sigma \vdash \mathsf{letrec}\ f \Leftarrow \widetilde{x}D,E\ \mathsf{in}\ A \xrightarrow[w_A]{t_A} a}$$

(E-IF$_\mathrm{T}$)
$$\frac{\sigma \vdash B \xrightarrow[w_B]{t_B} \mathsf{t} \qquad \sigma \vdash A \xrightarrow[w_A]{t_A} a}{\sigma \vdash \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C \xrightarrow[1+w_B+w_A]{1+t_B+t_A} a}$$

(E-IF$_\mathrm{F}$)
$$\frac{\sigma \vdash B \xrightarrow[w_B]{t_B} \mathsf{f} \qquad \sigma \vdash C \xrightarrow[w_C]{t_C} c}{\sigma \vdash \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C \xrightarrow[1+w_B+w_C]{1+t_B+t_C} c}$$

(E-IT)
$$\frac{\{\ \sigma \vdash B_i \xrightarrow[w_i]{t_i} \langle b_{ji}\rangle_{j=1}^{n}\ \}_{i=1}^{\ell} \qquad \{\ \sigma, \widetilde{x} \Leftarrow \widetilde{b_j} \vdash A \xrightarrow[w_j]{t_j} a_j\ \}_{j=1}^{n}}{\sigma \vdash \left[\widetilde{x} \Leftarrow \widetilde{B} \colon A\right] \xrightarrow[1+n+(\Sigma w_i)+(\Sigma w_j)]{1+(\Sigma t_i)+(\max t_j)} \langle a\rangle_{j=1}^{n}}$$

(E-EIT)
$$\frac{\{\ \sigma \vdash xs_i \xrightarrow[0]{0} \langle b_{ji}\rangle_{j=1}^{n}\ \}_{i=1}^{\ell} \qquad \{\ \sigma, \widetilde{x} \Leftarrow \widetilde{b_j} \vdash A \xrightarrow[w_j]{t_j} a_j\ \}_{j=1}^{n}}{\sigma \vdash \langle \widetilde{x} \Leftarrow \widetilde{xs} \colon A\rangle \xrightarrow[n+\Sigma w_j]{1+\max t_j} \langle a_j\rangle_{j=1}^{n}}\ n \geq 1$$

(E-APP$_\mathrm{P}$)
$$\frac{\{\ \sigma \vdash A_i \xrightarrow[w_i]{t_i} a_i\ \}_{i=1}^{\ell} \qquad p\,\widetilde{a} \xrightarrow[w_p]{t_p} d}{\sigma \vdash p\widetilde{A} \xrightarrow[1+(\Sigma w_i+1)+w_p]{1+(\Sigma t_i+1)+t_p} d}$$

(E-PAPP$_\mathrm{P}$)
$$\frac{\{\ \sigma \vdash A_i \xrightarrow[w_i]{t_i} \langle a_{ji}\rangle_{j=1}^{n}\ \}_{i=1}^{\ell} \qquad \{\ p\,\widetilde{a}_j \xrightarrow[w_j]{t_p} d_j\ \}_{j=1}^{n}}{\sigma \vdash p^1\widetilde{A} \xrightarrow[1+(\Sigma w_i+1)+\Sigma w_j]{1+(\Sigma t_i+1)+t_p} \langle d_j\rangle_{j=1}^{n}}\ n \geq 1$$

(E-APP$_\mathrm{F}$)
$$\frac{\{\ \sigma \vdash A_i \xrightarrow[w_i]{t_i} a_i\ \}_{i=1}^{\ell} \qquad \sigma, \widetilde{x} \Leftarrow \widetilde{a} \vdash D \xrightarrow[w_D]{t_D} d}{\sigma \vdash f\widetilde{A} \xrightarrow[1+(\Sigma w_i+1)+w_D]{1+(\Sigma t_i+1)+t_D} d}\ \sigma(f) = \widetilde{x}D,E$$

(E-PAPP$_\mathrm{F}$)
$$\frac{\{\ \sigma \vdash A_i \xrightarrow[w_i]{t_i} a_i\ \}_{i=1}^{\ell} \qquad \sigma, \widetilde{x} \Leftarrow \widetilde{a} \vdash E \xrightarrow[w_E]{t_E} e}{\sigma \vdash f^1\widetilde{A} \xrightarrow[1+(\Sigma w_i+1)+w_E]{1+(\Sigma t_i+1)+t_E} e}\ \sigma(f) = \widetilde{x}D,E$$

In both primitive and function application, charges are assessed for storing the return value, as well as for each parameter passed. Note that (E-APP$_\mathrm{F}$) and (E-PAPP$_\mathrm{F}$) differ only in which definition, D or E, is executed.

In Section 7 we prove that the typed version of the source language can be implemented in terms of the target language and that the translation respects the step and work complexities of the source semantics. There remains the question of whether the target language can be implemented on any actual machine. We treat this issue informally, by sketching an implementation of the target language on the VRAM [1].

