Shapes and Flattening

John Reppy
Computer Science
University of Chicago
Chicago, IL, USA
jhr@cs.uchicago.edu

Joe Wingerter
Computer Science
University of Chicago
Chicago, IL, USA
wings@cs.uchicago.edu

ABSTRACT

NESL is a first-order functional language with an apply-to-each construct and other parallel primitives that enable the expression of irregular nested data-parallel (NDP) algorithms. To compile NESL, Blelloch and others developed a global flattening transformation that maps irregular NDP code into regular flat data parallel (FDP) code suitable for executing on SIMD or SIMT architectures, such as GPUs.

While flattening solves the problem of mapping irregular parallelism into a regular model, it requires significant additional optimizations to produce performant code. Nessie is a compiler for NESL that generates CUDA code for running on Nvidia GPUs. The Nessie compiler relies on a fairly complicated *shape analysis* that is performed on the FDP code produced by the flattening transformation. Shape analysis plays a key rôle in the compiler as it is the enabler of fusion optimizations, smart kernel scheduling, and other optimizations.

In this paper, we present a new approach to the shape analysis problem for NESL that is both simpler to implement and provides better quality shape information. The key idea is to analyze the NDP representation of the program and then preserve shape information through the flattening transformation.

ACM Reference Format:
John Reppy and Joe Wingerter. 2020. Shapes and Flattening. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

Nessie [17, 20] is a compiler for the nested-data-parallel (NDP) language NESL [4, 5] that targets GPUs. The NESL language was originally designed by Guy Blelloch as a way to program irregular parallel algorithms on the wide-vector and SIMD architectures of the early to mid 1990s. The enabling technology for NESL was a global flattening transformation developed by Blelloch and others that maps irregular NDP code into regular flat data-parallel (FDP) code suitable for SIMD execution [7, 9, 12, 16].

The Nessie compiler follows the same approach for compiling NESL to GPUs. The front half of the Nessie compiler does the usual parsing and type checking, followed by monomorphization (NESL has parametric polymorphism). We then apply a modified form of Keller's flattening transformation [9] to produce a flat-data-parallel (FDP) representation of the program. In the flat representation, the only sequences are flat sequences of base types (`bool`, `float`, *etc.*). The original nested-sequence structure is captured in associated *segment descriptors*. The back-end of our compiler converts the FDP representation to CUDA code by first applying a flow-based shape analysis to determine the size and structure of sequences [20], then converting the FDP form to our λ_{cu} [18, 25] representation, optimizing the λ_{cu} representation, and finally generating code from it. The shape analysis is key to enabling optimizations: it identifies sequences of the same size, which enables the compiler to find more opportunities for fusion, memory reuse, and other optimizations.

Examining the results of shape analysis, however, one observes that the information recovered by the analysis is present in the original NDP representation and can be more easily detected before flattening. Furthermore, there is information that is trivially determined by the NDP representation that is more difficult to extract from the flattened program. For example, a sequence of pairs in the NDP representation will be transformed to a pair of sequences in the FDP representation;[1] in such a case, it may be difficult for a conservative analysis to recognize that the two sequences must have the same length.

This paper describes a new approach to mapping from a monomorphic NDP IR to a FDP IR annotated with shape information. Instead of applying shape analysis after flattening, we instead first analyze the NDP IR and record the shape information by annotating the types of the NDP program. We then apply a flattening transformation that transcribes the shape information from the NDP types into the type of the segment descriptors, such that if two segment descriptors have the same type (*i.e.*, shape), then they are equal.

The fundamental problem with shape analysis on the post-flattening FDP IR is that the flattening process disassociates related structures. The example of sequences of pairs has already been mentioned, but there are other examples, such as the fact that segment descriptors and the sequences that they describe are not directly connected. Thus the analysis has to work hard to identify which descriptors are associated with which data sequences. By performing the shape analysis prior to flattening, we do not have to rely on analysis to make those connections; instead, they are readily available in the structure of the NDP IR. The thesis of this work is that shape analysis on the NDP IR is inherently simpler and more precise than our previous flow-based analysis on the FDP IR. Furthermore, our new annotation language for shape information allows us to keep track of more precise shape information than before. We also

IFL'19, September 2019, Singapore

[1] Flattening converts sequences of pairs to pairs of sequences (the so-called "AoS to SoA" transformation).

present a one-pass flattening algorithm that propagates the shape information into the FDP representation.

The rest of the paper is organized as follows. We first provide some background about the Nesl language, the flattening transformation, the need for shape information, and the problem with analysis after flattening. In Section 3, we present a representation of types with shape annotations. We then describe how we analyze an NDP kernel language to produce a representation with shape information. In Section 5, we describe the flattening transformation that produces the FDP representation, where the segment descriptors are given types that reflect the shape of the sequences that they describe. We then describe the current status of our work and discuss future plans; finally we present concluding remarks in Section 8.

2 BACKGROUND

In this section, we describe the context in which we are operating, including a quick introduction to Nesl, a brief description of the flattening transformation, and motivation for why shape information is important.

2.1 Nesl

Nesl is a first-order data-parallel functional language that supports data parallelism in two ways: through a parallel sequence comprehension (*apply-to-each*) and through a set of parallel primitive operators. Apply-to-each allows the programmer to map an arbitrary computation over a sequence. For example, the following function squares each element of the sequence `xs`:

```
function sqr (xs) : [float] -> [float] =
  { x * x : x in xs };
```

We have annotated this function with its type (sequence of float to sequence of float), but Nesl supports Hindley-Milner polymorphic type inference,[2] we omit the types from subsequent examples.

It is also possible to map a computation over multiple sequences of the same length, as in this example that computes the element-wise product of `xs` and `ys`.

```
function prod (xs, ys) =
  { x * y : x in xs; y in ys };
```

Notice that Nesl uses *zip* semantics for iteration over multiple sequences. To compute the outer product of these two sequences, we nest the iteration over `ys` inside the iteration over `xs`.

```
function outer (xs, ys) =
  { { x * y : y in ys } : x in xs };
```

An apply-to-each may include an optional predicate to specify which elements to apply the computation to; for example, we could compute the product of the positive elements of `xs` and `ys` as follows:

```
function prod_if_pos (xs, ys) =
  { x * y : x in xs; y in ys
  | x >= 0 and y >= 0 };
```

[2]Nesl, in fact, extends the standard type inference mechanism with a pre-defined set of Haskell-like type classes, so it would have inferred the more general type "`[a] -> [a]:: (a in num)`" for the `sqr` function.

In this case, the size of the resulting sequence is unknown at compile time.

As seen in the `outer` example above, the computations that are mapped by an apply-to-each may themselves be parallel computations; thus the term *nested-data parallelism* is used for this programming pattern. The NDP model is well-suited to matrix operations; for example, we can use nested parallelism to multiply a matrix by a sequence:

```
function dotp (xs, ys) =
  sum({ x * y : x in xs; y in ys });

function mxv (m, v) =
  { dotp(row, v) : row in m };
```

The outer apply-to-each applies `dotp` in parallel to each row of the matrix. Within each call to `dotp`, the elements of the row are all multiplied in parallel, then added up with the parallel `sum` operation.

While the previous example is a computation over a rectangular matrix, nested parallelism does not have to be regular in Nesl. For example, the computation of a sparse matrix (represented by rows of index-element pairs) times a dense sequence can be coded as

```
function sparse_mxv (sm, v) =
  { sum({ x * v[i] : (i, x) in sv })
  : sv in sm
  }
```

This function has the same nested structure as before, but the amount of work per row varies. The fact that nested parallel computations can be arbitrary parallel computations, gives Nesl a great deal of expressiveness that can be used to implement a wide variety of irregular parallel computations [21].

In addition to the parallel apply-to-each construct, Nesl has a library of parallel sequence operations. These include reductions, such as the `sum` function above, prefix-scans, permutations, and various other operations on sequences [4].

2.2 NKL

We present our shape analysis and flattening techniques using a nested-data-parallel kernel language called NKL.[3] This language, whose abstract syntax is given in Figure 1, captures the important features of Nesl in a more concise syntax. A program in NKL is a sequence of function definitions terminated by an exported main function. The expression language is normalized (*i.e.*, all arguments are either variables or constants) to simplify the shape analysis and flattening transformation. The last three right-hand-side forms support parallel sequences.

