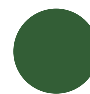
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN



Master's Thesis

Frederik M. Madsen

A streaming model for nested data parallelism

Supervisor: Andrzej Filinski

March 2013

Abstract

Efficient parallel algorithms are often written with embedded knowledge of the back-end that they are meant to be executed on, and if they are not, the translation to target language often produces inefficient code. A concrete problem is space complexity in nested data parallel (NDP) languages such as NESL and Data Parallel Haskell, where large intermediate arrays are often allocated during execution. This thesis presents an NDP language with a streaming based model where the time complexity of programs is just as good as in traditional NDP languages, but the space complexity is significantly better in many cases. A minimal NDP language with semantics and a desirable cost model is defined, as well as a streaming based target language, and the two languages are related with a translation, a proof-of-concept implementation and a conjecture about value and cost preservation.

Resumé

Effektive parallel algoritmer er ofte skrevet med indlejret viden om platformen de skal køre på, og hvis de ikke er, producerer oversættelsen til målsprog ofte ineffektiv kode. Et konkret problem er pladsforbruget i nestede data-parallele (NDP) sprog så som NESL og Data Parallel Haskell, hvor store midlertidige array's ofte allokeres under eksekvering. Dette speciale præsenterer et NDP sprog med en strøm-baseret model hvor tidsforbuget er det samme som for traditionelle NDP sprog, men pladsforbruget er signifikant bedre i mange tilfælde. Et minimalt NDP sprog med semantik of ønskværdig omkostning-model bliver defineret, så vel som et strøm-baseret målsprog, og de to sprog sammenholdes med en oversættelse, en proof-of-concept implementation samt en formodning omkring værdi- og omkostnings-bevaring.

Contents

1 Introduction **4**
- 1.1 Data parallelism 4
- 1.2 Nested data parallelism (NDP) 6
- 1.3 Problem statement 9

2 Analysis **10**
- 2.1 NDP language 10
- 2.2 CUDA 12

3 Source language **14**
- 3.1 Syntax 14
- 3.2 Type system 16
- 3.3 Syntactic sugar 20
- 3.4 Big-step semantics 22
- 3.5 Cost semantics 26
- 3.6 Examples 30
- 3.7 Evaluation 34

4 Streaming language **35**
- 4.1 Syntax 35
- 4.2 Type system 36
- 4.3 Interpretation 39
- 4.4 Derived complexities 47
- 4.5 Examples 49
- 4.6 Recursion 52
- 4.7 Evaluation 52

5 Transformation **53**
- 5.1 Value representation 53
- 5.2 Type transformation 55
- 5.3 Expression transformation 56
- 5.4 Value and cost preservation 67

6 Implementation **68**
- 6.1 Runtime system 68
- 6.2 Compiler 68
- 6.3 Back end 69
- 6.4 Evaluation 69

7 Conclusion **71**

1 Introduction

The amount of space required to run an NDP program is often less then ideal if the the program contains reductions, even for trivial programs. The main cause lies in the vectorization to a flat target language, where programs are viewed as a series of very big vector instructions, executed one at a time. This thesis proposes a different approach where the vector instructions are connected in a directed acyclic graph, and each instruction is allowed to execute partially and multiple times, thus creating a data-flow network. Instead of vectors, we have streams, and instead of vector instructions, we have stream transformers, with one additional detail - the transformers are operates in chunks and are data parallel. To achieve such a model, we first describe a suitable source language and a target language and we present a transformation from the former to the latter. Both languages are given a reasonable cost model.

1.1 Data parallelism

Data parallelism is one way of approaching parallelism in programming. The data parallel approach covers parallelism based on data, as opposed to other fundamental entities, such as number of threads. In essence, a data parallel programming language is able to express that some non-unit quantity of computational work can be done in a parallel , and the actual amount of work can depend on runtime data (e.g. input size).

It is important to note, that data parallel programming languages almost always express work that *can* be done in parallel, not work that *must* be done in parallel. This is because the *available parallel degree*, the number of available processing units, can vary greatly from back-end to back-end, and it is usually easy for the compiler to "chunk up" too parallel steps into several steps, whereas it would be tedious, if not impossible, for the programmer to do so for each back-end. The amount of work in a single step can therefore be used as a measure of *potential parallel degree* of a program, and once a program is executed on a specific back-end, we can measure the *actual parallel degree* of the program, which should be the same as the minimum of the available and potential parallel degree.

Data parallelism in a programming language can be achieved simply by having a library of parallel algorithms. A common way to introduce more expressive data parallelism, is to introduce a new construct that essentially is a parallel *foreach loop*, or a parallel *map*:

$$\{e_0 : x \textbf{ in } e_1\}$$

We call it apply-to-each, and the semantic is, first e_0 is evaluated to a list, and then the body expression e_1 is evaluated for each element in the list substituted for x to produce the resulting list.

A key point is, that each evaluation of the body must be independent in order to claim that they can be executed in parallel or out of order without unexpected results, and it is therefore necessary that the body expression has no side-effects, that can effect the parallel evaluations.

1.1.1 Homogeneous data parallelism

The parallel foreach loop construct and the parallel map closely resembles the notion of single instruction multiple data (SIMD). The resemblance can be seen, by realizing that

an expression in a source language is translated into a series of instructions, and the parallel foreach construct has a single body expression executed over multiple elements of a data list. This translates into having the same series of instructions executed over multiple data elements, where each instruction in the series of instructions is essentially SIMD parallel.

The resemblance is not an equivalence, since it breaks in the presence of branching where the body expression might branch to completely different instructions based on the data element. In the case of an if-then-else we can run all the true branches first and all the false branches afterward at the cost of increased step complexity. In the worst case, the execution of the body branch to a different instruction for all the elements in the list, and in this case absolutely no parallelism is exposed by translation to a SIMD model.

Homogeneous data parallelism is data parallelism, but where parallel expressions are homogeneous in the sense that they may only have *contained* branching. Contained branching means that at most one branch have non-trivial depth. Homogeneous data parallelism is well-suited for the SIMD model, since there is not asymptotically increase in splitting up the execution of different branches. Considering SIMD targets for data parallelism opens up a wide range of possible backends. One such target is general purpose graphics processing units.