In implementing the target language on the VRAM, one is confronted with two main difficulties: representing nested sequences in terms of vectors, and implementing the primitives. Implementations of the other constructs of the target language — function definition and application, let expressions and conditionals — are simple and direct.

The representation for sequences is crucial, as this sets a lower bound on the steps and work required to implement the primitives; this, in turn, affects the implementability of the source language. Blelloch and Sabot introduced the *segment-vector* encoding of sequences [7]. In this encoding, a depth-d sequence is represented as a tuple of d vectors: one to describe the data and $d-1$ to describe the nesting structure that contains it. Using segment vectors, the $\mathsf{prom}(as, b)$ primitive must create n copies of b, where

n is the length of as; this operation requires a minimum work of $n \cdot \mathbb{S}\, b$. Unfortunately, this means that our transformation rule for constant expressions is invalid. Consider the iterator '$[x \Leftarrow xs : y]$', which (X-CONST) translates to '$\mathsf{prom}(xs, y)$'. Suppose the length of xs is n. Looking at the source term, '$[x \Leftarrow xs : y]$' takes work proportional n. However, '$\mathsf{prom}(xs, y)$' takes work proportional to n times the size of y. If y refers to a non-scalar value, its size may easily dominate n. More important, the size of y depends on the environment; thus we cannot bound the work of the target expression with respect to the work of the source expression, not even asymptotically.

A solution adopted by Blelloch [2, appendix], is to change the costing of iterators to include the size of free variables. This change creates an unintuitive cost model for programmers that discourages the use of iterators. Here, we adopt a different strategy, representing nested sequences as vectors of references. Using this representation, $\mathsf{prom}(xs, y)$ takes work proportional n, creating n references to y.

This representation allows us to prove the transformations correct with respect to the natural high-level metric. However, it also leads to a greater number of concurrent reads, when compared to the segment vector representation, and hence greater memory contention at runtime. We believe that a reference-based implementation can perform well using techniques from [21], but we have no experimental results as of yet.

In [27], we present a semantics which captures the work/step model used in the implementation of NESL [6]. By adapting the techniques presented here, [27] provides the first proof of the correctness of flattening for NESL.

5 Improvement

To demonstrate the extensional correctness of the transformations, one can show:

$$\text{if } D \overset{\star}{\rightsquigarrow} D' \text{ then } \sigma \vdash D \longrightarrow d \text{ iff } \sigma \vdash D' \longrightarrow d$$

This states that transformation preserves the extensional meaning of programs.We wish to show something stronger, however. Our goal is to show that the transformations preserve computational cost, in some sense, not just extensional meaning. We wish to show that $D \overset{\star}{\rightsquigarrow} D'$ implies $D \succ D'$, for some relation $\succ$ that captures the intuition that if D reduces to a value, then D' reduces to the same value and does so as fast or faster. As a first attempt, we might say that $D \succ E$ if for all σ,

$$\sigma \vdash D \xrightarrow[w]{t} d \text{ implies } \sigma \vdash E \xrightarrow[\leqslant w]{\leqslant t} d$$

where "$\sigma \vdash D \xrightarrow[\leqslant w]{\leqslant t} d$" abbreviates "there exists $t' \leq t$ and $w' \leq w$ such that $\sigma \vdash D \xrightarrow[w']{t'} d$." This relation is known as *strong improvement*; however, this relation is too strong to be useful directly. The transformations do not imply strong improvement, as one can easily see by looking at, *e.g.*, (X-CONST), (X-IF_2) or (X-IT_2).

While we cannot prove that flattening strictly improves performance with respect to our operational semantics, we can prove that it does so *up to a constant factor* (in some cases). Formally, we will define $D \succ E$ if there exist constants u and v such that for all σ:

$$\sigma \vdash D \xrightarrow[w]{t} d \text{ implies } \sigma \vdash E \xrightarrow[\leqslant v \cdot w]{\leqslant u \cdot t} d$$

This relation is called *weak improvement*.

Unfortunately, there are programs in our language for which flattening does not imply even weak improvement. Suppose that $f(x)$ is defined as follows:

$$f(x) \Leftarrow \text{ if } x \leq 1 \text{ then } 1 \text{ else } (\text{if even} x \text{ then } f(x/2) \text{ else } f(x/2))$$

Then $f(2^n)$ evaluates to 1 in $O(n)$ steps. If xs is the sequence $\langle 2^n, 2^n+1, .., 2^{(n+1)}\rangle$ of 2^n values, then $[x \Leftarrow xs \colon f\,x]$ also evaluates in $O(n)$ steps. The transformations sequentialize the branches of the conditional so that the two recursive calls to f are performed one after the other. The result is that after the transformations, $f^1(xs)$ takes $O\left(n^2\right)$ steps, destroying any hope that the transformation of f might result in even a weak improvement.