Note that NKL's parallel map is more restrictive than Nesl's general apply-to-each construct, since it does not include the optional predicate expression. We can represent the Nesl expression

$$\{\ e\ :\ x_1\ \textbf{in}\ xs_1,\ \ldots,\ x_k\ \textbf{in}\ xs_k\ \mid\ e'\ \}$$

by the expression

$$\begin{array}{l}\textbf{let}\ flgs\ =\ \{\ e'\ :\ x_1\ \textbf{in}\ xs_1,\ \ldots,\ x_k\ \textbf{in}\ xs_k\ \};\\ \quad xs'_1\ =\ \texttt{filter}(xs_1,\ flgs);\\ \quad \ldots\end{array}$$

[3]This language is inspired by Keller's NKL [9].

p	::=	$fn\ p$	function definition
	|	f	main function name
fn	::=	**fun** $f\ (x_1, \ldots, x_k) : (\tau) \to \tau'$	function definition
		$= e$	
e	::=	v	value
	|	**if** v **then** e_1 **else** e_2	conditional
	|	$f\ (v_1, \ldots, v_k)$	function application
	|	**let** $x : \tau$ **=** r **in** e	let binding
r	::=	e	r.h.s. expression
	|	$p\ (v_1, \ldots, v_k)$	primop application
	|	$\langle v_1, v_2 \rangle$	pair
	|	$\{\ e : b_1, \ldots, b_k\ \}$	parallel map
	|	$[\ v_1, \ldots, v_k\ ]$	sequence
	|	$[\ \tau\ ]$	empty sequence
v	::=	x	variable
	|	n	constant
b	::=	x **in** xs	binding
τ	::=	$[\tau]$	sequence type
	|	$\tau_1 \times \tau_2$	pair type
	|	ι	
ι	::=	**int** | **bool** | **float** | $\cdots$	base types

Figure 1: Syntax of NKL

```
    xs'_k = filter(xs_k, flgs)
in
  { e : x_1 in xs'_1, ..., x_k in xs'_k }
```

where `filter` is a builtin function that takes two sequences of equal length and projects out the elements of the first that correspond to true entries in the second.

NKL programs are monomorphic and explicitly typed.[4] In the remainder of the paper, we omit the type annotations on function definitions and let-bound variables to reduce notational clutter. Instead, we use "typeof(f)" to denote the type of functions and "typeof(x)" to denote the type of variables when necessary.

2.3 Flattening

Flattening is a global program transformation that converts nested data parallelism into flat data parallelism [6, 7, 9]. A key aspect of this transformation is that nested sequences are transformed into *segmented sequences*, which are flat sequences of atomic data values paired with segment descriptors that define the original sequences's nesting structure. The flattening transformation also converts from "Array-of-Structs" (AoS) data layout to "Struct-of-Arrays" (SoA) layout. For example, the nested sequence:

```
[ [ (true, 1), (false, 2), (true, 3) ],
  [ ],
  [ (false, 4) ] ]
```

is represented by two data sequences

```
[ true, false, true, false ]
[ 1, 2, 3, 4 ]
```

and two segment descriptors that specify the lengths of the sequences at each level of nesting from inside out.

```
[ 3, 0, 1 ]
[ 3 ]
```

In practice, there are many different ways to represent segment descriptors that provide different benefits for different operations [1, 26]. For example, using an array of boolean segment-start flags allows segmented-scans to be defined using a flat scan operation [2].

In addition to affecting the data representation, the flattening transformation also affects the program structure. Code that is inside a parallel apply-to-each must be *lifted* to compute over sequences. For example, a function (or primitive operation) application

```
{ f(e_1, e_2) : x in xs }
```

is transformed to

```
if (empty xs)
  then []
  else let ys_1 = { e_1 : x in xs };
           ys_2 = { e_2 : x in xs }
    in
      f↑(ys_1, ys_2)
```

where $f^{\uparrow}$ is f lifted to work on sequences (*i.e.*, it is the parallel map of f over the zip of the argument sequences). The test for the empty input is the termination condition for recursion. To complete the above transformation, we recursively flatten the parallel maps of e_1 and e_2.

The flattening transformation uses bookkeeping operations to handle multiple levels of nested apply-to-each constructs (*i.e.*, we do not lift already lifted operations). Informally, these operations can be given the following types:

$\mathcal{S}$	:	$([[\tau]]) \to [\mathbf{int}]$	extract segment descriptor
$\mathcal{F}$	:	$([[\tau]]) \to [\tau]$	flatten one-level of nesting
$\mathcal{P}$	:	$([\tau], [\mathbf{int}]) \to [[\tau]]$	partition using descriptor

For example, the nested expression

```
{ { f(x, y) : x in xs, y in ys }
  : xs in xss, ys in yss
}
```

will be transformed to the code

```
let xs = F(xss);
    ys = F(yss);
    sd = S(xss)
in
  if (empty xs)
    then []
    else P (f↑ (xs, ys), sd)
```

[4] Recall that the Nessie compiler monomorphizes the program prior to flattening.

Because of the way that nested sequences are represented after flattening, these bookkeeping operations are constant-time.

Lastly, conditionals inside apply-to-each constructs are replaced with partitioning of the inputs. For example, we transform

```
{ if e then e1 else e2 : x in xs }
```

to

```
let flgs = { e : x in xs };
    (xs1, xs2) = SPLIT(xs, flgs);
    res1 = { e1 : x in xs1 };
    res2 = { e2 : x in xs2 }
in
  COMBINE(res1, res2, flgs)
```

Here the **SPLIT** and **COMBINE** operations are used to split and recombine sequences based on the sequence of booleans *flgs*.

2.4 The importance of shape information

Shape analysis is the process of statically identifying the nesting structure and sizes of sequences. Shape analysis plays a key rôle in the compiler as it is the enabler for a number of important optimizations, including

- Kernel fusion — Our compiler performs several kinds of fusion on parallel computations [17, 18, 25]. These include simple producer-consumer fusion; horizontal fusion of maps, scans, and reductions; and filter fusion [19]. The validity of these transformations requires that the compiler be able to reason about the sizes and structure of the argument sequences, which is information provided by the shape analysis.
- Memory reuse — reusing previously allocated memory for new results requires both lifetime analysis and information about the size of the memory object. The latter information is provided by shape analysis.
- Segment descriptor optimization — by attaching shape information to segment descriptors, we can identify when segment descriptors are redundant. We can also use this information to optimize their representation in some situations.
- Optimizing for rectangular structure — while, in general, nested sequences in Nesl are irregular, there are examples of programs that impose a regular rectangular structure on arrays (*e.g.*, matrix multiplication). Segmented computations on such nested sequences can take advantage of the rectangular structure to gain efficiency. Shape analysis can discover when rectangular structure is implied by the computation.

Currently, we support the first two of these optimizations, with plans for adding the other two.

The challenge for shape analysis after the flattening transformation is that information about the connections between different arrays may be obscured by flattening. For example, consider the following Nesl function:

```
function norm2 (xys) =
  let xs = { x : (x, y) in xys };
      ys = { y : (x, y) in xys };
      sum1 = sum(xs);
      sum2 = sum(ys);
  in
    ( { x / sum1 : x in xs },
      { y / sum2 : y in ys } )
```

that takes a sequence of pairs as an argument. Flattening will transform the sequence of pairs into a pair of sequences — the flattened version of this function will be roughly

```
function norm2 (xs, ys) =
  let sum1 = sum(xs);
      sum2 = sum(ys);
  in
    ( { x / sum1 : x in xs },
      { y / sum2 : y in ys } )
```

From this code, there is no way for shape analysis to determine that `xs` and `ys` have the same length, which prevents an opportunity for *horizontal* fusion of both the reductions and the maps. It is obvious before flattening, however, that the two sequences must have the same size, since they are both projected from `xys`.

3 SHAPES

Previous work on type systems for sequence shapes has focused on multidimensional arrays with rectangular shapes [10, 22–24]. In this work, we are interested in tracking information about the *irregular* shapes that arise in NDP programs. For example, consider the following Nesl program that adds two nested sequences:

```
function mm (xss, yss) =
  { { x + y : x in xs; y in ys }
  : xs in xss; ys in yss };
```

While the shapes of `xss` and `yss` may be irregular, they must be the same. The semantics of the Nesl apply-to-each construct means that we can infer that `xss` and `yss` have the same number of rows and, at the inner level, we can infer that each row of `xs` of `xss` has the same number of elements as the corresponding row `ys` of `yss`.

