From *Programming Models for Massively Parallel Computers (MMPM'95)*,
IEEE Computer Society Press, 1996.

Provably Correct Vectorization of Nested-Parallel Programs

James Wheelis Riely,[1] Jan Prins[1] and S. Purushothoman Iyer[2]

[1] *Univ. of North Carolina*
Chapel Hill, NC 27599-3175 USA
{riely,prins}@cs.unc.edu

[2] *North Carolina State Univ.*
Raleigh, NC 27695-8206 USA
purush@csc.ncsu.edu

Abstract

The work/step framework provides a high-level cost model for nested data-parallel programming languages, allowing programmers to understand the efficiency of their codes without concern for the eventual mapping of tasks to processors. Vectorization, or flattening, is the key technique for compiling nested-parallel languages. This paper presents a formal study of vectorization, considering three low-level targets: the EREW, bounded-contention CREW, and CREW variants of the VRAM. For each, we describe a variant of the cost model and prove the correctness of vectorization for that model. The models impose different constraints on the set of programs and implementations that can be considered; we discuss these in detail.

1 Introduction

Many complexity models (or *cost* models) have been proposed for parallel programs. *High-level* models such as Blelloch's step/work metrics for NESL [3, 2] and Skillicorn's calculus for BMF [13] are based on a rich, highly-parallel expression language with *compositional* cost metrics: the complexity of an expression can be understood by combining the complexities of its subexpressions. *Low-level* models such as the PRAM [8], VRAM [2] and BSP model [14] are statement-based with very limited compositionality.

Cost models are important for programming because they help guide the construction of efficient code. High-level models allow each code-fragment to be considered separately, so that optimizing any fragment in isolation should improve overall performance.

There is a danger, however, in using high-level models that do not have a proven relation to a lower-level machine model and—therefore—to a working implementation. An inaccurate cost model may be worse than no model at all; it may encourage "optimizations" that *diminish* the performance on an actual parallel machine.

In this paper, we prove that the high-level step/work metric accurately reflects the implementation of a nested-parallel language on a VRAM. Blelloch [2] proved a similar result for a limited class of expressions with no free variables, but until now his work has stood in isolation. Our proofs are both more formal and more general than his, although we also must restrict our attention to *contained* programs (see Section 4). Our work has been motivated by—and is complementary to—ongoing implementation and performance studies of the Proteus programming language [7, 10].

Nested-parallel languages such as NESL and Proteus are characterized by a nested sequence datatype along with a set of second-order functions to manipulate them. The main source of parallelism is the apply-to-each function; both NESL and Proteus have a special form for this, the *iterator*. For example, the expression $\left[\mathsf{x} \leftarrow \langle 1..\mathsf{n}\rangle : \bar{\mathsf{z}}[\mathsf{x}]\right]$ indexes the sequence stored in the variable $\bar{\mathsf{z}}$ by each of the values 1 through n, returning $\left\langle \bar{\mathsf{z}}[1], ..., \bar{\mathsf{z}}[\mathsf{n}]\right\rangle$.

Roughly speaking, the *step* complexity gives the running time under the assumption that all specified parallelism is realized; the *work* complexity counts the total number of (scalar) operations performed.

Like the PRAM, the VRAM model has variants such as EREW and CREW. In the EREW case, the step/work metrics accurately predict performance on parallel machines that have sufficient memory bandwidth, such as vector machines. The step/work metrics also work well for CREW and CRCW models as long as memory contention is bounded by a small constant. In general, however, step and work complexity fail to predict eventual performance when a VRAM supports CREW operations with unbounded contention. For a discussion, see [5].

In Section 3 we define a profiling semantics for a simple nested-parallel programming language, giving an explicit account of the step/work paradigm for this language. For each possible input, the semantics specifies not only the final output of a program, but also the number of steps and total work required before the program halts on that output. This semantics induces an equivalence on programs: two programs are equivalent if on every possible input they produce the same results with the same steps and work. This equivalence is finer than we would like, however, because it is sensitive to constant factors. We define a more abstract equivalence which ignores constant factors: the *asymptotic efficiency* (AE) equivalence. Since we consider programs modulo this

$$\begin{tikzcd}
\boxed{\mathit{Prog}} \arrow[d, "\mathit{xform}"'] \arrow[drr, "\llbracket\cdot\rrbracket"] & & \\
\boxed{\text{I.F. } \mathit{Prog}} \arrow[r, "\llbracket\cdot\rrbracket"] & \underset{\approx}{\sqsupseteq} & \boxed{\mathit{Env} \to \mathit{Val}_\perp \times \mathbb{N}^\infty \times \mathbb{N}^\infty}
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{\text{I.F. } \mathit{Prog}} \arrow[r, "\llbracket\cdot\rrbracket"] \arrow[d, "\mathit{impl}"'] & \underset{\approx}{\sqsubseteq} & \boxed{\mathit{Env} \to \mathit{Val}_\perp \times \mathbb{N}^\infty \times \mathbb{N}^\infty} \\
\boxed{\text{VRAM}} \arrow[rr, "\llbracket\cdot\rrbracket'"] & & \boxed{\mathit{Env}' \to \mathit{Val}'_\perp \times \mathbb{N}^\infty \times \mathbb{N}^\infty} \arrow[u, "\mathit{impl}^{-1}"']
\end{tikzcd}$$

Figure 1: Proof outline.

equivalence, we call our semantics an "AE semantics." The notion of an AE equivalence may be new.

A formal semantics is a contract between a language's implementors and its users. Traditionally this contract has been limited to the extensional aspects of a language; in an AE semantics, intensional issues (that is, performance) must be addressed as well. Tighter cost bounds in the semantics make things harder for the implementors, but easier for the programmers: they get more ways to write efficient code.

In our language, the treatment of free variables has a profound effect on the nature of this contract, and it is the explication of this issue that is the main contribution of our work, occupying Sections 4–6. In summary, we report three alternative semantics:

• An *ideal* semantics, in which free variables are truly *free*; the step complexity of an iterator is simply the maximum of the steps taken by the iterator expression under each binding of the variables. This semantics forces a reference-based implementation of sequences, with the necessity of unbounded contention on a CREW VRAM.

• A *construct-parameters* semantics which copies out the free variables—once for each binding—then evaluates the iterator expression under each binding simultaneously. This allows for implementation on an EREW VRAM but makes the cost model sensitive to the presence of free variables, leading to counterintuitive complexity results and complicating programming. This is the approach taken in a recent implementation of NESL [3].