As we stated in the introduction, it has long been known that flattening is not correct for all expressions, leading Blelloch to define containment [1]. Roughly stated, a recursive function such as f is *contained* if it always evaluates in the same way, calling the same functions and primitives in the same order, regardless of its actual parameters. According to Blelloch's definition, f is contained, although it is not correctly flattened by the standard transformations. This apparent anomaly can be explained by looking more closely at Blelloch's results. His *containment theorem* does not address flattening, but rather uses an entirely different simulation technique which appeals to the semantics, rather than the syntax, of expressions.

One of the main contributions of this work is to move containment from a semantic criterion to a syntactic one, thus allowing us to precisely characterize a set of programs for which flattening is correct. This is achieved using a typing system, presented next.

6 A Typing System for Containment

We introduce a typing system that captures the essential properties of containment using three *complexity annotations*:

$$\Phi ::= \mathsf{cnst} \mid \mathsf{flat} \mid \mathsf{exp}$$

The complexity annotation cnst refers to constant-step (although not necessarily terminating) expressions, flat refers to (a subset of) contained expressions, and exp refers to all expressions. Every constant-step expression is contained, and every contained expression is an expression. This gives rise to a natural ordering on complexity annotations and, by extension, to types.

The syntax of types is parameterized with respect to a set *TVar* of *type variable names*, α, β. The type language is stratified between value types U, V and types S, T. The latter include function types:

$$\mathrm{V} ::= \alpha \mid \mathsf{int} \mid \mathsf{bool} \mid \mathrm{V}^1 \qquad\qquad \mathrm{T} ::= \mathrm{V} \mid (\mathrm{U}_1, .., \mathrm{U}_\ell) \rightarrow \Phi\, \mathrm{V}$$

For function types, we require that $fv(\mathrm{V}) \subseteq \bigcup_i fv(\mathrm{U}_i)$. Functions are constrained to act over values. Additionally, the function body is constrained to be an expression with complexity Φ; thus, a function's type tells us something of how it evaluates.

The *subcomplexity* relation (notation $\Phi <: \Psi$) is defined to be the smallest preorder on complexity annotations such that $\mathsf{cnst} <: \mathsf{flat}$ and $\mathsf{flat} <: \mathsf{exp}$. The *subtype*

Table 5 Typing Rules: Part I

(VAL-INT)
$$\frac{}{\Gamma \vdash n : \mathsf{cnst\ int}}$$

(VAL-BOOL)
$$\frac{}{\Gamma \vdash bv : \mathsf{cnst\ bool}}$$

(VAL-SEQ)
$$\frac{\Gamma \vdash a_j : \mathsf{cnst}\ \mathrm{V} \quad (\forall j)}{\Gamma \vdash \langle a_1, .., a_n \rangle : \mathsf{cnst}\ \mathrm{V}^1}$$

(EXP-SUB)
$$\frac{\Gamma \vdash A : \Phi\ \mathrm{S} \quad \Phi <: \Psi}{\Gamma \vdash A : \Psi\ \mathrm{T} \quad \mathrm{S} <: \mathrm{T}}$$

(EXP-VAR)
$$\frac{\Gamma(x) = \mathrm{T}}{\Gamma \vdash x : \mathsf{cnst}\ \mathrm{T}}$$

(EXP-PRIM)
$$\frac{\delta(p) = \mathrm{T}}{\Gamma \vdash p : \mathsf{cnst}\ \mathrm{T}}$$

(EXP-LETREC)
$$\frac{\begin{array}{l}\Gamma, f : \widetilde{\mathrm{U}} \to \Phi\ \mathrm{V} \vdash f \Leftarrow \widetilde{x} D,E \\ \Gamma, f : \widetilde{\mathrm{U}} \to \Phi\ \mathrm{V} \vdash A : \Psi\ \mathrm{W}\end{array}}{\Gamma \vdash \mathsf{letrec}\ f \Leftarrow \widetilde{x} D,E\ \mathsf{in}\ A\ : \Psi\ \mathrm{W}}$$

(EXP-LET$_{\mathrm{E}}$)
$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{exp}\ \mathrm{U} \\ \Gamma, x : \mathrm{U} \vdash A : \mathsf{exp}\ \mathrm{V}\end{array}}{\Gamma \vdash \mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A : \mathsf{exp}\ \mathrm{V}}$$

(EXP-IT$_{\mathrm{E}}$)
$$\frac{\begin{array}{l}\Gamma \vdash B_i : \mathsf{exp}\ \mathrm{U}_i \quad (\forall i) \\ \Gamma, \widetilde{x} : \widetilde{\mathrm{U}} \vdash A : \mathsf{flat}\ \mathrm{V}\end{array}}{\Gamma \vdash \left[\widetilde{x} \Leftarrow \widetilde{B} \colon A\right] : \mathsf{exp}\ \mathrm{V}}$$