Consider a slightly different example that adds the sequence `ys` to each row of the sequence of sequences `xss`:

```
function mv (xss, ys) =
  { { x + y : x in xs; y in ys }
  : xs in xss };
```

In this case, we know that each row of `xss` must have the same length as `ys` and, thus we infer that `xss` must have rectangular shape.

The shape system that we describe below has been designed both to represent these kinds of structural properties and to provide a way to infer them via analysis.

3.1 Representing sizes

Because the Nesl type system does not have a notion of multidimensional arrays, we focus on describing the size (or dimension) of sequences. We assume a countable set of dimension variables (α, β, *etc.*), which represent unknown sizes, and dimension-function variables (ϕ, ψ, *etc.*), which represent unknown *dependent sizes*. The syntax of dimension expressions is given in Figure 2. In the syntax,

$$
\begin{array}{rcll}
d & ::= & n & \text{known size} \\
 & | & \alpha & \text{dimension variable} \\
 & | & \phi(\alpha) & \text{applied dimension function} \\
 & | & d_1 + d_2 & \text{dimension addition} \\
 & | & \sum_{\alpha=1}^{d} \phi(\alpha) & \text{summation} \\
v & ::= & d & \text{fixed dimension} \\
 & | & \phi & \text{dimension function}
\end{array}
$$

Figure 2: Sizes — fixed and variable dimensions

$$
\begin{array}{rcll}
\widehat{\tau} & ::= & [\widehat{\tau} \# v] & \text{sequence type} \\
 & | & \widehat{\tau_1} \times \widehat{\tau_2} & \text{pair type} \\
 & | & \widehat{\iota} & \text{base types} \\
\widehat{\iota} & ::= & \mathbf{int}(v) & \text{integer type} \\
 & | & \mathbf{bool} \mid \mathbf{float} \mid \cdots & \text{other base types}
\end{array}
$$

Figure 3: Annotated types

we distinguish between *fixed dimensions* d, which specify a single (possibly unknown) size of a sequence, and *varying dimensions* v, which are used to specify the size of nested arrays that may be irregular.

We use the term *size* to describe either a fixed or variable dimension, and *size variable* to describe both dimension and dimension-function variables. We use the term *shape* informally to refer to either the size of a single sequence or the size structure of nested sequences.

Size expressions are organized into three levels based on the quality of information that they provide.

$$
\begin{array}{rl}
\textit{Varying dimensions:} & \phi \\
\textit{Fixed unknown dimensions:} & \alpha, \phi(\alpha), d_1 + d_2, \sum_{\alpha=1}^{d} \phi(\alpha) \\
\textit{Fixed known dimensions:} & n
\end{array}
$$

Varying dimensions are the most general, then we have fixed dimensions of unknown size, and the most precise are known fixed dimensions.

3.2 Annotated types

We augment the NKL types (τ) from Section 2.2 with shape information to form *annotated types* ($\widehat{\tau}$), the syntax of which is given in Figure 3. We use sizes to annotate types in two ways:

(1) The type $[\widehat{\tau} \# v]$ is a sequence type annotated with the size v of the sequence. When the sequence is the outermost sequence in a type expression, v must be a fixed dimension d, but a nested sequences may have a varying dimension represented by a dimension function ϕ.
(2) The type $\mathbf{int}(v)$ represents integers, where v tracks the value of the integer. For sequences of integers, we use dimension functions to represent represent the fact that the value of the integer depends on the index of the element.

Here are some examples of annotated types:

$[[\mathbf{float} \# 5] \# 5]$	a 5×5 matrix
$[[\mathbf{float} \# \phi] \# 5]$	a 5-element sequence of sequences
$\mathbf{int}(17)$	the integer 17
$[\mathbf{int}(\phi) \# \alpha]$	a sequence of integers

For an annotated type $\widehat{\tau}$, we follow the convention of writing τ for the type with its annotations erased.

When we analyze the body of an apply-to-each, the elements may have types that are annotated with dimension variables; since such types are only well-formed when they are nested inside an outer sequence type, we have to *instantiate* non-nested shape functions. To address this need, we define the *instantiation* of a dimension with index α, written $v@\alpha$, as follows:

$$
\begin{array}{rcl}
d@\alpha & = & d \\
\phi@\alpha & = & \phi(\alpha)
\end{array}
$$

and then define the instantiation of an annotated type with index α as

$$
\begin{array}{rcl}
[\widehat{\tau} \# v]@\alpha & = & [\widehat{\tau} \# v@\alpha] \\
\widehat{\tau_1} \times \widehat{\tau_2}@\alpha & = & \widehat{\tau_1}@\alpha \times \widehat{\tau_2}@\alpha \\
\mathbf{int}(v)@\alpha & = & \mathbf{int}(v@\alpha) \\
\widehat{\tau}@\alpha & = & \widehat{\tau} \quad \text{otherwise}
\end{array}
$$

We use this operation in Section 4.3.3 when stripping off a sequence-type constructor to ensure that the resulting type is well formed.

3.3 Annotated function types

While functions in Nesl can be shape polymorphic (*e.g.*, the `length` function), the process of monomorphization fixes the ranks of sequence types. The resulting functions, however, many still be size polymorphic. Furthermore, the result shape of a function may not be determined by the shapes of its arguments. Therefore, we use the following general pattern for specifying the annotated type of a function:

$$\forall \vec{\alpha}, \vec{\phi}.(\widehat{\tau_1}, \ldots, \widehat{\tau_k}) \to \exists \vec{\beta}, \vec{\psi}.\widehat{\tau}$$

Returning to the two example functions (`mm` and `mv`) from the beginning of this section, we give them the following annotated types:

$$\texttt{mm} : \forall \alpha, \phi.([[\mathbf{float} \# \phi] \# \alpha], [[\mathbf{float} \# \phi] \# \alpha]) \to [[\mathbf{float} \# \phi] \# \alpha]$$

$$\texttt{mv} : \forall \alpha, \beta.([[\mathbf{float} \# \beta] \# \alpha], [\mathbf{float} \# \beta]) \to [[\mathbf{float} \# \beta] \# \alpha]$$

Notice that the type of `mm` captures the fact that the arguments and results must all have the same shape, albeit an irregular one. On the other hand, the type of `mv` shows that the first argument must have rectangular shape with the row dimensions equal to the second argument's size (*i.e.*, β).

There are two sources of uncertainty about result sizes: conditionals and builtin functions like `filter`. We discuss the types we assign to the builtin functions in Section 4.2.

4 SHAPE ANALYSIS

We present our shape analysis and lifting techniques using the NKL kernel language given in Figure 1.

$$
\begin{array}{rcl}
+_{\mathbf{int}} & : & \forall\alpha,\beta.(\mathbf{int}(\alpha),\mathbf{int}(\beta)) \rightarrow \mathbf{int}(\alpha+\beta) \\
*_{\mathbf{int}} & : & \forall\alpha,\beta.(\mathbf{int}(\alpha),\mathbf{int}(\beta)) \rightarrow \exists\gamma.\mathbf{int}(\gamma) \\
\texttt{length} & : & \forall\alpha.([\widehat{\tau}\ \#\ \alpha]) \rightarrow \mathbf{int}(\alpha) \\
\texttt{lengths} & : & \forall\alpha,\phi.([[\widehat{\tau}\ \#\ \phi]\ \#\ \alpha]) \rightarrow [\mathbf{int}(\phi)\ \#\ \alpha] \\
\texttt{iota} & : & \forall\alpha.(\mathbf{int}(\alpha)) \rightarrow \exists\phi.[\mathbf{int}(\phi)\ \#\ \alpha] \\
\texttt{!} & : & \forall\alpha,\beta.([\widehat{\tau}\ \#\ \alpha],\mathbf{int}(\beta)) \rightarrow \widehat{\tau} \\
\texttt{++} & : & \forall\alpha,\beta.([\widehat{\tau}\ \#\ \alpha],[\widehat{\tau}\ \#\ \beta]) \rightarrow [\widehat{\tau}\ \#\ \alpha+\beta] \\
\texttt{concat} & : & \forall\alpha,\phi.([[\widehat{\tau}\ \#\ \phi]\ \#\ \alpha]) \rightarrow [\widehat{\tau}\ \#\ \sum_{\beta=1}^{\alpha}\phi(\beta)] \\
\texttt{sum}_{\mathbf{float}} & : & \forall\alpha.([\mathbf{float}\ \#\ \alpha]) \rightarrow \mathbf{float} \\
\texttt{filter} & : & \forall\alpha.([\widehat{\tau}\ \#\ \alpha],[\mathbf{bool}\ \#\ \alpha]) \rightarrow \exists\beta.[\widehat{\tau}\ \#\ \beta]
\end{array}
$$