• A *construct-results* semantics which strikes a middle ground. It removes sensitivity to free-variables from the cost model while allowing implementation on a low-contention CREW VRAM. There is some loss of generality, however, which can be partially recouped using static analysis. This is the approach taken in the current implementation of Proteus [10].

x–$z \in \mathit{Var}$	Variable names
f–$g \in \mathit{UserFun}$	(user-defined) Function names
$p \in \mathit{PrimOp}$	Primitive (operation) names
a–$e \in \mathit{Expr}$	Expressions
F–$G \in \mathit{FunDecl}$	Function declaration
P–$Q \in \mathit{Prog}$	Programs
$\textsc{a}$–$\textsc{e} \in \mathit{Val}$	Values
σ–$\rho \in \mathit{Env}$	Variable environment

Table 1: Usual meaning of Metavariables.

We prove that these semantics are implementable on a CREW, EREW, and bounded-contention CREW VRAM, respectively. The proof is accomplished in two steps, outlined in Section 1: source-to-source transformation to eliminate iterators, followed by implementation of the *iterator-free* language on the VRAM. (The terms in the figure are defined in Section 3.) This paper is devoted almost entirely to the first part; we address the second in Sections 4–6, but only informally. While we would like to formalize this second step, we believe that it has already been well established by implementation, experimentation, and some less-formal proofs [4, 2, 10].

Because of space limitations, we have cut quite a bit from this extended abstract. We assume that the reader has a familiarity with the basic notions of operational semantics [15]. For pointers to the literature on high-level cost models, see the excellent summaries in articles by Blelloch and Greiner [6] and Skillicorn and Cai [13]; these two articles are closely related to ours.

2 The language

Throughout the paper we use many metavariables, most of which are listed in Section 1. The notation "x–$z \in \mathit{Var}$" indicates that the symbols x through z—possibly decorated with overlines, primes, sub- or super-scripts—are used to range over the set Var. We use h–n for naturals and r–v for reals and write multiplication as infix "$*$".

2.1 Values and Types

As a convenience every value in our language is considered an element of a sequence type. A sequence type comprises a scalar base type and a depth. For simplicity, we consider only two scalar types, integers and Booleans. Zero-depth sequences are simply scalars. Boolean constants are drawn from the set $\{\mathsf{t}, \mathsf{f}\}$. We write (non-scalar) sequence values as lists of elements between angled brackets; for example, $\langle\rangle$ is the empty sequence, and $\langle\langle 1, 2\rangle, \langle 3, 4, 5\rangle, \langle\rangle\rangle$

is a sequence of three elements, a "sequence of sequences of integers." All sequences have uniform depth. The special value err represents run-time error and inhabits all types.

We also use overlines to write literal sequences; thus, $\langle \overline{1\,2}, \overline{3\,4\,5}, {}^{\bullet} \rangle$, $\overline{\langle 1,2\rangle \;\; \langle 3,4,5\rangle \;\; \langle\rangle}$ and $\overline{\overline{1\,2} \;\; \overline{3\,4\,5} \;\; \bullet}$ are all synonymous with the previous example. This use of overlines is distinct from that for metavariables. When decorating a metavariable, the number of overlines indicates the minimum depth of the value; thus $\overline{\overline{A}}$ is a value whose depth is at least two. The overline is not an "operator" on metavariables: there is no implied relation between the values $\overline{A}$ and A.

We define the *depth* $\mathcal{D}$, *length* $\mathcal{L}_1$ and *size* $\mathcal{S}$ of a value as follows:

$$\mathcal{D}A \overset{def}{=} \begin{cases} 0 & \text{, if } A \text{ is a scalar} \\ 1 + \mathcal{D}A_j & \text{, any } j \text{, if } A = \langle A_j \rangle_{j=1}^{n} \end{cases}$$

$$\mathcal{L}_1 A \overset{def}{=} n, \text{ if } A = \langle A_j \rangle_{j=1}^{n}$$

$$\mathcal{S}A \overset{def}{=} \begin{cases} 1 & \text{, if } A \text{ is a scalar} \\ 1 + \sum_{j=1}^{n} \mathcal{S}A_j & \text{, if } A = \langle A_j \rangle_{j=1}^{n} \end{cases}$$

In the case of the example

$$C = \overline{\overline{\overline{1} \;\; \overline{2\,3\,4} \;\; \overline{5\,6}} \quad \overline{\overline{7\,8} \;\; \bullet} \quad \bullet}$$

we have $\mathcal{D}C = 3$, $\mathcal{L}_1 C = 3$, and $\mathcal{S}C = 17$.

2.2 Expressions

The abstract syntax of expressions is given inductively as follows, where a–e are expressions:

$$\begin{array}{rcl} a & ::= & x \mid A \\ & \mid & p^{k}(a_1, \ldots, a_\ell) \\ & \mid & f^{k}(a_1, \ldots, a_\ell) \\ & \mid & \mathsf{let}\ x = a\ \mathsf{in}\ c \\ & \mid & \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c \\ & \mid & [y_1 \leftarrow \overline{a}_1, \ldots, y_m \leftarrow \overline{a}_m :\ c] \end{array}$$

Here x is a variable dereferencing, and A a constant. The let and conditional constructs are standard.

Primitive operations of arity ℓ are applied by writing $p^k(a_1, \ldots, a_\ell)$; likewise for user-defined functions. We usually drop the parentheses if $\ell \leq 1$. Application is call-by-value. The superscript k indicates the depth at which the operation is to be applied. If k is zero, the application is *basic* otherwise it is *lifted*; we usually drop k when it is zero. To avoid error, the nesting structures of the arguments must be identical down to depth k. For example:

$$\mathsf{plus}\,(5,6) = 11 \qquad \mathsf{plus}^1\,(\overline{4\,3\,1}, \overline{3\,6\,7}) = \overline{7\,9\,8}$$

$$\mathsf{plus}^2\left(\overline{{}^{\bullet}\,\overline{2\,3}}, \overline{{}^{\bullet}\,\overline{7\,1}}\right) = \overline{{}^{\bullet}\,\overline{9\,4}} \qquad \mathsf{plus}^1\,(\overline{4\,3\,1}, \overline{3\,6}) = \mathsf{err}$$

Lifted calls need not appear in a source program; any lifted call can be written using iterators and basic calls. However, both basic and 1-lifted calls are necessary in the iterator-free language of Section 4. The generalization to k-lifted sequences is a convenience: it simplifies the program transformations and helps to ensure the production of efficient code.

Both the let and the iterator construct bind variables; neither construct allows for recursive definition. The free variables in an expression ($\mathsf{free}\ e$) can be determined statically using the usual definition. We write *a dependent-on x* if x occurs free in a, and *a independent-of x* otherwise.

The iterator construct allows several equal-length sequences $\overline{a}_h$ to be drawn from in correspondence. Thus $[\mathsf{x} \leftarrow \overline{\mathsf{x}}, \mathsf{y} \leftarrow \overline{\mathsf{y}} :\ \mathsf{x} + \mathsf{y}]$ is the same as $\mathsf{plus}^1(\overline{\mathsf{x}}, \overline{\mathsf{y}})$. The expressions $\overline{a}_h$ are evaluated *independently*.