(EXP-IF$_{\mathrm{E}}$)
$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{exp\ bool} \\ \Gamma \vdash A : \mathsf{exp}\ \mathrm{V} \\ \Gamma \vdash C : \mathsf{exp}\ \mathrm{V}\end{array}}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C : \mathsf{exp}\ \mathrm{V}}$$

(EXP-APP$_{\mathrm{E}}$)
$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{exp}\ (\widetilde{\mathrm{U}} \to \mathsf{exp}\ \mathrm{V}) \\ \Gamma \vdash A_i : \mathsf{exp}\ (\mathrm{U}_i \pi) \quad (\forall i)\end{array}}{\Gamma \vdash B \widetilde{A} : \mathsf{exp}\ (\mathrm{V}\pi)}$$

(EXP-PAPP$_{\mathrm{E}}$)
$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{exp}\ (\widetilde{\mathrm{U}} \to \mathsf{exp}\ \mathrm{V}) \\ \Gamma \vdash A_i : \mathsf{exp}\ (\mathrm{U}_i^1 \pi) \quad (\forall i)\end{array}}{\Gamma \vdash B^1 \widetilde{A} : \mathsf{exp}\ (\mathrm{V}^1 \pi)}$$

(ENV-$\emptyset$)
$$\frac{}{\Gamma \vdash \emptyset}$$

(ENV-VAL)
$$\frac{\begin{array}{l}\Gamma \vdash x : \mathsf{cnst}\ \mathrm{V} \\ \Gamma \vdash a : \mathsf{cnst}\ \mathrm{V}\end{array}}{\Gamma \vdash x \Leftarrow a}$$

(ENV-UNION)
$$\frac{\begin{array}{l}\Gamma \vdash \sigma \\ \Gamma \vdash \rho\end{array}}{\Gamma \vdash \sigma, \rho}$$

(ENV-FUN$_{\mathrm{E}}$)
$$\frac{\begin{array}{l}\Gamma \vdash f : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \Phi\ \mathrm{V}) \\ (\Gamma \backslash_{\mathcal{F}}), \widetilde{x} : \widetilde{\mathrm{U}} \ \vdash D : \Phi\ \mathrm{V} \\ (\Gamma \backslash_{\mathcal{F}}), \widetilde{x} : \widetilde{\mathrm{U}}^1 \vdash E : \Phi\ \mathrm{V}^1 \quad C \overset{\star}{\rightsquigarrow} D\end{array}}{\Gamma \vdash f \Leftarrow \widetilde{x} D,E \quad \left\langle \widetilde{y} \Leftarrow \widetilde{x} \colon C\{\!\!\{{}^{\widetilde{y}}\!/\!_{\widetilde{x}}\}\!\!\} \right\rangle \overset{\star}{\rightsquigarrow} E}$$

relation (notation $\mathrm{S} <: \mathrm{T}$) is defined to be the smallest preorder on types such that $\Phi <: \Psi$ implies $\widetilde{\mathrm{U}} \to \Phi\ \mathrm{V} <: \widetilde{\mathrm{U}} \to \Psi\ \mathrm{V}$.

The typing rules are given in Tables 5 and 6. We prove that evaluation and transformation preserve typing. We also prove an important property of cnst expressions, described below. The significance of flat expressions is made clear in the proofs of Proposition 6.1c and Theorem 7.2 where the typing rules for flat are used in conjunction with Proposition 6.1a to prove the flattening transformations correct.

The judgments of the type system have the form:

$\Gamma \vdash a : \mathsf{cnst}\ \mathrm{V}$	Value a has type V.
$\Gamma \vdash A : \Phi\ \mathrm{T}$	Expression A has type T and complexity Φ .
$\Gamma \vdash \sigma$	Environment σ is well typed.

Here Γ is a type environment that maps type variables to types. Let us first look at Table 5, which gives the rules for values, exp expressions and environments. The three rules for values are given on the first line of the table. Ignoring the complexity annotations, these are standard rules for monomorphic sequences. The rule (VAL-SEQ), for example, states that in order for a sequence value to have type V^1, every element of the sequence must have type V. The complexity annotation cnst indicates that the construction of a literal value takes a constant number of steps (independent of the runtime environment).