Figure 4: Annotated types for builtins

4.1 Pre-lifting

Before shape analysis, we perform a *pre-lifting* pass over the program. This pass first identifies functions that are invoked inside parallel filter or foreach constructs (or inside other lifted functions) and then adds lifted versions as necessary. For the function definition

$$\texttt{fun}\ f\ (x_1,\ \ldots,\ x_k) = e$$

where f has been identified as being used in a parallel context, we add the lifted definition

$$\texttt{fun}\ f^{\uparrow}\ (xs_1,\ \ldots,\ xs_k) = \{\, e : x_1\ \texttt{in}\ xs_1,\ \ldots.\ x_k\ \texttt{in}\ xs_k \,\}$$

to the program. The reason for applying the pre-lifting transformation prior to shape analysis is that it allows a clean separation between shape analysis and the flattening transformation (otherwise we would have to run shape analysis on lifted functions during the flattening phase). Identifying which functions will require lifting is somewhat similar to the problem of identifying maximal sequential subexpressions in vectorization avoidance [11] and could be performed at the same time. Doing shape analysis on the pre-lifted functions does not add any additional complication to the analysis, since shape analysis of a pre-lifted function is no different than analysis of any other function.

4.2 Shape types for library functions

A key source of shape information comes from the application of primitive operations and library functions. Figure 4 presents a sample of these. The integer addition operator ($+_{\mathbf{int}}$) has a special type that tracks the value; other integer operators, such as multiplication return a result with unknown value. The `length` function returns an integer that is equal to the dimension of the argument sequence; likewise, the `lengths` function tracks the original subsequence lengths. The type of the `iota` function captures the fact that the length of the result is determined by the value of the argument (`iota(n)` generates the sequence `[1,...,n]`). For example, we can reason that `iota (length xs)` will have the same length as `xs`. The types of `++` and `concat` reflect the fact that the length of their results will be the sum of the lengths of their arguments.

4.3 Shape analysis

Shape analysis can be viewed as a type inference problem. In our implementation, we assign annotated types to variables and functions, and also generate a set C of equality constraints. We track three kinds of constraints: equality of dimensions ($d_1 = d_2$), equality of dimension functions ($\phi = \psi$), and equality between constant functions and dimensions ($\phi = d$)[5].

Equality constraints are induced by setting two annotated types that have the same underlying structure equal.

$$
\begin{array}{rcl}
\mathbb{C}[\![\ [\widehat{\tau_1}\ \#\ v_1] = [\widehat{\tau_2}\ \#\ v_2]\ ]\!] & = & \mathbb{C}[\![\ \widehat{\tau_1} = \widehat{\tau_2}\ ]\!] \cup \{v_1 = v_2\} \\
\mathbb{C}[\![\ \widehat{\tau_{11}} \times \widehat{\tau_{12}} = \widehat{\tau_{21}} \times \widehat{\tau_{22}}\ ]\!] & = & \mathbb{C}[\![\ \widehat{\tau_{11}} = \widehat{\tau_{21}}\ ]\!] \cup \mathbb{C}[\![\ \widehat{\tau_{12}} = \widehat{\tau_{22}}\ ]\!] \\
\mathbb{C}[\![\ \mathbf{int}(v_1) = \mathbf{int}(v_2)\ ]\!] & = & \{v_1 = v_2\} \\
\mathbb{C}[\![\ \iota = \iota\ ]\!] & = & \emptyset
\end{array}
$$

In the case where we get a constraint of the form $\phi = d$ or $\phi(\alpha) = d$ (where d is not indexed by α), then we mark ϕ as being a *constant* function.

We currently do not do any arithmetic reasoning to determine if two dimensions are equal. Instead, we have found that it is sufficient to put the dimension expressions into canonical form and compare them for symbolic equality. We do check, however, for obvious inconsistencies when adding constraints to the set (*e.g.*, a dimension of 5 cannot be equal to a dimension of 3).

A key assumption that we make for the analysis is that the program is correct with respect to the dimensions of arrays (*i.e.*, it does not have out-of-bounds or unequal-length runtime errors). Because such errors will terminate the Nesl program's execution (either with a runtime error or with a crash when using the "unsafe" compilation mode), being conservative about dimension equality constraints does not seem beneficial. We must be careful, however, to avoid eliminating potential runtime errors that might be misidentified as dead code. As we discuss in Section 6, we might be able to use the shape analysis to identify where to place bounds checks.

The shape analysis proceeds by processing each function definition in turn. We initially annotate the types of the function by assigning fresh shape variables. Over the course of analyzing the function body, constraints will be placed on these variables. When analysis of the function is complete, we resolve the variables to their canonical representation and then close over the free shape variables to produce the function's annotated type.

In the remainder of this section, we describe how our shape analysis handles several of the NKL constructs.

4.3.1 Conditional. For conditionals, we compute an annotated type for each of the arms of the conditional. If $\widehat{\tau_1}$ and $\widehat{\tau_2}$ are the annotated types of the conditional's arms, then we know that $\tau_1 = \tau_2$, since we start with a well-typed program. Thus, the annotated types $\widehat{\tau_1}$ and $\widehat{\tau_2}$ can only differ in their shapes. Given two shapes v_1 and v_2 that appear in corresponding positions of $\widehat{\tau_1}$ and $\widehat{\tau_2}$, we compute a new shape as follows:

[5]This last constraint should be read as as "$\phi = \lambda\alpha.d$;" *i.e.*, ϕ is the constant function that returns d.

- If $C \vdash v_1 = v_2$, where C is the constraint set, then we use v_1.
- If $v_1 = \phi_1(\alpha)$ and $v_2 = \phi_2(\alpha)$, then we use $\psi(\alpha)$, where ψ is fresh. Note that since v_1 and v_2 share the same index (α), we preserve the iteration structure in the result.
- If $v_1 = \alpha_1$, $v_2 = \alpha_2$, and these are not nested inside a sequence, then we use a fresh dimension variable β as the shape.
- Otherwise, we use a fresh dimension-function variable ψ as the shape.

4.3.2 Function application. Consider the function application $f\ (v_1, \ldots, v_k)$, where f has the annotated type

$$\forall \vec{\alpha}, \vec{\phi}.(\widehat{\tau_1}, \ldots, \widehat{\tau_k}) \to \exists \vec{\beta}, \vec{\phi}.\widehat{\tau}$$

We instantiate the function's type by replacing the bound variables with fresh shape variables to get the type

$$(\widehat{\tau'_1}, \ldots, \widehat{\tau'_k}) \to \widehat{\tau'}$$

Then, for each argument v_i with type $\widehat{\tau''_i}$. we add the constraints induced by $\mathbb{C}[\![\ \widehat{\tau'_i} = \widehat{\tau''_i}\ ]\!]$ to the set of constraints. We use a similar approach to handling primitive-operator applications.

4.3.3 Parallel map. A parallel map induces equality constraints on the outermost dimension of the argument arrays. For example, the outermost dimensions of `xs` and `ys` must be equal in the following expression:

```
{ x * y : x in xs; y in ys }
```

Furthermore, the outermost dimension of the result will also be equal to the other dimensions. It is important to note that in this situation we are only imposing equality constraints on the outermost dimension, since the shapes and types of the elements do not have to match.

The one subtlety with analyzing parallel maps is reconstructing nested structure without losing shape information. Consider the expression

```
{ { x * x : x in xs } : xs in xss }
```

where `xss` has the annotated type $[[\textbf{float} \# \phi] \# \alpha]$. When we analyze the outer parallel map, we need to assign a valid annotated type to `xs`. We use $[\textbf{float} \# \phi]@\beta = [\textbf{float} \# \phi(\beta)]$ as the type of `xs`, where β is a fresh variable that represents the iteration index of the map. In this example, the result type of the inner map will be $[\textbf{float} \# \phi(\beta)]$, we then generalize the shape $\phi(\beta)$ to ϕ and get the type $[[\textbf{float} \# \phi] \# \alpha]$ for the whole expression.