1.1.2 General purpose graphics processing unit

General purpose graphics processing units (GPGPUs) are excellent targets for data parallel programming languages since they exhibit a large degree of available parallelism at a relatively low cost. A number of data parallel languages that targets GPGPUs exists, most noteworthy CUDA [1] and OpenCL [2]. Indeed many of the other languages targeting GPGPUs do so by first compiling to one of these two languages [3, 4]. The programming model in CUDA and OpenCL is almost identical, so what we mention about CUDA in the following sections applies to OpenCL as well.

CUDA is a programming model created by NVIDIA that supports data parallelism and runs on graphics cards. The language itself is basically C with the addition of a special class of functions called kernels. The same kernel function is executed a number of times in parallel, where each execution has its own thread and runs on the graphics card. A kernel is executed by calling the kernel function with the special parameters block dimension and grid dimension. The block dimension specifies how many times the kernel is executed in a block, and the grid dimension specifies how many blocks should be dispatched. So by multiplying the block and grid dimensions we get the potential parallel degree of a CUDA program. Each thread executes the same code as the other threads, and the only thing that differentiates them is a local context holding the thread's position in the block, and the block's position in the grid.

CUDA deserves a special mention because it is seemingly not very declarative with respect to parallelism, but in practice it is possible to specify a grid that is larger than the available number of processing units. In this case, the blocks are streamed, meaning that the CUDA runtime environment dispatches as many blocks as possible on the graphics card, and the remaining blocks are dispatched once the currently running blocks completes execution. In essence, CUDA takes a data parallel operation and "chunks" it into several steps, meaning that the potential parallel degree of a CUDA program may exceed the actual parallel degree of the running program, and the details are left to the runtime environment instead of the programmer. Therefore, there is

some degree of declarative with respect to concurrency in CUDA, and although the programmer has more control over the process of "chunking", it is not necessary (but possible) to exert this control per back-end. Another point is that the execution of a CUDA program is controlled by a single thread on the host that dispatches kernel to the GPGPU device. We therefore have a homogeneous data parallel machine with a relatively large available parallel degree, together with a machine with practically no available parallel degree controlling the first. The structure of this setting has influenced the design of our target language.

1.2 Nested data parallelism (NDP)

That a programming language can express nested data parallelism means that it can express data parallelism, and the way that data parallelism can be expressed in the language, allows nesting, such that a parallel computation is several parallel sub-computations all running in parallel. An example of a nested data parallel expression is:

$$\{\{e_0 : x \textbf{ in } e_1\} : y \textbf{ in } e_2\}$$

Similarly, in a functional setting

$$\textbf{map } (\textbf{map } f)$$

is an example of a nested data parallel expression. The total work in a parallel step or potential parallel degree is the sum of the potential parallel degree of each sub-step.

Conversely, a flat data parallel programming language is a programming language that can express data parallelism, but does not allow this kind of nesting.

As can be seen, nestedness is an inherent property of a data parallel language with the parallel apply-to-each. Consequently, nested data parallel languages are generally more expressive than flat parallel languages, and many common parallel algorithms are indeed more concise and/or more parallel, when written in a nested data parallel language [5]. At the source level, nested data parallelism is clearly more desirable than flat data parallelism, but at the compiler level the story is the opposite.

As mentioned earlier, it is usually up to the compiler to perform "chunking" as necessary in a data parallel language. "Chunking" is trivial if the data parallel operation is flat since it is just a matter of doing some work in one step, and then proceed to do the rest. In the case of nested data parallelism the process of "chunking" becomes less obvious as each sub-computation may have a different potential parallel degree. A common approach for NDP languages is to *vectorize* NDP expressions into equivalent flat data parallel expressions first, and then compile it or interpret it using a standard flat data parallel compiler or interpreter.

1.2.1 Vectorization

Lists are an important ingredient in data parallel languages. The elements of a list may be lists themselves. To emphasize that a list contains sub-lists, we can call the list a *nested* list. If a list it not nested, then all the elements are primitives, and we can refer to the list as a *flat* list. A flat list is also called a *vector*. We use the word list to refer a data structure that supports random-access.

One common way of implementing a compiler/interpreter for a nested data parallel language, is to apply *vectorization* (also called flattening) at some point. Vectorization

is a language transformation that turns a nested data parallel language expression into an equivalent flat data parallel expression. In the flat expression, all nested lists are eliminated and replaced by one or more vectors, hence the name vectorization.

Vectorization was first proposed by Guy Blelloch and realized in the programming language NESL [6–8], and has later been studied and refined by others such as Jan Prins and Daniel Palmer with the language Proteus [9–11], and Gabriele Keller, Simon Peyton Jones and Manual Chakravarty with the language Data Parallel Haskell [12–14].

The main idea of vectorization is to have two versions of every primitive operation in the language: The standard version and a lifted version, that lifts the input and output type to list-type. E.g if $(+) : \mathbf{int} * \mathbf{int} \texttt{ -> } \mathbf{int}$, then the lifted version has type $(+\hat{}) : \mathbf{[int]} * \mathbf{[int]} \texttt{ -> } \mathbf{[int]}$. Apply-to-each

$$
\begin{array}{l}
\mathbf{for}\ (x\ \mathbf{in}\ e_0)\{ \\
\quad\quad e_1 \\
\}
\end{array}
$$

is the vectorized by lifting all operations within the body e_0 and simply binding e_1 to x as it is. In the case of nested data parallelism (nested apply-to-each), the operation in the inner-most body will be lifted twice. In order to avoid having to generate lifted versions of every operation up to an arbitrary high level, a doubly-lifted operation is transformed into a singly-lifted operation by concatenating the input, applying the singly-lifted operation and partitioning the output according to the shape of the original input. This approach is made viable by a clever representation of nested lists that allows concatenation and partitioning to be cheap.

A nested vector is represented by a flat data vector with an accompanying segment descriptor that holds the nesting structure of the nested vector. As an example the nested vector

$$v = \big[[1,2,3],\ [4],\ [],\ [5,6]\big]$$

will be represented by a flat data vector

$$d = [1,2,3,4,5,6]$$

and a segment descriptor describing the length of each sub-vector

$$s_0 = [3,1,0,2]$$

This representation can be generalized to vectors of any nesting depth. For example

$$v = \Big[[]\ ,\ \ \big[[1,2,3],\ [4],\ [],\ [5,6]\big]\ ,\ \ \big[[7],\ [],\ [8,9,10]\big]\Big]$$

will be represented by

$$d = [1,2,3,4,5,6,7,8,9,10]$$

and

$$
\begin{aligned}
s_0 &= [3,1,0,2,1,0,3] \\
s_1 &= [0,4,3]
\end{aligned}
$$

Concatenation is then achieved by simply removing the top-most segment descriptor, and similarly partitioning by the shape of the original input is achieved by attaching the top segment descriptor of the original input on top of the result. A general

property of the segment descriptors is that the sum of the elements in a segment descriptor equals the length of the next segment descriptor (or data vector). Depending on the backend, vectorization is a necessary step in order to execute nested data parallel programs. Vectorization also gives work balancing basically for free, since we concatenate all sub-lists in NDP expressions.