Table 6 Typing Rules: Part II

(EXP-EIT$_{\mathrm{C}}$)
$$\dfrac{fv(A) \subseteq \widetilde{x} \qquad \Gamma \vdash xs_i : \mathsf{cnst}\ \mathrm{U}_i \ \ (\forall i) \qquad \Gamma, \widetilde{x} : \widetilde{\mathrm{U}} \vdash A : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \langle \widetilde{x} \Leftarrow \widetilde{xs} \colon A \rangle : \mathsf{cnst}\ \mathrm{V}}$$

(EXP-EIT$_{\mathrm{F1}}$)
$$\dfrac{fv(A) \subseteq \widetilde{x} \qquad \Gamma \vdash xs_i : \mathsf{cnst}\ \mathrm{U}_i \ \ (\forall i) \qquad \Gamma, \widetilde{x} : \widetilde{\mathrm{U}} \vdash A : \mathsf{flat}\ \mathrm{V}}{\Gamma \vdash \langle \widetilde{x} \Leftarrow \widetilde{xs} \colon A \rangle : \mathsf{flat}\ \mathrm{V}}$$

(EXP-LET$_{\mathrm{C}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ \mathrm{U} \qquad \Gamma, x : \mathrm{U} \vdash A : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A : \mathsf{cnst}\ \mathrm{V}}$$

(EXP-LET$_{\mathrm{F1}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ \mathrm{U} \qquad \Gamma, x : \mathrm{U} \vdash A : \mathsf{flat}\ \mathrm{V}}{\Gamma \vdash \mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A : \mathsf{flat}\ \mathrm{V}}$$

(EXP-LET$_{\mathrm{F2}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{flat}\ \mathrm{U} \qquad \Gamma, x : \mathrm{U} \vdash A : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \mathsf{let}\ x \Leftarrow B\ \mathsf{in}\ A : \mathsf{flat}\ \mathrm{V}}$$

(EXP-APP$_{\mathrm{C}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \mathsf{cnst}\ \mathrm{V}) \qquad \Gamma \vdash A_i : \mathsf{cnst}\ (\mathrm{U}_i\,\pi) \ \ (\forall i)}{\Gamma \vdash B\widetilde{A} : \mathsf{cnst}\ (\mathrm{V}\pi)}$$

(EXP-APP$_{\mathrm{F1}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \mathsf{flat}\ \mathrm{V}) \qquad \Gamma \vdash A_i : \mathsf{cnst}\ (\mathrm{U}_i\,\pi) \ \ (\forall i)}{\Gamma \vdash B\widetilde{A} : \mathsf{flat}\ (\mathrm{V}\pi)}$$

(EXP-APP$_{\mathrm{F2}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \mathsf{cnst}\ \mathrm{V}) \qquad \Gamma \vdash A_i : \mathsf{cnst}\ (\mathrm{U}_i\,\pi) \ \ (\forall i \neq h) \qquad \Gamma \vdash A_h : \mathsf{flat}\ (\mathrm{U}_h\,\pi)}{\Gamma \vdash B\widetilde{A} : \mathsf{flat}\ (\mathrm{V}\pi)}$$

(EXP-PAPP$_{\mathrm{C}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \mathsf{cnst}\ \mathrm{V}) \qquad \Gamma \vdash A_i : \mathsf{cnst}\ (\mathrm{U}^1_i\pi) \ \ (\forall i)}{\Gamma \vdash B^1\widetilde{A} : \mathsf{cnst}\ (\mathrm{V}^1\pi)}$$

(EXP-PAPP$_{\mathrm{F1}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \mathsf{flat}\ \mathrm{V}) \qquad \Gamma \vdash A_i : \mathsf{cnst}\ (\mathrm{U}^1_i\pi) \ \ (\forall i)}{\Gamma \vdash B^1\widetilde{A} : \mathsf{flat}\ (\mathrm{V}^1\pi)}$$

(EXP-PAPP$_{\mathrm{F2}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ (\widetilde{\mathrm{U}} \to \mathsf{cnst}\ \mathrm{V}) \qquad \Gamma \vdash A_i : \mathsf{cnst}\ (\mathrm{U}^1_i\pi) \ \ (\forall i \neq h) \qquad \Gamma \vdash A_h : \mathsf{flat}\ (\mathrm{U}^1_h\pi)}{\Gamma \vdash B^1\widetilde{A} : \mathsf{flat}\ (\mathrm{V}^1\pi)}$$

(EXP-IT$_{\mathrm{C}}$)
$$\dfrac{\Gamma \vdash B_i : \mathsf{cnst}\ \mathrm{U}_i \ \ (\forall i) \qquad \Gamma, \widetilde{x} : \widetilde{\mathrm{U}} \vdash A : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \left[\widetilde{x} \Leftarrow \widetilde{B} \colon A\right] : \mathsf{cnst}\ \mathrm{V}}$$

(EXP-IT$_{\mathrm{F1}}$)
$$\dfrac{\Gamma \vdash B_i : \mathsf{cnst}\ \mathrm{U}_i \ \ (\forall i) \qquad \Gamma, \widetilde{x} : \widetilde{\mathrm{U}} \vdash A : \mathsf{flat}\ \mathrm{V}}{\Gamma \vdash \left[\widetilde{x} \Leftarrow \widetilde{B} \colon A\right] : \mathsf{flat}\ \mathrm{V}}$$