We extend the ideas of depth, length and size to expressions; thus if an expression has depth two then the values that it yields are at least two deep.

2.3 Programs

A *function declaration* is a set of mutually recursive definitions:

$$f_1(x_1, \ldots, x_{\ell_1}) \overset{def}{=} d_1$$
$$\vdots$$
$$f_m(x_1, \ldots, x_{\ell_m}) \overset{def}{=} d_m$$

For f_h, we call x_1 through x_{ℓ_h} *parameters* and d_h the *body*. Within a function body the only variable names that may occur free are those of the parameters; any function name may appear.

A program is a pair $P \equiv (F, e)$ of a function declaration and an expression. We require that programs be well typed, although we do not present the details of the rules for well-typing, which are very standard.* In addition, a function name that occurs anywhere in a program must have exactly one definition in the function declaration.

The variables that occur free in e are called *input variables*. Intuitively, a program is executed by reading-in values corresponding to each of the input variables; the expression e is then evaluated in the context of these variable bindings and of the function declaration F.

*Well typing requires that input variables and occurrences of the empty sequence be typed. If one imposes the restriction that input variables have ground types, then polymorphic functions can be eliminated via specialization and the depth of expressions in the resulting program can be determined statically. NESL imposes this restriction, allowing its implementors to claim that every instance of a primitive call has constant step complexity.

2.4 Basic primitive operations

Although we do not precisely specify them, we assume a set of basic primitive operations on scalar types—perhaps with some special notation. For example, we make use of plus (infix $+$), not (prefix $\neg$), and $\mathsf{less\text{-}than}$ (infix $<$).

Below, we give extensional descriptions of the sequence primitives that we use. The running-time of a primitive depends upon its implementation; we discuss implementations and an intermediate form of primitive *specification* in Section 3.3. Further motivation for the use of these primitives can be found in [12, 10, 2,], and formal definitions in the full paper.

• build_ℓ *builds* a ℓ-length sequence out of its ℓ arguments. For example: $\mathsf{build}_3(1,2,3) = \langle 1,2,3\rangle$. We allow a special notation with ℓ implicit: $\langle a,b\rangle \overset{def}{=} \mathsf{build}_2(a,b)$.

• len returns the *length* of its argument. For example: $\mathsf{len}\,\langle 9,8,7,6\rangle = 4$. We allow a special notation: $\#a \overset{def}{=} \mathsf{len}\ a$.

• dist *distributes* a value, making a number of copies. For example, $\mathsf{dist}(5,\overline{1\,2}) = \overline{\overline{1\,2}\ \overline{1\,2}\ \overline{1\,2}\ \overline{1\,2}\ \overline{1\,2}}$.

• iota is a function familiar from APL. dom is a related function that returns the *domain* of a sequence when that sequence is viewed as a function from positive integers to values. For example: $\mathsf{iota}\ 5 = \mathsf{dom}\ \overline{9\,8\,7\,6\,5} = \overline{1\,2\,3\,4\,5}$. In general, $\mathsf{iota}\ n = \langle 1,2,...,n\rangle$, and $\mathsf{dom}\ \overline{A} = \langle 1,2,...,\#\overline{A}\rangle$. We allow a special notation: $\langle 1..a\rangle \overset{def}{=} \mathsf{iota}\ a$.

• rstr *restricts* a sequence to match the true elements of a companion Boolean sequence. For example: .16667em

$$\begin{aligned}\mathsf{rstr}(\overline{\mathsf{t\,f\,t\,f\,f}},\overline{1\,3\,2\,4\,5}) &= \overline{1\,2}\\ \mathsf{rstr}(\overline{\mathsf{t\,f\,t\,f\,f}},\overline{1\,3\,2\,4}) &= \mathsf{err}\end{aligned}$$

• comb *combines* two sequences based on a Boolean sequence. For example: .16667em

$$\begin{aligned}\mathsf{comb}(\overline{\mathsf{t\,f\,t\,f\,f}},\overline{1\,2},\overline{3\,4\,5}) &= \overline{1\,3\,2\,4\,5}.\\ \mathsf{comb}(\overline{\mathsf{t\,f\,t\,f\,f}},\overline{1\,2},\overline{3\,4}) &= \mathsf{err}\end{aligned}$$

Let not^1 be the lifted logical negation operator. When $\#\overline{B} = \#\overline{A}$, rstr and comb are complementary in the following sense: $\mathsf{comb}(\overline{B},\mathsf{rstr}(\overline{B},\overline{A}),\ \mathsf{rstr}(\mathsf{not}^1\overline{B},\overline{A})) = \overline{A}$.

In order to describe the remaining primitives concisely, we use the following example sequence:

$$A = \overline{\overline{\overline{1}\ \overline{2\,3\,4}\ \overline{5\,6}}\ \ \overline{\overline{7\,8}\ \bullet}\ \ \bullet}$$

• elt_ℓ is the *element retrieval* or "index" operation. It takes a sequence and ℓ integers. An error occurs if an attempt is made to index beyond the boundaries of a sequence. For example: .16667em

$$\begin{aligned}\mathsf{elt}_1(A,1) &= \overline{\overline{1}\ \overline{2\,3\,4}\ \overline{5\,6}}\\ \mathsf{elt}_3(A,1,3,2) &= 6\\ \mathsf{elt}_3(A,1,3,3) &= \mathsf{err}\end{aligned}$$

We allow a special notation with ℓ implicit: $a[b,c] \overset{def}{=} \mathsf{elt}_2(a,b,c)$.

• extr_k *extracts* k levels from a sequence, flattening it. For example:

$$\begin{aligned}\mathsf{extr}_0\ A &= A\\ \mathsf{extr}_2\ A &= \overline{1\,2\,3\,4\,5\,6\,7\,8}\end{aligned}$$

• insrt_k *inserts* the top k levels of the structure from one sequence onto another. For example: .16667em

$$\begin{aligned}\mathsf{insrt}_0(A,\overline{9\,8\,7}) &= \overline{9\,8\,7}\\ \mathsf{insrt}_2(A,\overline{9\,8\,7\,6\,5\,4\,3\,2}) &= \overline{\overline{\overline{9}\ \overline{8\,7\,6}\ \overline{5\,4}}\ \ \overline{\overline{3\,2}\ \bullet}\ \ \bullet}\\ \mathsf{insrt}_2(A,\overline{9\,8\,7\,6\,5\,4\,3}) &= \mathsf{err}\end{aligned}$$

There is a close relation between extr and insrt; for all $k \geq 0$: $\mathsf{insrt}_k(A,\mathsf{extr}_k A) = A$.