1.2.2 Ideal cost model

Not surprisingly, data parallel programming languages seldom have any guarantees about the structure of execution in a parallel loop, and as such, data parallelism can be said to be declarative with respect to concurrency. The declarative nature of data parallelism means that the actual time and space complexity of a data parallel program may depend heavily on the architecture that the program is executed on. The time and space costs therefore becomes more difficult to reason about at the level of the source language.

The usual way to reason about time in parallel programs is to consider the two extreme cases of back-ends: A single processor machine (available parallel degree = 1), and a machine with an unbounded number of processors (available parallel degree = ∞). The time complexity on any machine between these two extremes can then be derived from these two values. The time complexity on a sequential machine is often called work and the time complexity on an unbounded parallel machine is often called steps or depth. The ideal work and step complexities can be quantified in a natural language-based cost model alongside the value semantics, providing a good compositional mental model of the time complexity for the programmer, covering any instantiation of target machine with some degree of available parallelism. In other words, the cost model is fundamentally independent from the target machine, even though the actual running time is not.

There is no usual way to reason about space in a data parallel program; the subject is not covered as much as time complexity. Most existing data parallel languages do not distinguish the space complexity on a sequential machine from the space complexity on a parallel machine[1], i.e. there is no space equivalence of work and steps giving a space complexity cost model in the two extreme case of available parallel degree = 1 and available parallel degree = ∞. One of the assertions of this thesis is that we need to make this distinction, and one of the contributions is to provide a natural language based cost model, that sufficiently quantifies the space complexity, and derives an ideal space cost given a concrete machine.

Vectorized data parallel programs without any space optimizations use many intermediate arrays, often with a size that is proportional to the degree of potential parallelism in the program. It may be safe to assume that a program is allowed to use space proportional to the degree of actual parallelism, but a space complexity proportional to the degree of potential parallelism may be prohibitive in several common cases. Since we do not want to punish the programmer for exposing too much parallelism, the space complexity should not depend on the potential degree of parallelism, but rather the degree of realized parallelism.

With the well-founded concepts of data parallelism, homogeneous data parallelism,

[1] There are some notable exception to this statement, Guy Blelloch et. al. has given a provably space efficient scheduling for nested parallelism [15], but their work only applies to fine-grained control parallelism.

NDP, vectorization and ideal cost model in hand, we proceed to present our problem statement.

1.3 Problem statement

The main question the thesis contributes to is:

> "Is it feasible in terms of space complexity to express parallel algorithms in a high-level, platform-independent language?"

By feasible we mean comparable to hand-written low-level implementations, and by high-level we mean a nested data parallel language without explicit concurrency. Besides being feasible in terms of measurable space and time, we also want the efficiency to be provable to give formal guarantees about the work-, step- and space complexity. To narrow the scope we will only consider execution on homogeneous data parallel architectures - in particular GPGPUs. Compiling nested data parallelism to GPGPU code has already been done by John Reppy and Lars Bergstrom [3], but they do not address the space complexity issue regarding potential versus realized degree of parallelism, and they do not provide a cost model. The goal of this thesis is to present a more space-efficient compilation by transparently sequentializing computations that are too parallel. Additionally in contrast to existing vectorizing compilers, we will explore the possibility of using data flow network as an internal representation of programs. This choice will hopefully alleviate formalizing complexity models as well as proving their preservation across multiple program transformations. The thesis will be about nested data parallelism, and we will give novel contributions in form of sequentialization and cost models.

As an idiomatic example, consider the following basic arithmetic expression:

$$\sum_{i=0}^{l-1} i^2$$

We will use this example extensively and refer to it as the *iota-square-sum* example. The expression is an instance of the common map-reduce pattern where the squaring corresponds to a map, and the summation corresponds to a reduction. Given a concrete machine with available parallel degree $= P$, it is certainly possible to compute this expression in $O(l/P)$ time complexity (ignoring the logarithmic depth of sum for simplicity), which is similar to what most existing NDP languages does. On the other hand, it is certainly possible to compute the result using $O(\min(P, l))$ space, but this is unlike existing NDP languages that generally require $O(l)$ space. We believe it is possible to achieve a more ideal space complexity in a NDP framework.

2 Analysis

This section is devoted to presenting the analysis that led to the choice of using streaming to perform automatic sequentialization at runtime. The analysis is based on an investigation of efficient implementations of a simple problem in low-level languages compared to a NDP implementation. Previous research has indicated that it is possible to obtain close to ideal work and step complexity of a data parallel language under reasonable assumptions [11,16,17], so the focus of the analysis is the space complexity. As described in the problem statement, NDP languages sometimes have a space complexity that is worse, than an equivalent program written in a low level data parallel language, namely in the presence of reductions. Reduction is a common occurrence in parallel algorithms, so the problem is real.

2.1 NDP language

To see why nested data parallel languages are worse in terms of space we return to the iota-square-sum example, and for simplicity, assume l is a power of 2, so $l = 2^n$

$$\sum_{i=0}^{l-1} i^2$$

Writing this expression in a nested data parallel language could look like:

$$\mathbf{sum}(\{\mathbf{sqr}(i) : i \ \mathbf{in} \ [0..l-1]\})$$

, for all i in $[0..l-1]$, square i, and sum the result.