(EXP-IT$_{\mathrm{F2}}$)
$$\dfrac{\Gamma \vdash B_i : \mathsf{flat}\ \mathrm{U}_i \ \ (\forall i) \qquad \Gamma, \widetilde{x} : \widetilde{\mathrm{U}} \vdash A : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \left[\widetilde{x} \Leftarrow \widetilde{B} \colon A\right] : \mathsf{flat}\ \mathrm{V}}$$

(EXP-IF$_{\mathrm{F1}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{flat}\ \mathsf{bool} \qquad \Gamma \vdash A : \mathsf{cnst}\ \mathrm{V} \qquad \Gamma \vdash C : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C : \mathsf{flat}\ \mathrm{V}}$$

(EXP-IF$_{\mathrm{F2}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ \mathsf{bool} \qquad \Gamma \vdash A : \mathsf{flat}\ \mathrm{V} \qquad \Gamma \vdash C : \mathsf{cnst}\ \mathrm{V}}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C : \mathsf{flat}\ \mathrm{V}}$$

(EXP-IF$_{\mathrm{F3}}$)
$$\dfrac{\Gamma \vdash B : \mathsf{cnst}\ \mathsf{bool} \qquad \Gamma \vdash A : \mathsf{cnst}\ \mathrm{V} \qquad \Gamma \vdash C : \mathsf{flat}\ \mathrm{V}}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ C : \mathsf{flat}\ \mathrm{V}}$$

(EXP-SUB) is a standard rule for subsumption; the side conditions specify constraints on Ψ and T. The rule for primitives (EXP-PRIM) makes use of the function δ which maps primitive names to types; the definition is elided. Both primitive and variable occurrences can be resolved dynamically in a constant number of steps and therefore are assigned complexity cnst.

The rule (EXP-LETREC) relies on the environment rule (ENV-FUN), described below. In (EXP-LETREC), also note that the complexity and type of the expression A need not be the same as the complexity or type of the function being defined. The rules for let-expressions (EXP-LET$_{\mathrm{E}}$) and conditionals (EXP-COND$_{\mathrm{E}}$) are standard. Note that using subsumption, these rules can be applied even if a subexpression is in cnst or flat. The iterator rule (EXP-IT$_{\mathrm{E}}$) is similar to the let-rule in its treatment of binders, as should be expected. Here, however, the bound expression A is required to be flat. This is an essential aspect of the typing system; the main purpose of the type system, after all, is to ensure that iterator expressions are correctly flattenable.

The rules for application and parallel application allow for the instantiation of type variables via a type substitution π. Note the difference between these rules. If B has type $\widetilde{\mathrm{U}} \to \mathsf{exp}\ \mathrm{V}$, then $B\widetilde{A}$ has type V, whereas $B^1\widetilde{A}$ has type V^1.

The rules for runtime environments are presented in the bottom row of the table. These are straightforward, but for (ENV-FUN). Note the difference in the treatment of two function bodies, D and E. Whereas D must evaluate to a value of type V, E must evaluate to a value of type V^1; the types of the input parameters are adjusted accordingly. The unusual side conditions enforce a syntactic relation between the two function bodies, formalizing the intuition that E and D must be derived from a common source C. The conditions are not onerous; in practice E is automatically generated from D. We write $(\Gamma\backslash_{\mathcal{F}})$ for the type environment derived by removing all value-typed variables from Γ. Thus the type rules require that function declarations be fully parameterized; *i.e.* D and E cannot refer to free value variables.

We now turn to Table 6. Here we find the first rule for e-iterators; the rule has a side condition requiring that all variables in the iterator expression be bound.

The table is best read in columns. The first column gives rules for cnst expressions. The exception is the conditional. Expressions that include a conditional may take a varying number of steps depending on the value of the condition, which may in turn depend on the runtime environment; therefore, no conditional expression is in cnst. Note that cnst expressions can be recursive, although in this case the typing rules guarantee that they are nonterminating, since no conditionals are allowed in cnst expressions.

The second and third columns give rules for flat expressions. These require that at most one subexpression is flat, all others are cnst, ensuring that at most one sequential component is recursive. This is a sufficient condition for containment.

It is important to emphasize that our typing system is not overly conservative. For example, all but one of the programs on the Scandal website `http://www.cs.cmu.edu/~scandal/` can be typed using our system (although some require trivial rewriting). Potential improvements are discussed in Section 8.

Proposition 6.1. (a) *Suppose that* $\Gamma \vdash \sigma$ *and* $\Gamma \vdash \rho$ *and that* σ *and* ρ *differ only in their value bindings. If* $\Gamma \vdash D : \mathsf{cnst}\ \mathrm{V}$, $\sigma \vdash D \xrightarrow[w]{t} d$ *and* $\rho \vdash D \xrightarrow[w']{t'} d'$, *then* $t = t'$.
(b) *If* $\Gamma \vdash \sigma$ *and* $\Gamma \vdash D : \Phi\ \mathrm{T}$ *and* $\sigma \vdash D \xrightarrow[w]{t} d$, *then* $\Gamma \vdash d : \mathsf{cnst}\ \mathrm{T}$.
(c) *If* $\Gamma \vdash D : \Phi\ \mathrm{T}$ *and* $D \rightsquigarrow D'$, *then* $\Gamma \vdash D' : \Phi\ \mathrm{T}$. □

7 Correctness of the Reference Implementation

We can now state the main result.