• prom_k *promotes* a value to match the top k levels of the structure of a sequence. For example:

$$\begin{aligned}\mathsf{prom}_0(A,6) &= 6\\ \mathsf{prom}_3(A,6) &= \overline{\overline{\overline{6}\ \overline{6\,6\,6}\ \overline{6\,6}}\ \ \overline{\overline{6\,6}\ \bullet}\ \ \bullet}\end{aligned}$$

3 Semantics and complexity

We present a profiling semantics for the language given in the last section and define an asymptotic efficiency (AE) preorder on programs. Intuitively, Q is AE-greater than P if they compute the same things and Q is more efficient than P, with allowances made for constant overhead and slowdown. It is these "allowances" that make our preorder an *asymptotic* efficiency preorder. *Simple* efficiency preorders [1] do not make such allowances, but rather compare the absolute running times of programs.

In addition to being *more efficient*, we also allow that an AE-greater program be *more defined*. This coarser definition lessens the burden of proof in Sections 4–6, allowing us to simplify the program transformations used there. In this context, non-termination is *less defined* than run-time error which is in turn *less defined* than any other value. Thus, the smallest programs with respect to the preorder are those that never terminate. (Note that if our language included error-handling, we would have to use the finer preorder; this would require us to change the transformation rules ID and CONST.)

Ideally an implementation should be AE-equivalent to it's specification in the semantics. However, we believe that it is acceptable for an AE-semantics to *overestimate* the cost of a program in some cases. If the overestimation is too great, however, the semantics looses much of its value.

3.1 The profiling semantics

In giving the semantics of our language we make use of the notion of an *environment*, which is a mapping from

$x \in \mathit{Var})$	$\sigma \vdash_F x \Rightarrow \sigma x \vdash ;1 \vdash ;1$	
$A \in \mathit{Val})$	$\sigma \vdash_F A \Rightarrow A \vdash ;\mathcal{D}A \vdash ;\mathcal{S}A$	$A \neq \mathsf{err}$
$\mathit{Let})$	$\dfrac{\begin{array}{c}\sigma \vdash_F a \Rightarrow A \vdash ; t_a \vdash ; w_a \\ \sigma\{\!\!\{A/x\}\!\!\} \vdash_F c \Rightarrow C \vdash ; t_c \vdash ; w_c\end{array}}{\sigma \vdash_F \mathsf{let}\ x = a\ \mathsf{in}\ c \Rightarrow C \vdash ;t_a + t_c \vdash ;w_a + w_c}$	$A \neq \mathsf{err}$
$\mathit{Cond})$	$\dfrac{\begin{array}{c}\sigma \vdash_F a \Rightarrow \mathsf{t} \vdash ; t_a \vdash ; w_a \\ \sigma \vdash_F b \Rightarrow B \vdash ; t_b \vdash ; w_b\end{array}}{\sigma \vdash_F \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c \Rightarrow B \vdash ;t_a + t_b \vdash ;w_a + w_b}$	
	$\dfrac{\begin{array}{c}\sigma \vdash_F a \Rightarrow \mathsf{f} \vdash ; t_a \vdash ; w_a \\ \sigma \vdash_F c \Rightarrow C \vdash ; t_c \vdash ; w_c\end{array}}{\sigma \vdash_F \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c \Rightarrow C \vdash ;t_a + t_c \vdash ;w_a + w_c}$	
$p \in \mathit{PrimOp})$	$\dfrac{\{\ \sigma \vdash_F a_i \Rightarrow A_i \vdash ; t_{a_i} \vdash ; w_{a_i}\ \}_{i=1}^{\ell}}{\begin{array}{r}\sigma \vdash_F p^0(a_1,\dots,a_\ell) \Rightarrow D \vdash ;\big(\sum_{i=1}^{\ell} t_{a_i}\big) + t_d \\ \vdash ;\big(\sum_{i=1}^{\ell} w_{a_i}\big) + w_d\end{array}}$	$A_i \neq \mathsf{err}$, $\delta_p(A_1,\dots,A_\ell) = D \vdash ;t_d \vdash ;w_d$
$f \in \mathit{UserFun})$	$\dfrac{\begin{array}{c}\{\ \sigma \vdash_F a_i \Rightarrow A_i \vdash ; t_{a_i} \vdash ; w_{a_i}\ \}_{i=1}^{\ell} \\ \{\!\!\{A_1/x_1,\dots,A_\ell/x_\ell\}\!\!\} \vdash_F d \Rightarrow D \vdash ; t_d \vdash ; w_d\end{array}}{\begin{array}{r}\sigma \vdash_F f^0(a_1,\dots,a_\ell) \Rightarrow D \vdash ;\big(\sum_{i=1}^{\ell} t_{a_i}\big) + t_d \\ \vdash ;\big(\sum_{i=1}^{\ell} w_{a_i}\big) + w_d\end{array}}$	$A_i \neq \mathsf{err}$, $f(x_1,\dots,x_\ell) \overset{\mathit{def}}{=} d$ in F
$\Big(g \in \mathit{PrimOp} \cup \mathit{UserFun}\Big)$	$\dfrac{\begin{array}{c}\{\ \sigma \vdash_F \overline{a}_h \Rightarrow \langle A_{hj}\rangle_{j=1}^n \vdash ; t_{a_h} \vdash ; w_{a_h}\ \}_{h=1}^m \\ \{\ \{\!\!\{A_{1j}/y_1,\dots,A_{mj}/y_m\}\!\!\} \vdash_F g^k(y_1,\dots,y_m) \Rightarrow C_j \vdash ; t_c^j \vdash ; w_c^j\ \}_{j=1}^n\end{array}}{\begin{array}{r}\sigma \vdash_F g^{k+1}(\overline{a}_1,\dots,\overline{a}_m) \Rightarrow \langle C_j\rangle_{j=1}^n \vdash ;\big(\sum_{h=1}^m t_{a_h}\big) + \big(\max_{j=1}^n t_c^j\big) \\ \vdash ;\big(\sum_{h=1}^m w_{a_h}\big) + \big(\sum_{j=1}^n w_c^j\big)\end{array}}$	$A_{hj} \neq \mathsf{err}$
$\mathit{Iterator})$	$\dfrac{\begin{array}{c}\{\ \sigma \vdash_F \overline{a}_h \Rightarrow \langle A_{hj}\rangle_{j=1}^n \vdash ; t_{a_h} \vdash ; w_{a_h}\ \}_{h=1}^m \\ \{\ \sigma\{\!\!\{A_{1j}/y_1,\dots,A_{mj}/y_m\}\!\!\} \vdash_F c \Rightarrow C_j \vdash ; t_c^j \vdash ; w_c^j\ \}_{j=1}^n\end{array}}{\begin{array}{r}\sigma \vdash_F [y_1 \leftarrow \overline{a}_1,\dots,y_m \leftarrow \overline{a}_m:\ c] \Rightarrow \langle C_j\rangle_{j=1}^n \vdash ;\big(\sum_{h=1}^m t_{a_h}\big) + \big(\max_{j=1}^n t_c^j\big) \\ \vdash ;\big(\sum_{h=1}^m w_{a_h}\big) + \big(\sum_{j=1}^n w_c^j\big)\end{array}}$	$A_{hj} \neq \mathsf{err}$
$\mathit{Error})$	rules for *Val*, *Let*, *Cond*, *PrimOp*, *UserFun* and *Iterator* if $A_{\_\_} = \mathsf{err}$, and for PrimOp^{k+1}, $\mathit{UserFun}^{k+1}$ and *Iterator* if A_i have differing lengths.	