Vectorizing this expression yields

$$\texttt{sum(sqr\^{}(iota(}l\texttt{)))}$$

where $iota(l$ generates the vector $[0..l-1]$, and `sqr^` is the vector version of squaring that takes a vector of integers and squares each element. The usual interpretation is that we *first* compute the vector `iota(`l`)` and only *after* it is computed we compute `sqr^(`x`)`. Finally, after this has computed we sum up the result. Clearly, this approach requires $O(l)$ space, since we allocate the entire vector $[0..l-1]$, followed by the entire vector $[0^2..(l-1)^2]$ before we sum the latter. Large intermediate arrays are indeed the root of the space problem for nested data parallel languages, pin-pointed by other researches before [3, 17], and a technique called *fusion* is often employed. Fusion basically means that two primitive operators are fused into a super-operator performing both operations without any intermediate results. In the context of GPGPUs fusion involves generating kernels with multiple primitive operations. This solution is not general: What if the intermediate result is used for something else? For example

$$\begin{aligned} &\mathbf{let}\ xs = [0..l-1] \\ &\mathbf{in}\ \mathbf{sum}(xs) + \mathbf{prod}(xs) \end{aligned}$$

In order for fusion to solve the space problem, both the product and the summation will have to be fused with the iota operator, which is possible, but certainly complicates kernel code generation. It is impossible to generate a single fused super-operator for an entire program.

```
// 2^(n-BLOCK_POWER) blocks, and the size of each block is
BLOCK_POWER = 8;
BLOCK_SIZE = 1<<BLOCK_POWER; // 256

// Entry point. Calls the kernel
int iota_square_sum(int n) {

  int n_blocks = 1<<(n-BLOCK_POWER) // n^2/256 = 1/256

  int ret = 0;

  int *sum = (int *)malloc(n_blocks * sizeof(int));
  int *dsum;
  cudaMalloc((void**)xs, n_blocks);
  kernel<<<1>>>(dsum);
  cudaMemcpy( sum, dsum, n_blocks*sizeof(unsigned long)
            , cudaMemcpyDeviceToHost);
  for(int j = 0; j < n_blocks; ++j)
      ret += sum[j];

  return ret;
}

// This kernel is run on 2^n threads divided evenly amongst
__global__ void kernel(int *dsum) {

  int tid = threadIdx.x;

  // iota(2^n)
  int i = tid + blockIdx.x * BLOCK_SIZE;

  // i^2
  __shared__ unsigned long res[BLOCK_SIZE];
  res[tid] = i*i;

  __syncthreads();

  //Reduction phase
#pragma unroll
  for(int i = BLOCK_POWER-1; i >= 0; --i) {
    if(tid < (1<<i)) {
      res[tid] = res[tid] + res[tid + (1<<i)];
      __syncthreads();
    }
  }

  if(tid == 0)
    dsum[blockIdx.x] = res[0];
}
```

Figure 1: CUDA implementation of iota-square-sum.

2.2 CUDA

An implementation of iota-square-sum in CUDA is much more complicated, see figure 1. Writing such a simple program with so many lines of code is the opposite of productivity. We will investigate if it is possible to generate similar code or provide interpretation with similar performance, from the high level machine-independent NDP program.

What we see is that the CUDA code consists of 1 kernel running 2^n threads divided into `N_BLOCKS = ` $2^{n-\texttt{BLOCK_POWER}}$ blocks of size `BLOCK_SIZE = ` $2^{\texttt{BLOCK_POWER}}$. The result of running the kernel is a vector of size `N_BLOCKS` that is summed on the host to obtain a single result. There are two points to note about this kernel regarding space:

1. `iota`(n) is fused into the kernel. It is computed from a thread id. When this is possible, space > realized degree of parallelism is not an issue, since CUDA will only schedule blocks to available streaming multiprocessors, so during the execution of the kernel, the space usage is not greater than the realized degree of parallelism. In other words, the CUDA version requires less space, if there is more work to be done, than processors available. This could be a hint that it is possible to achieve better space cost semantics, using CUDA as a backend, if we manage to fuse all computations into a single kernel. But unfortunately, fusing an arbitrary NDP expression into a single kernel is not possible.

2. Efficient reduction in CUDA relies on shared memory identified by the `__shared__` keyword. Shared memory is local to blocks. Each thread computes one instance of `pow`(i), and each block reduces the results of all the threads in the block to a single value. Once reduced, the result of the kernel is a vector of size `N_BLOCKS`. The space requirements for the resulting vector (which must be allocated before the kernel is launched) is therefore $2^{n-\texttt{BLOCK_POWER}}$ (e.g. $(2^n)/256$). The result of the kernel is then reduced sequentially on the CPU. Without knowing the exact numbers, chances are that space issue will only be a problem for very large values of 2^n, but asymptotically, the CUDA version is just as bad in space complexity as the NDP version - unless we add a loop on the host calling the kernel several times. What we can conclude from this is, that even though CUDA's execution model is based on streaming, the programmer still has to manually sequentialize code even simple problems, and by doing so, he must be aware of the expected size of the problem contra hardware specific parameters such as block size.

By analyzing the space complexity of a simple program, we have identified a well-known problem with traditional NDP languages. There are positive solutions to the problem in the context of explicit concurrency and multi-threading [15, 18], but to the knowledge of the author, there are no good solution for (homogeneous) data parallel back-ends yet. The solutions rely on dynamic (online) scheduling, and the need to have dynamic scheduling is further supported by the observation that we need to have problem-size specific sequentialization. Since the architecture of GPGPUs consists of a sequential host machine and a homogeneous data parallel device machine, it seems natural to investigate the possibilities of performing dynamic scheduling on the host, that schedules data parallel operations on the device, while sequentializing the problem as needed. The CUDA runtime environment already provides sequentialization on kernels in isolation, but as we have seen, this is generally not sufficient. We need

the sequentialization to happen across the boundaries of each kernel call. We therefore propose a solution based on streaming of chunks, where the host dynamically dispatches relatively small kernels operating on chunks. The chunk sizes should not be too small since we want to utilize all the available parallel degree, but not too large, since we do not want to allocate too large intermediate arrays. If it is possible to keep the intermediate arrays in order of the chunk size, and the chunk size is in order of the parallel degree, it should be possible to achieve ideal space and step complexity, given the program already has ideal work complexity.

3 Source language

In this section we describe a reference source language syntax, semantics, types and cost model. The reference language is a minimalistic toy language stripped of all unnecessary features. It lies very close to a subset of NESL both in terms of syntax, type system and ideal cost model, with one exception: The source language includes a *sequence type*, which can be though of as a high level stream. Sequences resembles lists, but unlike lists, the elements of a sequence is not expected to be available at the same time, instead the elements arrive in blocks of constant size, and sequences are processed left to right. Consequently, a sequence can only be used once, and every operation that uses a sequence, must use the whole sequence.