Definition 7.1. D is *weakly improved by* E *under* Γ (notation $D \gtrapprox_\Gamma E$) if $\Gamma \vdash D$, $\Gamma \vdash E$, and there exist constants u and v such that for all σ such that $\Gamma \vdash \sigma$,

$$\sigma \vdash D \xrightarrow[w]{t} d \ \text{ implies } \ \sigma \vdash E \xrightarrow[\leqslant v \cdot w]{\leqslant u \cdot t} d \qquad \square$$

Theorem 7.2. *If* $\Gamma \vdash A$ *and* $A \overset{\star}{\rightsquigarrow} B$ *then* $A \gtrapprox_\Gamma B$. □

This theorem is very hard to prove directly. Weak improvement has some nice properties; for example it is a preorder. However, it is not substitutive. In light of the context rules given in Table 2, this makes it very difficult to prove the transformations correct directly.

To prove Theorem 7.2, we define an alternative, *costed* semantics, and, using this, a *strong improvement* relation $\underset{\sim}{\rhd}_{\Gamma}$. Strong improvement is a congruence that allows us to establish the following results, which together prove Theorem 7.2.

$$
\begin{array}{l}
A \underset{\sim}{\rhd}_{\Gamma} B \text{ implies } A \underset{\approx}{\rhd}_{\Gamma} B \\
\Gamma \vdash A \text{ and } A \overset{\star}{\rightsquigarrow} B \text{ imply } A \underset{\sim}{\rhd}_{\Gamma} B
\end{array}
$$

The close relation between standard evaluation and costed evaluation (denoted $\Vdash\!\!\longrightarrow$) is given by a costing function $\mathcal{C}$, which is determined by the syntax of a term. We have:

$$
\begin{array}{lll}
\sigma \vdash D \Vdash\!\!\xrightarrow[w]{t} d & \text{implies} & \sigma \vdash D \xrightarrow[\leqslant w]{\leqslant t} d \\
\sigma \vdash D \xrightarrow[w]{t} d & \text{implies} & \sigma \vdash D \Vdash\!\!\xrightarrow[\leqslant \mathcal{C}(\sigma \vdash D)\cdot w]{\leqslant \mathcal{C}(\sigma \vdash D)\cdot t} d
\end{array}
$$

Our proof technique is similar to Sands' use of the tick algebra [29]. We introduce "ticks" in the costed semantics in order to account for the costs introduced later by the transformations. There are differences, however; for example, our "ticks" depend on the nesting depth of iterators and conditionals. Unfortunately, a detailed discussion is beyond the scope of this extended abstract.

8 Related and Future Work

This paper is derived from [27] which closes many problems left open in [28], where we first outlined our approach. The techniques and results in this paper are all new; in particular, [28] does not mention the typing system, the costed semantics, strong improvement, nor the counterexample for the adequacy of containment.

Several other authors have considered the implementation of nested parallelism via flattening transformations. Steele and Hillis [32] presented a set of laws for relating expressions that include an apply-to-each operator. Blelloch and Sabot [7] picked up on this theme to define a flattening compiler for Paralation-LISP, which became the basis for NESL. Prins and Palmer [24] presented a different form of flattening using program transformations; this approach was further refined in [21,20] and here. The thread-based execution model of nested parallelism has been shown to respect the step and work complexities of the source-level metrics [9,5]. However, overheads and space requirements in the realization of this model require careful run-time scheduling [4], fast synchronization [25], and granularity control (in the sense of [10]) to make it practical. Blelloch [1] and Suciu and Tannen [34,33], have presented nested parallel languages and have argued that these languages can be implemented on the VRAM with the correct step/work complexity. However, these results are based on simulation techniques rather than explicit source-to-target translations.

Skillicorn and Cai [31] presented a cost calculus for parallel programs using the Bird-Meertens formalism. This approach has been developed further by Jay [16,15], using shape analysis. Another promising direction is that of Keller, who develops transformations that take distribution into account [17]. In this setting, flattening can profitably be combined with deforestation and related techniques [35,18,11].

Nested data-parallelism may be seen as a particular form of the more general and-parallelism found in logic programs [13]. Research on the parallel execution of logic programs has explored ideas similar to flattening to reduce communication [26] and scheduling overheads [13,22] for restricted nested and-parallel constructs. These are presented as optimizations but there are no formal performance guarantees. A source-level cost semantics is used in [10] to control the compilation and run-time execution of parallel logic programs.