4.4 Detecting rectangular structure

We conclude the discussion of shape analysis by revisiting the `mv` example from Section 3.

```
function mv (xss: [[float]], ys: [float]) =
  { { x + y : x in xs; y in ys }
  : xs in xss };
```

Shape analysis will begin by assigning the types of the parameters:

$$\texttt{xss} : [[\textbf{float} \# \phi] \# \alpha]$$
$$\texttt{ys} : [\textbf{float} \# \beta]$$

$$\begin{array}{rcll}
\overline{\tau} & ::= & \texttt{[:}\iota\texttt{:]} & \text{base-sequence type} \\
& | & \overline{\tau_1} \times \overline{\tau_2} & \text{pair type} \\
& | & \iota & \text{base type} \\
& | & \sigma & \text{shape descriptor type} \\
\sigma & ::= & \textbf{sd}(d * v) & \text{segment descriptor} \\
& | & \textbf{sz}(d) & \text{base-sequence size}
\end{array}$$

Figure 5: Flattened types

When processing the outer parallel map, we give `xs` the type

$$\texttt{xs} : [\textbf{float} \# \phi(\gamma)]$$

Processing the inner map requires setting the (outermost) dimensions of `xs` and `ys` to be equal — $\{\phi(\gamma) = \beta\}$ — which means that ϕ must be a constant function. We end up with a result type of $[\textbf{float} \# \beta]$ for the inner map and $[[\textbf{float} \# \phi(\gamma)] \# \alpha]$ for the outer map, and thus, `mv` has the type we predicted in Section 3.3.

5 FLATTENING WITH SHAPES

Once we have annotated the NKL program with shape information, the next step is to flatten the program into the FDP representation. The syntax of the FDP language, called FKL, is a subset of NKL with the parallel map constructs removed. The types for FKL are different, however, and are shown in Figure 5. Notice that sequences in FKL may only contain base types and that we have types for segment descriptors and sequence sizes. The segment-descriptor type $\textbf{sd}(d{*}v)$ describes a segment descriptor with d segments, where v defines the size of the segments, while the base-sequence size type $\textbf{sz}(d)$ specifies the size of an associated base sequence as being d. We do not transfer all of the shape information from the annotated types to the flattened types — specifically we no longer track integer values — but by including the shape information in the segment descriptors, we can determine when two segment descriptors are the same.

5.1 Translating types

The first part of the flattening transformation is the conversion from types annotated with shape information to FKL types. This translation is defined as follows:

$$\begin{array}{rcll}
\mathbb{T}[\![\ [\iota \# v]\ ]\!]\, d & = & \textbf{sz}(d) \times \texttt{[:}\mathbb{T}[\![\ \iota\ ]\!]\, 1\texttt{:]} & \\
\mathbb{T}[\![\ [[\widehat{\tau} \# v] \# d']\ ]\!]\, d & = & \textbf{sd}(d * v) \times \mathbb{T}[\![\ [\widehat{\tau} \# \lfloor v \rfloor]\ ]\!]\, (d' \circledast d) & \\
\mathbb{T}[\![\ [\widehat{\tau_1} \times \widehat{\tau_2} \# d']\ ]\!]\, d & = & \mathbb{T}[\![\ [\widehat{\tau_1} \# d']\ ]\!]\, d \times \mathbb{T}[\![\ [\widehat{\tau_2} \# d']\ ]\!]\, d & \\
\mathbb{T}[\![\ \widehat{\tau_1} \times \widehat{\tau_2}\ ]\!]\, d & = & \mathbb{T}[\![\ \widehat{\tau_1}\ ]\!]\, d \times \mathbb{T}[\![\ \widehat{\tau_2}\ ]\!]\, d & \\
\mathbb{T}[\![\ \textbf{int}(v)\ ]\!]\, d & = & \textbf{int} & \\
\mathbb{T}[\![\ \iota\ ]\!]\, d & = & \iota & \text{otherwise}
\end{array}$$

where the second argument (d) to $\mathbb{T}$ is the cumulative size of the array. We use an initial value of 1 when translating an annotated type and we use the dimension multiplication operator "$\circledast$", which is defined as

$$\begin{array}{rcll}
n \circledast m & = & n * m & \\
d \circledast d' & = & \alpha & \text{where } \alpha \text{ is fresh}
\end{array}$$

This allows us to statically determine the actual size of the data sequence in the case where we have a rectangular array of known dimensions. We also use the notation $\lfloor\, d \,\rfloor$, which is defined as follows:

$$\begin{array}{lcl} \lfloor\, d \,\rfloor & = & d \\ \lfloor\, \phi \,\rfloor & = & \alpha \quad \text{where } \alpha \text{ is fresh} \end{array}$$

This notation is used when stripping off a level of sequence so that the resulting type is well formed.

Here are some examples of the type translation:

$$\mathbb{T}[\![\, [[\mathbf{float} \# 5] \# 5] \,]\!]\, 1 = \mathbf{sd}(5 * 5) \times \mathbf{sz}(25) \times [\mathbf{:float:}]$$

$$\mathbb{T}[\![\, [[\mathbf{float} \# \phi] \# 5] \,]\!]\, 1 = \mathbf{sd}(5 * \phi) \times \mathbf{sz}(\alpha) \times [\mathbf{:float:}]$$

$$\mathbb{T}[\![\, [\mathbf{int}(\phi) \# \alpha] \,]\!]\, 1 = \mathbf{sz}(\alpha) \times [\mathbf{:int:}]$$

$$\mathbb{T}[\![\, [\mathbf{bool} \times [\mathbf{float} \# \phi] \# \alpha] \,]\!]\, 1 = (\mathbf{sz}(\alpha) \times [\mathbf{:bool:}]) \times (\mathbf{sd}(\alpha * \phi) \times \mathbf{sz}(\beta) \times [\mathbf{:float:}])$$

5.2 Target operations

In Section 2.3, we described how the $\mathcal{S}$, $\mathcal{F}$, and $\mathcal{P}$ operators are used to eliminate excess levels of lifting. These operations are implemented at the meta-level in our translation. Furthermore, we use type-indexed definitions because of the flat representation for sequences of pairs. These operations have the following annotated types, where the element type $\widehat{\tau}$ is the type index.

$$\begin{array}{rcl} \mathcal{S}_{\widehat{\tau}} & : & \forall\alpha, \phi.([[\widehat{\tau} \# \phi] \# \alpha]) \rightarrow [\mathbf{int}(\phi) \# \alpha] \\ \mathcal{F}_{\widehat{\tau}} & : & \forall\alpha, \phi.([[\widehat{\tau} \# \phi] \# \alpha]) \rightarrow [\widehat{\tau} \# \textstyle\sum_{\alpha=1}^{\phi(\beta)} \beta(\alpha)] \\ \mathcal{P}_{\widehat{\tau}} & : & \forall\alpha, \phi.([\widehat{\tau} \# \alpha], [\mathbf{int}(\phi) \# \alpha]) \rightarrow [[\widehat{\tau} \# \phi] \# \alpha] \end{array}$$

The definition of these meta operations is as follows:

$$\begin{array}{rcl} \mathcal{S}_{\widehat{\tau_1} \times \widehat{\tau_2}} & = & \lambda\langle a,\, b\rangle.\mathcal{S}_{\widehat{\tau_1}}(a) \\ \mathcal{S}_{\widehat{\tau}} & = & \lambda\langle sd,\, seq\rangle.sd \\ \mathcal{F}_{\widehat{\tau_1} \times \widehat{\tau_2}} & = & \lambda\langle a,\, b\rangle.\langle \mathcal{F}_{\widehat{\tau_1}}(a),\, \mathcal{F}_{\widehat{\tau_2}}(b)\rangle \\ \mathcal{F}_{\widehat{\tau}} & = & \lambda\langle sd,\, seq\rangle.seq \\ \mathcal{P}_{\widehat{\tau_1} \times \widehat{\tau_2}} & = & \lambda(\langle a,\, b\rangle, sd).\langle \mathcal{P}_{\widehat{\tau_1}}(a, sd),\, \mathcal{P}_{\widehat{\tau_2}}(b, sd)\rangle \\ \mathcal{P}_{\widehat{\tau}} & = & \lambda(seq, sd).\langle sd,\, seq\rangle \end{array}$$

We also need the primitive $\# : (\sigma) \rightarrow \mathbf{int}$ that returns the length of the sequence described by a shape descriptor.