Table 2: Ideal dynamic semantics.

variable names to non-erroneous values: $\mathit{Var} \rightarrow \mathit{Val} \setminus \{\mathsf{err}\}$. We write $\{\!\!\{ A_1/x_1, \ldots, A_n/x_n \}\!\!\}$ for the environment in which each variable x_i is mapped to the value A_i. We use the same notation for the pointwise perturbation function: $\sigma\{\!\!\{ C/y \}\!\!\}$ is equivalent to σ at all points, with the possible exception of y.

We give a big-step or "natural" operational semantics [15] in Section 2. The inference rules define a 6-ary relation

$$\sigma \vdash_F a \Rightarrow A\,;\, t\,;\, w$$

which should be read "when expression a is evaluated in environment σ using function declaration F, it yields value A; further, the process of determining the value takes $O(t)$ steps and $O(w)$ work." We use the big-O notation informally to communicate that we are not interested in constant factors; indeed, the AE preorder is insensitive to such constants. Thus we have *not* needed to include implementation-specific constant factors in our semantics, as would have been necessary had we used an efficiency preorder.

Our semantics does not include the costs of input and output. Nor does it calculate the cost of evaluating erroneous expressions. We adopt the convention that the cost of an erroneous computation is infinite; the semantics has the property that if $\sigma \vdash_F a \Rightarrow \mathsf{err}; t; w$ then $t = w = \infty$.

We now describe the rules in Section 2. Variables are dereferenced using the current environment, with constant access time. Values yield themselves, taking steps proportional to their depth and work proportional to their size. The rules for let and the conditional are straightforward. In reading them, it may help to think that the metavariables to the left of the $\Rightarrow$ are "bound" below the line; whereas those to the right are "bound" above.

The remaining rules look complicated, but have much common structure. The rules for basic function and primitive calls are very similar, as are those for lifted calls and iterators.

The semantics assumes that there exist implementations of the primitive operations; these implementations are described by a family of functions δ_p—one for each primitive p. Intuitively, $\delta_p(A_1, \ldots, A_\ell) = D\,;\, t_d\,;\, w_d$ if p takes t_d steps and w_d work to produce the result D on input $(A_1, \ldots, A_\ell)$. Section 3.3 discusses δ_p in more detail.

A basic primitive call p^0 is performed by evaluating the arguments and using the resulting values as input to δ_p. For a basic function call a new environment is created, binding the actual- to the formal-parameters; the function body is then evaluated in this environment. Note that non-local variables are prohibited inside a function body.

A lifted function call $g^{k+1}(\overline{a}_1, \ldots, \overline{a}_m)$ specifies the parallel application of g^k to the sequences $\overline{A}_h$. The semantics is given by induction on k. The arguments must evaluate to equal length sequences $\overline{A}_h = \langle A_{hj} \rangle_{j=1}^{n}$ for some n. (If no such n exists, then the result is err.) Chose a set of m distinct variable names, $y_1, \ldots, y_m$. For each j, a new environment is built that maps A_{hj} to y_h, and $g^k(y_1, \ldots, y_m)$ is evaluated in this environment. The results are then combined. Because we want to ensure parallel execution of each g^k, we take the step complexity to be the maximum of each. Note that the use of variables here is a "trick" to get the right complexities; the simpler solution—to recurse on $g^k(A_{1j}, \ldots, A_{mj})$—overcharges.

Iterator expressions are evaluated much like lifted function calls. The differences are that 1. an arbitrary expression c is evaluated in the subcalls (rather than g^k), and 2. the ambient environment σ is not lost in the subcalls—"non-local" references are possible. In an iterator expression, a *non-local* reference is an occurrence of any variable other than one bound by the iterator. It is the possibility of non-local reference within an iterator expression that forces us to abandon this ideal semantics in Section 5.

The semantics of Section 2 is *deterministic*: if a program has a derivation under environment σ, then it has only one. It is also *partial*: non-terminating programs have no finite derivation under σ. For clarity, we extend the semantics to a total function on programs by adding a bottom element to the set of values and mapping infinite computations there. We define a "nearly-flat" domain $\mathit{Val}_\perp$ with ordering:

$$A \sqsubseteq B \text{ iff } A = \perp \vee (A = \mathsf{err} \,\&\, B \neq \perp)$$

The signature and definition of the semantic function are:

$$[\![\cdot]\!] : \mathit{Prog} \rightarrow \mathit{Env} \rightarrow \mathit{Val}_\perp \times \mathbb{N}^\infty \times \mathbb{N}^\infty$$

$$[\![F, a]\!]\sigma = \begin{cases} A\,;\, t_a\,;\, w_a & \text{, if } \sigma \vdash_F a \Rightarrow A\,;\, t_a\,;\, w_a \\ \perp\,;\, \infty\,;\, \infty & \text{, otherwise} \end{cases}$$

From $[\![\cdot]\!]$, we derive functions $[\![\cdot]\!]^{\mathcal{E}}$, $[\![\cdot]\!]^{\mathcal{T}}$ and $[\![\cdot]\!]^{\mathcal{W}}$ which return, respectively, the extensional meaning, the step complexity and the work complexity of a program. We sometimes drop the function declaration when it is clear from context, writing these as a functions on expressions rather than programs.

3.2 The asymptotic efficiency preorder

We use the big-O/Ω notation with functions on values and environments. For example, let f, f' map environments to $\mathbb{N}^\infty$. Then we say that $f' = O(f)$ iff there exist r and u such that for all σ, $f'\sigma \leq r * (f\sigma) + u$. We now define the AE-preorder ($\precsim$).

Definition 1. $P \precsim P'$ iff there exist real numbers r, s, u, and v, such that for every environment σ, if $[\![P]\!]\sigma = A;t;w$

and $[\![P']\!]\sigma = \textsc{a}' \,;\, t' \,;\, w'$ then:

$$\textsc{a} \sqsubseteq \textsc{a}' \quad \& \quad t' \leq r * t + u \quad \& \quad w' \leq s * w + v$$

We write $\approx$ for the induced equivalence.