Our notion of weak improvement is similar that developed in [19,12]. There, however, the relation is a congruence by construction; it is the least congruence contained in our (stronger) relation. In our setting, little is gained by forcing weak improvement to be a congruence; therefore, we use the simpler definition.

There are several possibilities for further work. We believe it is possible to weaken the typing system to allow for sequential composition of flat expressions. Currently we require that for '$\mathsf{let}\ x \Leftarrow A\ \mathsf{in}\ B$' to be in flat, either A or B must be in cnst. It appears that both A and B could be in flat; however, we have not yet been able to establish a correctness proof in this case. We plan to implement our reference-based semantics with the intention of deriving an experimental measure of its performance. We would also like to adapt our results to the "construct-results" costing function outlined in [28]; this costing function allows the use of the segment-vector representation of nested sequences without the compromising the usability of the semantics.

Acknowledgements

We thank the referees and the DePaul Foundations of Programming Languages group for useful comments and references to related work. The work reported here began while the first author was a student at the University of North Carolina and was completed during an appointment at North Carolina State University. I would like to thank both institutions, and Rance Cleaveland who supervised me at NCSU, for their support.

References

1. G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
2. G. E. Blelloch. NESL: A nested data-parallel language (version 3.0). Technical report, Carnegie-Mellon University, Department of Computer Science, 1994.
3. G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), 1996.
4. G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank conetention and delay in high-bandwidth multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 84–94, Santa Barbara, CA, July 1995. ACM Press.
5. G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *International Conference on Functional Programming*, 1996.
6. G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.
7. G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.
8. R. P. Brent. The parallel evaluation of generic arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

9. D. Engelhardt and A. Wendelborn. A partitioning-independent paradigm for nested data parallelism. *International Journal of Parallel Programming*, 24(4):291–317, Aug. 1996.
10. P. L. Garcia, M. Hermenegildo, and S. K. Debray. A methodology for granularity based control of parallelism in logic programs. *J. of Symbolic Computation*, 22:715–734, 1998.
11. A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 58–67, Santa Barbara, July 1995.
12. J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In A. D. Gordon and A. M.Pitts, editors, *The Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.
13. M. Hermenegildo and M. Carro. Relating data-parallelism and (and-)parallelism in logic programs. *The Computer Languages Journal*, 22(2/3):143–163, July 1996.
14. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell version 1.2. *ACM SIGPLAN notices*, 27(5), May 1992.
15. C. Jay. The FISh language definition. http://www-staff.socs.uts.edu.au/~cbj/Publications/fishdef.ps.gz, 1998.
16. C. Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, 1999.
17. G. Keller. *Transformation-Based Implementation of Nested Data-Parallelism for Distributed Memory Machines.* PhD thesis, TU Berlin, 1999.
18. J. Launchbury and T. Sheard. Warm fusion: deriving build-catas from recursive definitions. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, CA, June 1995.
19. A. K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 43–56, San Antonio, Jan. 1999. ACM Press.
20. D. W. Palmer. *Efficient Execution of Nested Data Parallel Programs*. PhD thesis, University of North Carolina, 1996.
21. D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Frontiers '95*, 1995.
22. E. Pontelli and G. Gupta. Nested parallel call optimization. In *International Parallel Processing Symposium*. IEEE Computer Society Press, 1996.
23. J. Prins, M. Ballabio, M. Boverat, M. Hodous, and D. Maric. Fast primitives for irregular computations on the nec sx-4. *Crosscuts*, 6(4), 1997.
24. J. F. Prins and D. W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, May 1993. (ACM SIGPLAN Notices, 28(7), , 1993).
25. V. Ramakrishnan, I. Sherson, and R. Subramanian. Efficient techniques for fast nested barrier synchronization. In *ACM Symposium on Parallel Algorithms and Architectures*, 1995.
26. B. Ramkumar and L. Kale. Compiled execution of the reduced-or process model on multiprocessors. In *North American Conference on Logic Programming*. MIT Press, 1989.
27. J. Riely. *Applications of Abstraction for Concurrent Programs*. PhD thesis, University of North Carolina at Chapel Hill, 1999.
28. J. Riely, J. Prins, and S. Iyer. Provably correct vectorization of nested-parallel programs. In *Programming Models for Massively Parallel Computers (MPPM'95)*, Berlin, Dec. 1995.
29. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(10), Oct. 1996.
30. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, 1996.

31. D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming, 1994. Queens University Department of Computer Science TR-93-348.
32. G. L. Steele and W. D. Hillis. Connection machine Lisp: Fine-grained parallel symbolic processing. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 279–297, Cambridge, MA, Aug. 1986. ACM Press.
33. D. Suciu. *Parallel Programming Languages for Collections*. PhD thesis, University of Pennsylvania, 1995.
34. D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, June 1994.
35. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.