We also introduce several FKL primitives for working with sequences, which are described in Figure 6. The **SPLIT** and **COMBINE** primitives are used to implement conditionals and the **FILTER** primitive is used to implement parallel filters. The **DIST**, **INDEX**, **SDIST**, and **TDIST** primitives are used to generate argument sequences. Lastly, the $\mathbf{SD}_{\sigma}$ primitive creates a segment descriptor of the specified type.

As will be shown below, the **DIST** primitive is used to replicate constants and variables to form sequences that can be used as arguments to lifted operations. In the case where the value being replicated is not a base value, the replication operation is more complicated, since the result must conform to the flattened-type representation described in the previous section. Therefore, we define a type-indexed function $\mathtt{dist}_{\widehat{\tau}}$ that expands to the necessary code.

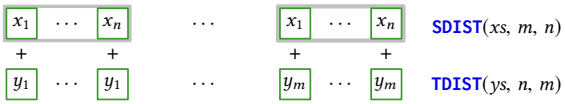
$$\begin{array}{rcl} \mathtt{dist}_{\widehat{\tau_1} \times \widehat{\tau_2}} & = & \lambda(\langle a,\, b\rangle, n).\langle \mathtt{dist}_{\widehat{\tau_1}}(a, n),\, \mathtt{dist}_{\widehat{\tau_2}}(b, n)\rangle \\ \mathtt{dist}_{[\widehat{\tau} \# v]} & = & \lambda(\langle sd,\, seq\rangle, n). \\ & & \quad \langle n,\, \langle \mathbf{SD}_{\mathbf{sd}(n * v)}(\mathbf{SDIST}(sd,\, n,\, \#(sd))), \\ & & \qquad \mathbf{SDIST}(seq,\, n,\, sd)\rangle\rangle \\ \mathtt{dist}_{[\widehat{\tau} \# v]} & = & \lambda(\langle sd,\, seq\rangle, n). \\ & & \quad \langle n,\, \langle \mathbf{SD}_{\mathbf{sd}(n * v)}(\mathbf{SDIST}(sd,\, n,\, \#(sd))), \\ & & \qquad \mathbf{SDIST}(\mathtt{dist}_{\widehat{\tau}}(seq,\, n),\, n,\, sd)\rangle\rangle \\ \mathtt{dist}_{\widehat{\tau}} & = & \lambda(x, n).\langle n,\, \mathbf{DIST}(x,\, n)\rangle \end{array}$$

Here the **SDIST** primitive is used to create multiple copies of a sequence and we also use it to build the segment descriptors for the nested sequences.

The **TDIST** primitive is required to support nesting of parallel maps. For example, consider the expression

```
{ { x + y : x in xs } : y in ys }
```

where $xs = [x_1, \ldots, x_n]$ and $ys = [y_1, \ldots, y_m]$. This expression is flattened to a lifted addition ($+^{\uparrow}$) applied to two sequences, where the first argument is the xs sequence replicated m times and the second sequence is the concatenation of n copies of y_1, followed by n copies of y_2, up to m copies of y_m. The **TDIST** primitive is used to create this second sequence, as illustrated in the following diagram:



Note that the **TDIST** primitive could be defined in terms of **DIST** as follows:

$$\mathbf{TDIST}(ys,\, n,\, m) = \mathbf{DIST}^{\uparrow}(ys,\, \mathbf{DIST}(n,\, m))$$

but it is useful to have it as a primitive. As with DIST, we define a type-indexed function $\mathtt{tdist}_{\widehat{\tau}}$ to handle the cases where the first argument is not a sequence of base values. We also define $\mathtt{combine}_{\widehat{\tau}}$ as a type-indexed version of the **COMBINE** primitive.

5.3 Flattening

Our presentation of the flattening transformation roughly follows that of Keller [9], except that we also track shape information. We organize the translation by the syntactic forms found in Figure 1 and define three corresponding flatting translations for terms inside a parallel map.

$$\begin{array}{rr} \mathbb{E}[\![\, e \triangleleft \vec{b} \,]\!]_{sd} & \text{flatten expression } e \\ \mathbb{R}[\![\, r \triangleleft \vec{b} \,]\!]_{sd} & \text{flatten right-hand-side } r \\ \mathbb{V}[\![\, v \triangleleft \vec{b} \,]\!]_{sd} & \text{flatten value } v \end{array}$$

For each of these translations, $\vec{b}$ is the sequence of bindings for the map and sd is the size/segment descriptor of the result.

Although FKL is a normalized IR, we take the liberty of using direct-style notation for the result of the translation to simplify its specification. In the actual implementation, the result is normalized.

COMBINE	:	$(\mathbf{sz}(d_1) \times \texttt{[:}\iota\texttt{:]},\ \mathbf{sz}(d_2) \times \texttt{[:}\iota\texttt{:]},\ \mathbf{sz}(d) \times \texttt{[:bool:]}) \to \texttt{[:}\iota\texttt{:]}$	*merge two sequences according to the boolean flags*
DIST	:	$(\iota,\ \mathbf{int}) \to \texttt{[:}\iota\texttt{:]}$	*replicate the first argument by given length*
FILTER	:	$(\texttt{[:}\iota\texttt{:]},\ \texttt{[:bool:]},\ \mathbf{sz}(d)) \to \mathbf{sz}(d_1) \times \texttt{[:}\iota\texttt{:]}$	*filter out elements of the first argument where the corresponding element in the second argument is true*
INDEX	:	$(\texttt{[:int:]},\ \texttt{[:int:]},\ \texttt{[:int:]},\ \mathbf{sz}(d)) \to \mathbf{sz}(d') \times \texttt{[:int:]}$	*given a sequence of start values, strides, and lengths, build a flattened sequence index sequences*
SPLIT	:	$(\texttt{[:}\iota\texttt{:]},\ \texttt{[:bool:]},\ \mathbf{sz}(d)) \to (\mathbf{sz}(d_1) \times \texttt{[:}\iota\texttt{:]}) \times (\mathbf{sz}(d_2) \times \texttt{[:}\iota\texttt{:]})$	*split the first argument according to the values in the second argument*
SDIST	:	$(\texttt{[:}\iota\texttt{:]},\ \mathbf{int},\ \mathbf{int}) \to \texttt{[:}\iota\texttt{:]}$	*replication of the first argument; the second argument is the number of copies and the third argument is the length of the first argument*
$\mathbf{SD}_\sigma$	:	$(\mathbf{int},\ \texttt{[:int:]}) \to \sigma$	*build a segment descriptor from a length and sequence of segment lengths*
TDIST	:	$(\texttt{[:}\iota\texttt{:]},\ \mathbf{int},\ \mathbf{int}) \to \texttt{[:}\iota\texttt{:]}$	*transposed replication of the first argument; the second argument is the number of copies and the third argument is the length of the first argument*

Figure 6: Primitive sequence operations for FKL

5.3.1 Flattening expressions. For expressions that are inside a parallel map context, we use the translation $\mathbb{E}[\![\ e \lhd \vec{b}\ ]\!]_{sd}$, where e is the expression being flattened, $\vec{b}$ is the sequence of bindings for the map, and sd is the size/segment descriptor of the result.