For $\mathbb{P} \subseteq \mathit{Prog}$, we define $e \precsim_{\mathbb{P}} e'$ iff for every function declaration F such that $(F, e) \in \mathbb{P}$:

$$(F, e') \in \mathbb{P} \quad \& \quad (F, e) \precsim (F, e')$$

We drop the superscript when $\mathbb{P} = \mathit{Prog}$. $\square$

An important fact about these relations is that they are indeed *preorders* (that is, reflexive and transitive). In addition, they are *substitutive* for our language (and for the classes of programs $\mathbb{P}$ that we consider): Define a *program context* $P[]$ to be a program in which one expression has been removed and replaced with a *hole*, written "[]"; $P[a]$ is then the program in which a is substituted into the hole. We then have that if $a \precsim b$ then $P[a] \precsim P[b]$ for all $P[]$.

3.3 Primitive specifications and language implementations

The semantics of the language is parametric with respect to a *primitive specification*: a set $\{\delta_p \mid p \in \mathit{PrimOp}\}$ of functions, one for each of the primitive functions. δ_p gives the (asymptotic) intensional meaning of the p. For a primitive p of arity ℓ, δ_p is a total function with the following signature:

$$\delta_p : (\textstyle\prod_\ell \mathit{Val}) \to \mathit{Val} \times \mathbb{N}^\infty \times \mathbb{N}^\infty$$

As we did for $[\![\cdot]\!]$, we derive functions $\delta_p^{\mathcal{E}}$, $\delta_p^{\mathcal{T}}$ and $\delta_p^{\mathcal{W}}$ from δ_p. We call this a primitive *specification* because it constrains the possible implementations of the primitives. A primitive specification is more concrete than an *abstract datatype*, which is purely extensional. We require that all primitives in the specification be strict in err and have infinite step and work complexities on err.

An implementation of the language need not follow the structure of the operational semantics. We only require that the implementation of each program be AE-greater than its definition in the semantics. In particular, we do not expect implementations to support iterators or lifted function calls.

The implementations discussed in this paper assume that the VRAM supports basic and 1-lifted calls for every primitive. $(k{+}1)$-lifted calls can be implemented in terms of 1-lifted calls via the following equation:

$$g^{k+1}(a_1, \ldots,\ a_\ell) \precsim \mathsf{insrt}_k(a_1, g^1(\mathsf{extr}_k\, a_1, \ldots, \mathsf{extr}_k\, a_\ell))$$

For the implementations we discuss, this equation is true for any primitive or user-defined function. This allows the implementation to create only basic and 1-lifted versions of each user-defined function.

The costs that can be reported in a primitive specification are constrained by the requirement that lifted primitive calls cost no more than allowed by the semantics: the costs reported for primitive calls must be "scalable". This fact makes it impossible for our semantics to accurately capture the costs of some implementations. For example, in Section 5 we describe an implementation in which extr^0 is more efficient than extr^1; however, the tighter bounds for extr^0 cannot be reported in the primitive specification, because this would cause the semantics to *underestimate* the cost of lifted calls to extr^1, making the semantics unsound.

4 The ideal semantics and iterator removal

A CREW VRAM can support a reference-based implementation of sequences. As noted in Section 3.3, however, we do not expect the VRAM models to support iterators—or any higher-order construct—directly. Thus the main implementation problem is *iterator removal*.

In Section 3, we present a simple set of conditional rewrite rules, or program transformations, which can be used to eliminate iterators from a program. These transformations can be applied to any subterm of a program, forming a rewrite system [11]. The transformations are terminating but non-confluent (CONST conflicts with the other rules.)

The main theorems of this paper state—for each semantics—the class of programs for which this rewrite system preserves the AE-preorder. The theorems are stated for a single step $a \rightsquigarrow b$; soundness of iterator removal follows because the AE-preorder is transitive and substitutive and because the transformations terminate.

For the ideal semantics, let us attempt to prove that $a \rightsquigarrow b$ implies $a \precsim b$. We do so by case analysis on the transforms and find that the cases for CONST and COND will not go through without further assumptions.

First, we must constrain the complexities of the primitives that are introduced by the transformations: all of these must have constant step complexity; len and not must have constant work complexity; $\mathsf{dist}(n, \textsc{a})$ must have work proportional to n; and comb and rstr must have work proportional to the length of their first arguments. Note that these restrictions point to a reference-based implementation.

More important, the COND transformation severely limits the programs that can be considered. In the source expression, instances of b and c are evaluated in parallel, whereas in the target the b's are evaluated separately from the c's. This is not a problem for the work complexities, since the same b's and c's are evaluated and the combining function (sum) doesn't

ID	$[x_1 \leftarrow \overline{e}_1, ..., x_h \leftarrow \overline{e}_h, ..., x_m \leftarrow \overline{e}_m :\ x_h]$ $\rightsquigarrow$ $\overline{e}_h$
CONST	$[x_h \leftarrow \overline{e}_h :\ a]$ $\rightsquigarrow$ $\boxed{\text{if } \forall h :\ a \text{ independent-of } x_h}$ $\mathsf{if}\ \overline{e}_1 = \langle\rangle\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ \mathsf{dist}\,(\mathsf{len}\overline{e}_1, a)$
APP	$[x_h \leftarrow \overline{e}_h :\ g^k(a_1, ..., a_\ell)]$ $\rightsquigarrow$ $g^{k+1}\big([x_h \leftarrow \overline{e}_h :\ a_1], ..., [x_h \leftarrow \overline{e}_h :\ a_\ell]\big)$ for $g \in \mathit{PrimOp} \cup \mathit{UserFun}$.
LET	$[x_h \leftarrow \overline{e}_h :\ \mathsf{let}\ y = a\ \mathsf{in}\ c]$ $\rightsquigarrow$ $[x_h \leftarrow \overline{e}_h, y \leftarrow [x_h \leftarrow \overline{e}_h :\ a] :\ c]$
COND	$[x_h \leftarrow \overline{e}_h :\ \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c]$ $\rightsquigarrow$ $\mathsf{comb}\,\big(\overline{a}, [x_h \leftarrow \mathsf{rstr}(\overline{a}, \overline{e}_h) :\ b], [x_h \leftarrow \mathsf{rstr}(\mathsf{not}^1\overline{a}, \overline{e}_h) :\ c]\big)$ where $\overline{a} \overset{def}{=} [x_h \leftarrow \overline{e}_h :\ a]$.

Table 3: Program transformations.

change via transformation. But it *is* a problem for the step complexity, where the combining function *does* change: from max to sum. Blelloch originally noticed this problem and defined a class of programs for which this transformation was sound: *contained* programs [2]. Intuitively, contained programs are those that do not have recursive calls on both sides of a conditional.