The sequence type and associated values and operations form the foundation of the contributions of the thesis. As we will demonstrate later, the sequences makes it possible to automatically sequentialize too parallel expressions, resulting in more space efficient execution, while hopefully maintaining ideal work and step complexity.

An actual front end that performs type inference and desugaring can be imagined, but will not be described thoroughly, although we will informally describe some simple desugaring transformations for some of the most common language constructs that we do not include in the reference

The language has some more or less serious restrictions that makes the transformation to target language more manageable at the cost of incompleteness.

- First-order: Higher-order functions in a nested data parallel language imposes many new challenges. We will postpone treatment of higher order function to future work in order to keep the focus on the problem at hand.
- No user-defined function: Adding non-recursive user-defined function is trivial, but recursive functions add many problems. This restriction is arguable the most severe, but we will argue in section 4.6 that it is possible to introduce recursion with minor overhead, given that recursion depth is logarithmic, which is almost always the case for data parallel algorithms [5].

.

3.1 Syntax

The proposed language is given by the following syntax:

$$n \in \mathbf{Z} \qquad b \in \{\mathbf{t}, \mathbf{f}\} \qquad x, y \in \mathbf{VarId}$$

$$\begin{aligned}
\mathbf{PVal} \ni a \quad &::= \quad n \mid b \mid \cdots \\
\mathbf{Exp} \ni e \quad &::= \quad \overline{a} \\
&\quad\ \mid x \\
&\quad\ \mid \mathbf{let}\ x\ \mathbf{=}\ e\ \mathbf{in}\ e \\
&\quad\ \mid \mathbf{()} \mid \mathbf{(}e\mathbf{,}\ e\mathbf{)} \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \\
&\quad\ \mid \mathbf{\{}e\ \mathbf{:}\ x\ \mathbf{in}\ e\ \mathbf{using}\ x\mathbf{,}\ldots\mathbf{,}x\mathbf{\}} \\
&\quad\ \mid o\ e
\end{aligned}$$

Furthermore we use the letters i and j for indexing and k and l for lengths, where k usually refers to small constant numbers related to expression size, and l usually refers

to large numbers related to data size.

$$i, j, l, k \in \mathbf{N}$$

We have literal constants $\overline{a}$, primitive operations o, let-bindings, pairs and apply-to-each. We extend the usual notation of apply-to-each

$$\{e_0 : x \textbf{ in } e_1\}$$

with the keyword **using**

$$\{e_0 : x \textbf{ in } e_1 \textbf{ using } x_1, \ldots, x_k\}$$

which can be read as "e_0 evaluated for every x in the sequence e_1 using the variables $x_1, \ldots, x_k$ from the environment", and the result is a sequence. In other words, we require that the use of variables from outer scope is explicitly stated in an apply-to-each. Such a list of variables is easily generated automatically by the front end, and they are made explicit in order to distribute them in the transformation phase.

The construct $o\ e$ is application. The primitive operations $o \in \mathbf{Op}$ is applied to e. The language contains a small set of primitive operations similar to other nested data parallel languages. Contrary to other nested data parallel languages, some of the operations are defined to work on sequences instead of lists. Many of the primitive operations are inherently polymorphic in their type signature. In order to deal with this, we introduce ad-hoc polymorphism by syntactically annotating polymorphic operations with type information such that we have infinitely many primitive operations, but the type signature of any operation is monomorphic. We introduce three syntactic categories on types inductively defined by:

$$\begin{aligned}
\mathbf{Typ} \ni \sigma &::= \sigma * \sigma \mid \mathbf{1} \mid \{\sigma\} \mid \tau \\
\mathbf{Typ} \supset \mathbf{CTyp} \ni \tau &::= \tau * \tau \mid \mathbf{1} \mid [\tau] \mid \pi \\
\mathbf{CTyp} \supset \mathbf{PTyp} \ni \pi &::= \mathbf{int} \mid \mathbf{bool} \mid \cdots
\end{aligned}$$

PTyp is the discrete category of primitive types. **CTyp** is the category of *concrete* types, and it introduces products $\tau * \tau$ and lists $[\tau]$. **Typ** is the category of all types, and it introduces the sequence type $\{\sigma\}$, as well as sequence products $\sigma * \sigma$.

The syntax of operations is given below:

$$\begin{aligned}
\mathbf{Op} \ni o \quad ::= \quad & \mathbf{list}_{k,\tau} \qquad k \geq 0 \\
& \mid \mathbf{append}_\sigma \\
& \mid \mathbf{iota} \\
& \mid \mathbf{elt}^{[]}_\tau \mid \mathbf{len}^{[]}_\tau \\
& \mid \mathbf{elt}^{\{\}}_\sigma \mid \mathbf{len}^{\{\}}_\sigma \\
& \mid \mathbf{zip}_{\sigma_0,\sigma_1} \\
& \mid \mathbf{concat}_\sigma \mid \mathbf{part}_\sigma \\
& \mid \mathbf{tab}_\tau \mid \mathbf{seq}_\tau \\
& \mid \mathbf{scan}_\otimes \mid \mathbf{reduce}_\otimes \\
& \mid \oplus \\
\oplus \quad ::= \quad & + \mid * \mid \texttt{<=} \mid \mathbf{not} \mid \mathbf{b2i} \mid \cdots \\
\otimes \quad ::= \quad & \mathbf{sum} \mid \mathbf{prod} \mid \mathbf{and} \mid \mathbf{or} \mid \cdots
\end{aligned}$$

Some operations, such as $\mathbf{elt}_\tau^{[]}$ and $\mathbf{elt}_\sigma^{\{\}}$, have a list version as well as a sequence version. In the cases where two versions exist, the versions are distinguished by a superscript $\mathbf{\{\}}$ for the sequence version or a $\mathbf{[]}$ for list versions. $\mathbf{tab}_\tau$ is the operation that converts a sequence to a list, and in the other direction, $\mathbf{seq}_\tau$ is the operation that converts a list to a sequence. The reason that we do not have list versions of partition, concat, zip, iota, scan and reduce is, that they are easily obtained by tabulation. Tabulating the other operations would break the cost model.