$$\mathbb{E}[\![\ v \lhd \vec{b}\ ]\!]_{sd} = \mathbb{V}[\![\ v \lhd \vec{b}\ ]\!]_{sd}$$

$$\begin{aligned}
&\mathbb{E}[\![\ \texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \lhd \overrightarrow{x_i\ \texttt{in}\ xs_i}\ ]\!]_{sd} = \\
&\quad \texttt{let}\ flgs = \mathbb{E}[\![\ p \lhd \overrightarrow{x_i\ \texttt{in}\ xs_i}\ ]\!]_{sd} \\
&\quad \overrightarrow{\texttt{let}\ \langle ys_i,\ zs_i\rangle = \texttt{split}_{\text{typeof}(x_i)}(xs_i, flgs, sd)} \\
&\quad \texttt{let}\ res_1 = \mathbb{E}[\![\ e_1 \lhd \overrightarrow{x_i\ \texttt{in}\ ys_i}\ ]\!]_{\mathcal{S}(ys_1)} \\
&\quad \texttt{let}\ res_2 = \mathbb{E}[\![\ e_2 \lhd \overrightarrow{x_i\ \texttt{in}\ zs_i}\ ]\!]_{\mathcal{S}(zs_2)} \\
&\quad \texttt{in} \\
&\qquad \texttt{combine}_{\text{typeof}(e_1)}(res_1, res_2, flgs)
\end{aligned}$$

$$\begin{aligned}
&\mathbb{E}[\![\ f\ (v_1,\ \ldots,\ v_k) \lhd \vec{b}\ ]\!]_{sd} = \\
&\quad \texttt{if}\ \#(sd) = 0 \\
&\qquad \texttt{then}\ \langle sd,\ [\ ]\rangle \\
&\qquad \texttt{else}\ f^{\uparrow}\ (\mathbb{V}[\![\ v_1 \lhd \vec{b}\ ]\!]_{sd},\ \ldots,\ \mathbb{V}[\![\ v_k \lhd \vec{b}\ ]\!]_{sd},\ sd)
\end{aligned}$$

$$\begin{aligned}
&\mathbb{E}[\![\ \texttt{let}\ y = r\ \texttt{in}\ e \lhd \vec{b}\ ]\!]_{sd} = \\
&\quad \texttt{let}\ ys = \mathbb{R}[\![\ r \lhd \vec{b}\ ]\!]_{sd} \\
&\quad \texttt{in} \\
&\qquad \mathbb{E}[\![\ e \lhd y\ \texttt{in}\ ys, \vec{b}\ ]\!]_{sd}
\end{aligned}$$

5.3.2 Flattening right-hand sides. For right-hand sides of let bindings, we use the translation $\mathbb{R}[\![\ r \lhd \vec{b}\ ]\!]_{sd}$, where r is the right-hand side being flattened, $\vec{b}$ is the sequence of bindings for the iteration, sd is the size/segment descriptor of the result. The flattening translation of most right-hand-side forms is straightforward:

$$\mathbb{R}[\![\ e \lhd \vec{b}\ ]\!]_{sd} = \mathbb{E}[\![\ e \lhd \vec{b}\ ]\!]_{sd}$$

$$\begin{aligned}
&\mathbb{R}[\![\ p\ (v_1,\ \ldots,\ v_k) \lhd \vec{b}\ ]\!]_{sd} = \\
&\quad p^{\uparrow}\ (sd,\ \mathbb{V}[\![\ v_1 \lhd \vec{b}\ ]\!]_{sd},\ \ldots,\ \mathbb{V}[\![\ v_k \lhd \vec{b}\ ]\!]_{sd})
\end{aligned}$$

$$\begin{aligned}
&\mathbb{R}[\![\ \langle v_1,\ v_2\rangle \lhd \vec{b}\ ]\!]_{sd} = \\
&\quad \langle \mathbb{V}[\![\ v_1 \lhd \vec{b}\ ]\!]_{sd},\ \mathbb{V}[\![\ v_2 \lhd \vec{b}\ ]\!]_{sd}\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathbb{R}[\![\ [\ v_1, \ldots, v_k\ ] \lhd \vec{b}\ ]\!]_{sd} = \\
&\quad \mathbb{V}[\![\ v_1 \lhd \vec{b}\ ]\!]_{sd}\ {}^{++}_{\widehat{\tau}}\ \cdots\ {}^{++}_{\widehat{\tau}}\ \mathbb{V}[\![\ v_k \lhd \vec{b}\ ]\!]_{sd} \\
&\qquad \text{where typeof}(v_1) = \widehat{\tau}
\end{aligned}$$

The case for flattening nested parallel maps is more complicated. Keller's translation relies on multiple translation passes to handle nested parallel maps, which results in multiple levels of lifting. As described in Section 2.3, the following identity allows multiple levels of lifting to be reduced to a single level:

$$f^{\uparrow\uparrow}(\overrightarrow{xss_i}) = \mathcal{P}(f^{\uparrow}(\overrightarrow{\mathcal{F}(xss_i)}), \#(xss_1))$$

In our approach, we lift the outer bindings in anticipation that they might be referred to inside the nested parallel map. Dead-code

elimination will remove any unused liftings.

$$
\begin{array}{l}
\mathbb{R}[\![\ \{\ e : \overrightarrow{x_i \texttt{ in } xs_i}\ \} \triangleleft \overrightarrow{y_j \texttt{ in } ys_j}\ ]\!]_{sd} = \\
\quad \texttt{let } sd' = \mathcal{S}_{\text{typeof}(x_1)}(xs_1) \\
\quad \overrightarrow{\texttt{let } xs'_i = \mathcal{F}_{\text{typeof}(xs_i)}(\mathbb{V}[\![\ xs_i \triangleleft \overrightarrow{y_j \texttt{ in } ys_j}\ ]\!]_{sd})} \\
\quad \overrightarrow{\texttt{let } ys'_j = \texttt{tdist}_{\text{typeof}(y_j)}(ys_j, \#(sd'), \#(sd))} \\
\quad \texttt{let } res = \mathbb{E}[\![\ e \triangleleft \overrightarrow{x_i \texttt{ in } xs'_i},\ \overrightarrow{y_j \texttt{ in } ys'_j}\ ]\!]_{sd'} \\
\quad \texttt{in} \\
\qquad \mathcal{P}_{\text{typeof}(e)}(res, sd)
\end{array}
$$

5.3.3 Flattening values. We use the translation $\mathbb{V}[\![\ v \triangleleft \overrightarrow{b}\ ]\!]_{sd}$ for values, where v is the value being flattened, $\overrightarrow{b}$ is the sequence of bindings for the iteration, sd is the size/segment descriptor of the result.

$$\mathbb{V}[\![\ n \triangleleft \overrightarrow{b}\ ]\!]_{sd} = \texttt{DIST}(n, sd)$$

$$\mathbb{V}[\![\ x \triangleleft x \texttt{ in } xs,\ \overrightarrow{b}\ ]\!]_{sd} = xs$$

$$\mathbb{V}[\![\ x \triangleleft y \texttt{ in } ys,\ \overrightarrow{b}\ ]\!]_{sd} = \mathbb{V}[\![\ x \triangleleft \overrightarrow{b}\ ]\!]_{sd}$$

$$\mathbb{V}[\![\ x \triangleleft \cdot\ ]\!]_{sd} = \texttt{dist}_{\text{typeof}(x)}(x, sd)$$

For base constants, we can use the **DIST** primitive to lift a single value to a sequence of values. If a variable x is in the binding list, then we can replace it with the corresponding sequence variable; otherwise, we use the type-indexed $\texttt{dist}_{\hat{\tau}}$ operation to generate the necessary FKL code.

6 STATUS AND FUTURE WORK

We have implemented a prototype of the analysis and shape-preserving flattening transformations for NKL. Based on this prototype, we have added the analysis to the Nessie compiler.[6] The analysis performs well on the roughly 35 test programs that we have tried. As hoped, it is able to identify rectangular structure and also handles the extensive collection of predefined functions that are part of the Nesl language. Rectangular structures arise frequently in Nesl programs, even when the underlying algorithm is an irregular parallel computation, so we expect hope to see real performance benefits from optimizing those cases. The analysis also does an effective job of identifying opportunities for horizontal fusion (as described in Section 2.4.

We have not addressed the *replication problem* that some Nesl programs suffer from, but we believe that the techniques developed by other researchers to solve this problem are compatible with our approach [14, 15].

Vectorization avoidance is a technique to increase the granularity of primitive operations prior to flattening [11]. This technique identifies maximal subexpressions that are sequential and packages them up as indivisible operations that are not subject to decomposition by the flattening transformation. We have previously implemented this technique in the Nessie compiler and believe that we can combine the pre-lifting analysis from Section 4.1 with the analysis that identifies sequential subexpressions.

[6] As of this writing, we have shape analysis for full Nesl implemented and are working on the flattening transformation.