Definition 2. A program is *contained* if every conditional expression has one branch whose step complexity is bounded. The step complexity of an expression e is *bounded* if $[\![e]\!]^{\mathcal{T}} = O(1)$. □

Theorem 3. *Assume the ideal semantics and a primitive specification that meets the constraints outlined in this section. Let $\mathbb{C}$ be the class of contained programs. Then $a \rightsquigarrow b$ implies $a \sqsubseteq\!\!\!\!\!\!\sim_{\mathbb{C}} b$.* □

To see that the ideal semantics forces a reference-based implementation of sequences, consider the expression $[\mathsf{x} \in \langle 1..\mathsf{n}\rangle :\ \mathsf{y}]$. Referring to the semantics, an evaluation of this expression must create a sequence of n copies of y, and must do so with a constant number of steps and work proportional to n. This implies that the *representation* of the value $\sigma\,\mathsf{y}$ must have a constant size, regardless of size of the value itself.

5 The construct-parameters semantics

Unfortunately, the ideal semantics is not implementable on an EREW or bounded-contention CREW VRAM. The constant cost of variable dereferencing is the source of the problem. An obvious solution is to modify the semantics for variable dereferencing to "charge" for the size of the value returned; however, the semantics must then charge for the full size of a variable each time it is passed to a function, getting asymptotically wrong complexities for recursive functions. Another solution would be to charge for input, or—thinking compositionally—for the size of the evaluation environment σ; however, this turns out to be equivalent to charging for variable references. A restricted version of this idea is useful, however, and will be developed below.

The central idea of the *construct-parameters* semantics is that free variable references inside an iterator should not be *free*: the semantics should charge for copying them out. This is the semantics adopted by NESL. The new semantics changes the rule for iterators in Section 2. For the iterator $[y_1 \leftarrow \overline{a}_1, ..., y_m \leftarrow \overline{a}_m :\ c]$ let X be the set $(\mathsf{free}\ c) \setminus \{y_1, ..., y_m\}$. Then the work complexity for an iterator in Section 2 is changed to:

$$\left(\textstyle\sum_{h=1}^{m} w_{a_h}\right) + \left(\sum_{j=1}^{n} w_c^j\right) + \left(n * \sum_{x \in X} \mathcal{S}(\sigma x)\right)$$

Likewise, the step complexity is changed to:

$$\left(\textstyle\sum_{h=1}^{m} t_{a_h}\right) + \left(\max_{j=1}^{n} t_c^j\right) + \left(\sum_{x \in X} \mathcal{D}(\sigma x)\right)$$

Given this semantics (and the new AE-preorder derived from it), what constraints must we place on programs and primitive specifications for the program transformations to be sound? Not surprisingly, programs must still be contained, but the complexity bounds on the primitives are relaxed considerably.

In the case of the primitives introduced by the transforms, all but dist must have work complexities bounded by the sum of the *sizes* of their arguments (not lengths) and step complexities bounded by the max of the *depths* of their arguments (not constant). The work complexity of $\mathsf{dist}(n, A)$ must be bounded by $n * (\mathcal{S}A)$.

The soundness proof also requires that the size of the result of an expression be bounded by the size of

ELT-BAR

$$\left[x_h \leftarrow \overline{e}_h :\ \underline{\mathsf{elt}}^k_\ell\,(\overline{a}, c_1, \ldots, c_\ell)\right]$$

$$\rightsquigarrow \boxed{\text{if } \forall h:\ \overline{a} \text{ independent-of } x_h}$$

$$\underline{\mathsf{elt}}^{k+1}_\ell\left(\overline{a}, [x_h \leftarrow \overline{e}_h :\ c_1], \ldots, [x_h \leftarrow \overline{e}_h :\ c_\ell]\right)$$

$$\rightsquigarrow \boxed{\text{if } \exists h:\ \overline{a} \text{ dependent-on } x_h}$$

$$\begin{aligned}\underline{\mathsf{elt}}^{k+1}_{\ell+1}\big(&[x_h \leftarrow \overline{e}_h :\ \overline{a}],\\ &\mathsf{prom}^1_k\left([x_h \leftarrow \overline{e}_h :\ c_1], \mathsf{dom}\ \overline{e}_1\right),\\ &[x_h \leftarrow \overline{e}_h :\ c_1], \ldots, [x_h \leftarrow \overline{e}_h :\ c_\ell]\big)\end{aligned}$$

Table 4: Transformation rules for $\underline{\mathsf{elt}}$.

free variables in that expression plus the work done to compute the result. This is true as long as all of the primitives of the language satisfy this constraint. We say that a primitive p is *non-magical* iff:

$$\begin{aligned}\mathcal{S}\,\delta^{\mathcal{E}}_p(A_1, \ldots, A_\ell) &= O\left(\left(\textstyle\sum_i \mathcal{S}A_i\right) + \delta^{\mathcal{W}}_p(A_1, \ldots, A_\ell)\right)\\ \mathcal{D}\,\delta^{\mathcal{E}}_p(A_1, \ldots, A_\ell) &= O\left((\max_i \mathcal{D}A_i) + \delta^{\mathcal{T}}_p(A_1, \ldots, A_\ell)\right)\end{aligned}$$

The primitives that we use are implemented by the Data-Parallel Library (DPL) [9], an extension of the C Vector Library [4]. The implementation is non-magical and meets the other constraints given above.

Theorem 4. *Assume the construct-parameters semantics and a primitive specification that meets the constraints outlined in this section. Then $a \rightsquigarrow b$ implies $a \precsim_{\mathbb{C}} b$.* $\square$

6 The construct-result semantics

The *construct-result* semantics takes a complimentary view of the problem, charging for "output" rather than "input". This gives a natural cost calculus to programmers—similar to the ideal semantics—but greatly complicates implementation. As we shall see, there is also a further loss of generality.

In the construct-result semantics, the work and step complexities for iterators are given respectively by:

$$\left(\textstyle\sum_{h=1}^m w_{a_h}\right) + \left(\textstyle\sum_{j=1}^n\ w^j_c + \mathcal{S}c_j\right)$$
$$\left(\textstyle\sum_{h=1}^m t_{a_h}\right) + \left(\max_{j=1}^n\ t^j_c + \mathcal{D}c_j\right)$$

Using this semantics and derived AE-preorder, soundness of the APP transformation imposes a much more rigid constraint on primitives than it did in the construct-parameters semantics. Each primitive p must *fully use its parameters*, that is, must satisfy the following equations:

$$\begin{aligned}\textstyle\sum_i \mathcal{S}A_i &= O\left(\mathcal{S}\,\delta^{\mathcal{E}}_p(A_1, \ldots, A_\ell) + \delta^{\mathcal{W}}_p(A_1, \ldots, A_\ell)\right)\\ \max_i \mathcal{D}A_i &= O\left(\mathcal{D}\,\delta^{\mathcal{E}}_p(A_1, \ldots, A_\ell) + \delta^{\mathcal{T}}_p(A_1, \ldots, A_\ell)\right)\end{aligned}$$

All but five of the primitives in DPL meet this criterion. We divide these in two groups: elt and rstr, and len, insrt and prom.