3.2 Type system

We use a type system to identify well-formed expressions. Well-formed expressions is the sub category of expressions that are expected to have an well-defined interpretation. The category of types is the same as the syntactic type category given for annotating operations, which is recalled as:

$$
\begin{aligned}
\mathbf{Typ} \ni \sigma \quad &::= \quad \sigma * \sigma \mid \mathbf{1} \mid \mathbf{\{}\sigma\mathbf{\}} \mid \tau \\
\mathbf{Typ} \supset \mathbf{CTyp} \ni \tau \quad &::= \quad \tau * \tau \mid \mathbf{1} \mid \mathbf{[}\tau\mathbf{]} \mid \pi \\
\mathbf{CTyp} \supset \mathbf{PTyp} \ni \pi \quad &::= \quad \mathbf{int} \mid \mathbf{bool} \mid \cdots
\end{aligned}
$$

Every primitive value a is assigned a primitive type π

$$
\begin{aligned}
&n : \mathbf{int} \\
&b : \mathbf{bool} \\
&\cdots
\end{aligned}
$$

Monoids $\otimes$ are assigned a single primitive type π that identifies the type of the associate operation $\oplus : \pi * \pi \texttt{ -> } \pi$ and identity element of the monoid.

$$
\begin{aligned}
&\mathbf{sum} : \mathbf{int} \\
&\mathbf{prod} : \mathbf{int} \\
&\mathbf{and} : \mathbf{bool} \\
&\mathbf{or} : \mathbf{bool} \\
&\cdots
\end{aligned}
$$

All the primitives operations and scalar operations are assigned a type signature of

the form σ_0 -> σ_1

$$
\begin{array}{ll}
\forall\tau\forall k \geq 0 \,.\, \mathbf{list}_{k,\tau} & : \tau^k \texttt{ -> } [\tau] \\
\forall\sigma \,.\, \mathbf{append}_\sigma & : \{\sigma\} * \{\sigma\} \texttt{ -> } \{\sigma\} \\
\mathbf{iota} & : \mathbf{int} \texttt{ -> } \{\mathbf{int}\} \\
\forall\tau \,.\, \mathbf{elt}^{[]}_\tau & : ([\tau] * \mathbf{int}) \texttt{ -> } \tau \\
\forall\tau \,.\, \mathbf{len}^{[]}_\tau & : [\tau] \texttt{ -> } \mathbf{int} \\
\forall\sigma \,.\, \mathbf{elt}^{\{\}}_\sigma & : \{\sigma\} * \mathbf{int} \texttt{ -> } \sigma \\
\forall\sigma \,.\, \mathbf{len}^{\{\}}_\sigma & : \{\sigma\} \texttt{ -> } \mathbf{int} \\
\forall\sigma_0\forall\sigma_1 \,.\, \mathbf{zip}_{\sigma_0,\sigma_1} & : (\{\sigma_0\} * \{\sigma_1\}) \texttt{ -> } \{\sigma_0 * \sigma_1\} \\
\forall\sigma \,.\, \mathbf{concat}_\sigma & : \{\{\sigma\}\} \texttt{ -> } \{\sigma\} \\
\forall\sigma \,.\, \mathbf{part}_\sigma & : (\{\sigma\} * \{\mathbf{bool}\}) \texttt{ -> } \{\{\sigma\}\} \\
\forall\tau \,.\, \mathbf{tab}_\tau & : \{\tau\} \texttt{ -> } [\tau] \\
\forall\tau \,.\, \mathbf{seq}_\tau & : [\tau] \texttt{ -> } \{\tau\} \\
\forall\otimes : \pi \,.\, \mathbf{scan}_\otimes & : \{\pi\} \texttt{ -> } \{\pi\} \\
\forall\otimes : \pi \,.\, \mathbf{reduce}_\otimes & : \{\pi\} \texttt{ -> } \pi \\
+ & : \mathbf{int} * \mathbf{int} \texttt{ -> } \mathbf{int} \\
* & : \mathbf{int} * \mathbf{int} \texttt{ -> } \mathbf{int} \\
\ldots &
\end{array}
$$

The informal description of the primitive operations is summarized in the table 1. The type of an expression is then given in the following inference system:

Operation	Description	Example
$\mathbf{list}_{k,\tau}$	Takes k arguments of type τ and constructs a k element list.	$\mathbf{list}_{k,\mathbf{int}}(3,8,7) = [3,8,7]$
$\mathbf{append}_{\sigma}$	Appends two sequences.	$\mathbf{append}_{\mathbf{int}}\ (\{1,2,3\},\{10,20\}) = \{1,2,3,10,20\}$
$\mathbf{elt}^{[]}_{\tau}$	Retrieves an element from a list of type $[\tau]$.	$\mathbf{elt}^{[]}_{\mathbf{int}}([3,8,7],\ 1) = 8$
$\mathbf{elt}^{\{\}}_{\sigma}$	Retrieves an element from a sequence of type $\{\sigma\}$.	$\mathbf{elt}^{[]}_{\mathbf{int}}(\{3,8,7\},\ 1) = 8$
$\mathbf{concat}_{\sigma}$	Takes a sequence of sequences and concatenates all sub-sequences into one sequence.	$\mathbf{concat}_{\mathbf{int}}(\{\{3,8\},\{7\}\}) = \{3,8,7\}$
$\mathbf{part}_{\sigma}$	Takes a sequence of type $\{\sigma\}$ and a sequence of flags and partitions the first sequence into a sequence of sequences.	$\mathbf{part}_{\mathbf{int}}(\{3,8,7\},\{\mathbf{f},\mathbf{f},\mathbf{t},\mathbf{f},\mathbf{t}\}) = \{\{3,8\},\{7\}\}$
$\mathbf{zip}_{\sigma_0,\sigma_1}$	Combines two sequences of equal length by pairing up elements with the same index.	$\mathbf{zip}_{\mathbf{int},\mathbf{int}}(\{3,8,7\},\ \{0,1,1\}) = \{(3,\ 0),(8,\ 1),(7,\ 1)\}$
$\oplus$	$\oplus$ covers all scalar operations.	$5 + 4 = 9$
iota	The ι function known from APL. Generates a consecutive sequence of integers.	$\mathbf{iota}(4) = \{0,1,2,3\}$
$\mathbf{scan}_{\otimes}$	Performs an exclusive prefix scan on a sequence given some monoid $\otimes$.	$\mathbf{scan}_{\mathbf{sum}}(\{3,8,7\}) = \{0,3,11\}$
$\mathbf{reduce}_{\otimes}$	Performs a reduction on a sequence given some monoid $\otimes$.	$\mathbf{reduce}_{\mathbf{and}}(\{\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t}\}) = \mathbf{f}$
$\mathbf{tab}_{\tau}$	Tabulates a sequence by converting it to a list.	$\mathbf{tab}_{\mathbf{int}}(\{1,2,3,4\}) = [1,2,3,4]$
$\mathbf{seq}_{\tau}$	Sequences a list by converting it to a sequence.	$\mathbf{seq}_{\mathbf{int}}([1,2,3,4]) = \{1,2,3,4\}$

Table 1: The primitive operations of the source language.