The shape information that annotates segment descriptors can be used to eliminate redundant descriptors and optimize the representation of descriptors. In particular, one of the major improvements of our new shape analysis is that it can identify regular rectangular arrays. There are some obvious opportunities for exploiting this information, such as using more compact representations of segment descriptors and more efficient segmented scan and reduction operations, but we have not worked out the details of these optimizations yet.

In Nesl, maps over multiple sequences require that the sequences have the same length; if they do not, the program terminates with an error (or crashes when running in *unchecked* mode). Our shape analysis assumes that execution does not encounter such errors. This assumption is manifest when we constrain the lengths of two sequences that are involved in a map to be equal.[7] It is interesting to note that source of potential unequal-length errors corresponds to constraining existentially-quantified shape variables. For example, consider the following contrived Nesl code:

```
let xs2 = { x : x in xs1 | x < 0 };
    ys2 = { y : y in ys1 | y < 0 };
in
  { x + y : x in xs2, y in ys2 }
```

Because these sequences are defined by filters, they will have existentially-quantified dimension variables as their lengths. For this program to run correctly, however, these variables must be equal, so the shape analysis will force that constraint. We could exploit the connection between existentially-quantified variables and unequal-length errors to place error-checking code. We may be able to use some of the ideas from Futhark for this purpose [8].

Once we have sorted out the issue of bounds checking, it should be possible to prove the correctness of the shape analysis, but this is left for future work.

7 RELATED WORK

The flattening transformation dates back to Blelloch's Ph.D. research [3, 7], but Blelloch only described the transformation using informal English-language prose. Keller developed a more rigorous description [9, 12] in her dissertation, which we loosely follow here. Our approach differs from Keller in that we handle nested parallel maps directly, whereas Keller transforms the inner maps first, which results in multiple passes. Keller's transformation also extends the Nesl model with support for inductive types, which we have not done. Leshchinskiy developed a formal transformation for Data-Parallel Haskell (DPH) [13]. DPH not only supports inductive types, but also supports higher-order functions. He uses an approach of representing functions as pairs of closures; one for the original function and one for the lifted version of the function. None of these flattening transformations, however, tracks the shapes of irregular nested computations.

Analyzing the shapes of arrays has been an important research problem in the area of array-language compilers, but all of the work that we are aware of has focused on rectangular arrays. Scholz has developed a type system for shapes in the functional array language Single-Assignment C (SaC) [22]. His system supports a limited form

[7] Various builtin functions introduce similar constraints.

of dimension polymorphism, which is not part of our system, but it is limited to rectangular array shapes. A related idea, also for SaC, allows users to define constraints on the shapes of arrays that have polymorphic structure as a way to provide more information about the structure of shape-polymorphic code [23]. Their constraints are similar to the constraints that we induce during shape analysis. The Repa library for GHC uses Haskell's powerful type system to define shape polymorphic code [10]. The Futhark language uses a hybrid approach that combines static analysis of array shapes with a dynamic fallback mechanism that can check situations that were not possible to resolve statically [8]. They also use a novel program slicing method to extract code for computing array shapes at runtime. Like the other works listed, Futhark does not support irregular nested arrays.

8 CONCLUSION

We have described a new approach to the problem of determining shape information for the FDP code produced by the flattening transformation. This approach is simpler than previous approaches, because it analyzes the shapes of the NDP code before the flattening transformation. We have prototyped this approach for a kernel language NKL and are currently integrating it into Nessie, our Nesl to CUDA compiler.

ACKNOWLEDGMENTS

Nora Sandler implemented the original flattening transformation and shape analysis for the Nessie compiler. Portions of this research were supported by the National Science Foundation under award CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

REFERENCES

[1] Lars Bergstrom and John Reppy. 2012. Nested Data-Parallelism on the GPU. In *ICFP '12* (Copenhagen, Denmark). ACM, New York, NY, 247–258.
[2] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *IEEE Computer* 38, 11 (Nov. 1989), 1526–1538.
[3] Guy E. Blelloch. 1990. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA.
[4] Guy E. Blelloch. 1995. *NESL: A nested data-parallel language (version 3.1).* Technical Report CMU-CS-95-170. School of C.S., CMU, Pittsburgh, PA.
[5] Guy E. Blelloch. 1996. Programming parallel algorithms. *CACM* 39, 3 (March 1996), 85–97.
[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. 1994. Implementation of a portable nested data-parallel language. *JPDC* 21, 1 (1994), 4–14.
[7] Guy E. Blelloch and Gary W. Sabot. 1990. Compiling collection-oriented languages onto massively parallel computers. *JPDC* 8, 2 (1990), 119–134.
[8] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2014. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In *FHPC '14* (Gothenburg, Sweden). ACM, New York, NY, 31–42.
[9] Gabriele Keller. 1999. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines.* Ph.D. Dissertation. Technische Universität Berlin, Berlin, Germany.
[10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP '10* (Baltimore, MD). ACM, New York, NY, 261–272. https://doi.org/10.1145/1863543.1863582
[11] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. 2012. Vectorisation Avoidance. In *HASKELL '12* (Copenhagen, Denmark). ACM, New York, NY, 37–48.
[12] Gabriele Keller and Martin Simons. 1996. A Calculational Approach to Flattening Nested Data Parallelism in Functional Languages. In *Concurrency and Parallelism, Programming, Networking, and Security (LNCS)*, Joxan Jaffar and Roland H. C. Yap (Eds.), Vol. 1179. Springer-Verlag, New York, NY, 234–243.
[13] Roman Leshchinskiy. 2005. *Higher-Order Nested Data Parallelism: Semantics and Implementation.* Ph.D. Dissertation. Technische Universität Berlin, Berlin, Germany.
[14] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. 2012. Work Efficient Higher-order Vectorisation. In *ICFP '12* (Copenhagen, Denmark). ACM, New York, NY, 259–270.
[15] Frederik M. Madsen. 2012. Flattening Nested Data Parallelism. Master's Project, DIKU. Available from http://hiperfit.dk/publications.
[16] Jan F. Prins and Daniel W. Palmer. 1993. Transforming High-Level Data-Parallel Programs into Vector Operations. In *PPoPP '93* (San Diego, CA). ACM, New York, NY, 119–128.
[17] John Reppy and Nora Sandler. 2015. Nessie: A NESL to CUDA Compiler. Presented at CPC 2015; London, UK. , 13 pages. Available from https://nessie.cs.uchicago.edu.
[18] John Reppy and Joe Wingerter. 2016. λ_{cu} — An Intermediate Representation for Compiling Nested Data Parallelism. Presented at CPC 2016; Valladolid, Spain.. , 13 pages. Available from https://cpc2016.infor.uva.es.
[19] Amos Robinson, Ben Lippmeier, and Gabriele Keller. 2014. Fusing Filters with Integer Linear Programming. In *FHPC '14* (Gothenburg, Sweden). ACM, New York, NY, 53–62.
[20] Nora Sandler. 2014. Nessie: A New NESL Compiler. (June 2014). BA Honors Thesis, Department of Computer Science, University of Chicago.
[21] Scandal Project. [n.d.]. A library of parallel algorithms written in NESL. Available from http://www.cs.cmu.edu/~scandal/nesl/algorithms.html.
[22] Sven-Bodo Scholz. 2001. A Type System for Inferring Array Shapes. In *IFL '01* (Stockholm, Sweden) *(LNCS)*, Thomas Arts and Markus Mohnen (Eds.). Springer-Verlag, New York, NY, 65–82.
[23] Fangyong Tang and Clemens Grelck. 2013. User-Defined Shape Constraints in SAC. Presented at IFL 2012; Oxford U.K.. , 19 pages. Available from www.sac-home.org.
[24] Kai Trojahner, Clemens Grelck, and Sven-Bodo Scholz. 2006. On Optimising Shape-Generic Array Programs Using Symbolic Structural Information. In *IFL '06* (Budapest, Hungary), Zoltán Horváth, Viktória Zsók, and Andrew Butterfield (Eds.). Springer-Verlag, New York, NY, 1–18.
[25] Joe Wingerter. 2017. λ_{cu} *— An Intermediate Representation for Compiling Nested Data Parallelism.* Master's thesis. University of Chicago.
[26] Yongpeng Zhang and Frank Mueller. 2012. CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures. In *ICPP '12* (Pittsburgh, PA). IEEE Computer Society Press, Los Alamitos, CA, 340–349.