One can easily see that the APP rule is unsound for $\mathsf{elt}_\ell(\overline{a}, c_1, \ldots, c_\ell)$ if $\overline{a}$ is independent of the iterator variables: in the target, the semantics requires that we pay for constructing $[x_h \leftarrow \overline{e}_h :\ \overline{a}]$, which could be enormous. Palmer *et al.* [10] propose a solution to this problem which avoids "pushing the iterator" around $\overline{a}$. They replace occurrences of elt with the new function $\underline{\mathsf{elt}}$ which has a non-standard semantics for lifted applications: $\underline{\mathsf{elt}}^k(\overline{a}, c_1, \ldots, c_\ell) \overset{def}{=} \mathsf{elt}^k(\mathsf{dist}(\#c_1, \overline{a}), c_1, \ldots, c_\ell)$. Correspondingly, they modify the transformation rule APP to include the special cases in Section 4; this modification of APP ensures that the transformation is sound (relative to the AE-preorder) for $\underline{\mathsf{elt}}$. With high probability, the implementation of $\underline{\mathsf{elt}}$ in DPL incurs little contention on an CREW VRAM.

The APP transformation is unsound for $\mathsf{rstr}^k(b, a)$ only when a evaluates to a value of depth greater than k; in this case, the problem is reminiscent of that for elt and can be handled by writing rstr in terms of $\underline{\mathsf{elt}}$.

The functions len, insrt and prom are a bit more difficult. Consider the example $[\mathsf{x} \leftarrow \overline{\mathsf{x}} :\ \mathsf{len}\,\mathsf{x}]$, which has work $\mathcal{L}_1(\sigma\,\overline{\mathsf{x}})$. After transformation this becomes $\mathsf{len}^1\,[\mathsf{x} \leftarrow \overline{\mathsf{x}} :\ \mathsf{x}]$, with the iterator subexpression having work $\mathcal{S}(\sigma\,\overline{\mathsf{x}})$ due the construct-result rule. Our solution to the problem is to ban this expression and others like it where the argument of len is dependent on the iterator variable but is neither *fully constructed* nor *fully-used* within the iterator.

Definition 5. An expression e is *fully constructed* iff: $[\![e]\!]^{\mathcal{W}} = \Omega(\mathcal{S}\,[\![e]\!]^{\mathcal{E}})$. An expression context $E[]$ is similar to a program context, defined in section Section 3.2. We say that $E[]$ is *fully used* iff for every expression e: $[\![E[e]]\!]^{\mathcal{W}} = \Omega(\mathcal{S}\,[\![e]\!]^{\mathcal{E}})$,

A program is *fully constructed* iff for every occurrence len e inside an iterator such that e depends on an iterator variable: either e is fully constructed or e appears in a context $E[e]$ within the same iterator and $E[]$ is fully used. In addition, we require the same for occurrences of insrt and prom, regardless of whether the expression depends on an iterator variable. $\square$

Occurrences of len e in which e is independent of all iterator variables can be handled by using the CONST rule; a wise thing to do in any case.

The requirement that programs be fully constructed does not significantly limit the expressiveness of the language. After all, the length of a sequence is usually of interest only when that sequence is otherwise used in an expression. The restrictions are still less onerous for insrt and prom: these functions are designed specifically to support our program transformations and are of little independent interest.

Finally, we address the issue of user-defined functions, which inherit problems from the primitives used to define them. Within the limits of static analysis, it is possible to detect parameters that are never used outside of calls to elt or rstr and to treat these parameters as we did those of elt; however, we have yet to fully explore this approach. Here, instead, we present a simple-minded solution that treats the five troublesome primitives uniformly, drastically reducing the expressiveness of the language. We require that each user-defined function *fully use its parameters* (defined in analogy to full-use for primitives).

Theorem 6. *Assume the construct-result semantics and a primitive specification that meets the constraints outlined in this section. Let $\mathbb{F}$ be the class of contained and fully-constructed programs in which each user-defined function fully uses its parameters. In addition, modify $\rightsquigarrow$ to ensure that* CONST *is applied whenever possible. Then $a \rightsquigarrow b$ implies $a \precsim_{\mathbb{F}} b$.* □

7 Conclusions

This study has clarified many issues which arise in the implementation of a "flattening" compiler for a nested-parallel programming language. In addition to the treatment of free variables, our work has suggested techniques for dealing with constant factors in the implementation, and for integrating flattening with other forms of program optimization. Our proof technique—using a profiling semantics, AE-equivalence, and program transformations—appears to be novel and may be independent interest.

There are several directions for further work. On the theoretical side, we have extended our results to a language with a fuller type system, including products, sums, and higher-order functions. More pragmatically, we hope to develop full implementations of both our ideal and construct-result semantics and conduct some performance studies. Most important here is the development of static analyses to lessen the restrictions on function declarations imposed in Theorem 6. Finally, we have embarked on an effort to improve the expressiveness of the language and to reduce some of the constant factors in the implementation.

Acknowledgments

Our thanks go to Dan Palmer, Lars Nyland and Rik Faith for their work developing and implementing the techniques discussed here, and to Guy Blelloch for discussing with us the subtleties of his NESL implementation.

Keywords

Data-parallelism, nested-parallelism, vectorization, flattening, NESL, work/step complexities, VRAM, profiling semantics, efficiency preorder.

References

[1] S. Arun-Kumar and Matthew Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29:737–760, 1992.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[3] Guy E. Blelloch. NESL: A nested data-parallel language (version 3.0). Technical report, Carnegie-Mellon University, Department of Computer Science, 1994.

[4] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, Carnegie-Mellon University, Department of Computer Science, February 1993.

[5] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Marco 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.

[6] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 226–237, June 1995.

[7] Allen Goldberg, Peter Mills, Lars Nyland, Jan Prins, John Reif, and James Riely. Specification and development of parallel algorithms with the Proteus system. In G.E. Blelloch, K.M. Chandy, and S.Jagannathan, editors, *Specification of Parallel Algorithms*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS Press, May 1994.

[8] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[9] Daniel W. Palmer. DPL: Data Parallel Library manual. Technical Report 93:064, University of North Carolina, Department of Computer Science, November 1993.

[10] Daniel W. Palmer, Jan F. Prins, and Stephen Westfold. Work-efficient nested data-parallelism. In *Frontiers '95*, 1995.

[11] David A. Plaisted. Term-rewriting systems. In Dov M. Gabbay, Christopher John Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2 Deduction Methodologies. Oxford University Press, 1993.

[12] Jan F. Prins and Daniel 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), July, 1993).

[13] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming, 1994. Queens University Department of Computer Science TR-93-348.

[14] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[15] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.