$\boxed{\Gamma \vdash e : \sigma}$

$$\frac{}{\Gamma \vdash \overline{a} : \pi}\ a : \pi \tag{T-Lit}$$

$$\frac{}{\Gamma \vdash x : \sigma}\ \Gamma(x) = \sigma \tag{T-Var}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \qquad \Gamma[x \mapsto \sigma_0] \vdash e_1 : \sigma_1}{\Gamma \vdash \textbf{let}\ x\ \textbf{=}\ e_0\ \textbf{in}\ e_1 : \sigma_1} \tag{T-Let}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash \textbf{(}e_0\textbf{,}\ e_1\textbf{)} : \sigma_0 * \sigma_1} \tag{T-Pair}$$

$$\frac{}{\Gamma \vdash \textbf{()} : \mathbf{1}} \tag{T-Unit}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0}{\Gamma \vdash o\ e_0 : \sigma_1}\ o : \sigma_0 \to \sigma_1 \tag{T-Op}$$

$$\frac{\Gamma \vdash e_1 : \textbf{\{}\sigma_1\textbf{\}} \quad [x \mapsto \sigma_1, x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k] \vdash e_0 : \sigma_0}{\Gamma \vdash \textbf{\{}e_0\ \textbf{:}\ x\ \textbf{in}\ e_1\ \textbf{using}\ x_1\textbf{,}\ldots\textbf{,}x_k\textbf{\}} : \textbf{\{}\sigma_0\textbf{\}}}\ \Gamma(x_i) = \tau_i \text{ for } i = 1..k \tag{T-Over}$$

The purpose of the **using** keyword now becomes apparent: We do not allow the body expression of an apply-to-each to use any variables from the environment bound to sequence types. All values from outer scope *must* be of concrete types τ. The reason is that sequences must only be used once. Having a sequence from outer scope inside an apply-to-each means that the body expression can *restart* the sequence a number of times, which is not allowed. If a sequence really needs to be restarted, it must be re-evaluated, which can be achieved by moving the expression that evaluates to the sequence inside the apply-to-each. It would however be possible to remove this restriction if we can infer that the length of e_1 is at most 1, which is relevant when desugaring if-then-else expressions and guarded apply-to-each. Note that we would have to ensure that the sequence is actually used elsewhere in order to ensure strictness.

It would also be possible to retain a list-version of apply-to-each, but such a construct would be redundant since it is equivalent in terms of value and asymptotic complexity to sequencing the qualifying list and tabulate the resulting sequence.

The given type system is not the complete story. It does not guarantee that a well-formed expression can actually be evaluated using constant size sequences. Consider

the expression that grabs the element n position from the end of a sequence xs:

$$\mathbf{elt}^{\{\}}_{\sigma}(xs,\ \mathbf{len}^{[]}_{\sigma}(xs - \overline{n}))$$

There is no way to stream this expression using constant space without knowing either the length of xs or the value of n in before-hand. The type system must be extended with a notion of time frames of availability of sequence elements, if we want to guarantee constant size sequences, but for now, we conclude the description of the source language. We have extended a traditional NDP language with a notion of sequences, by extending the syntax and providing a new type system.

3.3 Syntactic sugar

The language of the front end is enriched with more constructs and operations that does not appear in the source language. The new constructs and operations simplify writing expressions, but provide nothing new and are easily removed by desugaring rules. The front end language contains all constructs and operations of the source language, but operations are not annotated with types. Their types are inferred by type inference. Front end expressions e' to source language expressions e by a certain rules $e' \rightsquigarrow e$, and also for operations $o'' \rightsquigarrow o'$.

In the given rules, variables denoted with x' are variables that are selected so that they do not conflict with any other variable names, and $_$ is a wildcard variable that is never used.

Pairs are easily generalized to n-ary tuples for $n > 0$ by representing tuples as nested pairs:

$$\begin{array}{ll} (e'_0) & \rightsquigarrow e'_0 \\ (e'_0, e'_1, e'_2) & \rightsquigarrow ((e'_0,\ e'_1),\ e'_2) \\ \ldots & \end{array}$$

The following is a series of minor syntactic abbreviations:

$$\begin{array}{ll} \& & \rightsquigarrow \mathbf{iota} \\ \# & \rightsquigarrow \mathbf{len}^{\{\}} \\ [e_0, \ldots, e_{k-1}] & \rightsquigarrow \mathbf{list}_k\ (e_0, \ldots, e_{k-1}) \\ e_0[e_1] & \rightsquigarrow \mathbf{elt}^{[]}(e_0,\ e_1) \\ e_0\{e_1\} & \rightsquigarrow \mathbf{elt}^{\{\}}(e_0,\ e_1) \\ e_0 \oplus e_1 & \rightsquigarrow \oplus(e_0,\ e_1) \\ e_0 +\!\!+ e_1 & \rightsquigarrow \mathbf{append}(e_0,\ e_1) \end{array}$$

The given language is complex enough to express many standard language constructs found in other languages.

Replication of concrete values:

$$\mathbf{repl}(e'_0,\ e'_1) \rightsquigarrow \mathbf{let}\ x'\ =\ e'_0\ \mathbf{in}\ \{x' : _\ \mathbf{in}\ \&e'_1\ \mathbf{using}\ x'\}$$

with the derived type

$$\forall \tau\ .\ \mathbf{repl}_{\tau} : \tau * \mathbf{int}\ \text{->}\ \{\tau\}$$

In the front end, we use a general notation for apply-to-each

$$\{e'_0 : x_1\ \mathbf{in}\ e'_1;\ \ldots;\ x_k\ \mathbf{in}\ e'_k \mid e'_{k+1}\}$$

The general apply-to-each extends the normal apply-to-each with $k \geq 1$ qualifiers and an optional guard expression e'_{k+1}. The qualifying expression e'_1 to e'_k must have the same length, and the guard is evaluated for every tuple $(x_1, \ldots, x_k)$ in the k-ary zip of e'_1 to e'_k. When the guard evaluates to $\mathbf{t}$, the body expression e'_0 is evaluated an emits an element to the output sequence. Multiple qualifiers are replaced with a single qualifier (x' **in** e''_1) where e''_1 is a k-ary zip of $e'_1, \ldots, e'_k$ by inserting **zip** operations $k-1$ times, and x_1 to x_k are replaced in the body expression by a corresponding projection of x'. E.g.

$$\begin{array}{l}\{e'_0 \mathbf{:}\, x_1 \;\mathbf{in}\; e'_1\mathbf{;}\; x_2 \;\mathbf{in}\; e'_2 \mid e'_3\} \rightsquigarrow \\ \qquad \{e'_0[x_1 \mapsto \mathbf{fst}\; x', x_2 \mapsto \mathbf{snd}\; x'] \mathbf{:}\, x' \;\mathbf{in}\; \mathbf{zip}(e'_1, e'_2)\mathbf{;} \mid e'_3\}\end{array}$$

The guard expression is desugared by generating a sequence of one or zero elements:

$$\{e'_0 \mathbf{:}\, x \;\mathbf{in}\; e'_1 \mid e'_2\} \rightsquigarrow \mathbf{concat}(\{\{e'_0 \mathbf{:}\, _ \;\mathbf{in}\; \&(\mathbf{b2i}(e'_2))\} \mathbf{:}\, x \;\mathbf{in}\; e'_1\})$$

but this rule places x outside a map where e'_0 is inside, and consequently x must not have sequence type, which in turn implies that e' must be a stream of concrete types. The guarded apply-to-each is therefor not completely general.

Finally, once we have an singly-qualified, unguarded apply-to-each with desugared sub-expressions

$$\{e_0 \mathbf{:}\, x \;\mathbf{in}\; e_1\}$$

we desugar it to the source language apply-to-each, by finding the free variables of the desugared body expression

$$\{e_0 \mathbf{:}\, x \;\mathbf{in}\; e_1\} \to \{e_0 \mathbf{:}\, x \;\mathbf{in}\; e_1 \;\mathbf{using}\; x_1, \ldots, x_k\}$$

where $x_1, \ldots, x_k$ are the free variables of e_0.

The derived type of the general front end apply-to-each is then

$$\frac{\Gamma \vdash e_1 : \{\sigma_1\} \quad \cdots \quad \Gamma \vdash e_k : \{\sigma_k\} \quad \Gamma' \vdash e_{k+1} : \mathbf{bool} \quad \Gamma' \vdash e_0 : \tau_0}{\Gamma \vdash e_0 \mathbf{:}\, x_1 \;\mathbf{in}\; e_1\mathbf{;} \;\ldots\mathbf{;}\; x_k \;\mathbf{in}\; e_k \mid e_{k+1}\} : \{\tau_0\}} \; (*)$$

$$(*)\; \Gamma' = \Gamma^c[x_1 \mapsto \sigma_1, \ldots, x_k \mapsto \sigma_k]$$

where Γ^c is Γ restricted to the concrete types τ. I.e. all sequence types are filtered out.

The front end also has an if-then-else construct. It can be desugared in a similar way to desugaring guarded apply-to-each construct, by generating a sequence of length zero or one.

$$\begin{array}{l}\mathbf{if}\; e'_0 \;\mathbf{then}\; e'_1 \;\mathbf{else}\; e'_2 \\ \quad \rightsquigarrow \mathbf{let}\; x' \;=\; \mathbf{b2i}(e'_0) \;\mathbf{in}\; (\{e'_1 \mathbf{:}\, _ \;\mathbf{in}\; \&x'\} +\!\!+ \{e'_2 \mathbf{:}\, _ \;\mathbf{in}\; \&(\overline{1} - (x'))\})\{\overline{0}\}\end{array}$$

Using apply-to-each around the branch expressions e'_1 and e'_2 has the immediate implication that we cannot type an if-then-else in a context where the free variables of e'_1 and e'_2 have sequence type. The if-then-else construct is therefor not completely general. The derived type is

$$\frac{\Gamma \vdash e'_0 : \mathbf{bool} \qquad \Gamma^c \vdash e'_1 : \sigma \qquad \Gamma^c \vdash e'_2 : \sigma}{\Gamma \vdash \mathbf{if}\; e_0 \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 : \sigma}$$

As we observed, both if-then-else and guarded apply-to-each place unwanted restrictions on type contexts. These restriction could in principle be removed, if we use the information that an apply-to-each over a sequence of 0 or 1 elements, are allowed to use sequences from outer scope - the sequences from outer scope do not need to be restarted since we only need them 0 or 1 time. The other way around, if we can show that a sequence has at most B elements, where B is the block size we use for sequences, we can allow expressions to use it in an apply-to-each over a sequence of *any* length - restarting a 1-block sequence can be done by simply keeping the block in memory. By appropriate size analysis, and by making an effort to preserve strictness, we can therefore make the type system less restrictive, but this is left for future work.

3.4 Big-step semantics

The interpretation of an expression amounts to evaluating an expression to a value. The discrete category **PVal** of primitive values a has already been defined. An intuitive way to define the category of values is[2]:

$$\begin{aligned} \mathbf{Val} \ni v \quad &::= \quad \mathbf{()} \mid \mathbf{(}v\mathbf{,}\ v\mathbf{)} \mid \mathbf{\{}v, \dots, v\mathbf{\}} \mid c \\ \mathbf{Val} \supset \mathbf{CVal} \ni c \quad &::= \quad \mathbf{()} \mid \mathbf{(}c\mathbf{,}\ c\mathbf{)} \mid \mathbf{[}c, \dots, c\mathbf{]} \mid a \end{aligned}$$

In other words, sequence values are defined exactly the same way as list values. But as we shall see, this definition of sequences is not very representative of what an actual implementation could look like, but the purpose of defining the semantic and cost model of the source language the representation given by **Val** is advantageous.

Since we do not have recursion in the language, we do not need to worry about non-termination in the semantics. Consequently, there is no purpose in distinguishing runtime error from non-termination as is necessary in some settings. Runtime error of a well-typed expression in the big-step semantics will manifest itself as undefined. I.e. there will be no derivations for expression with runtime error.

Just like every well-formed expression has a type, every well-formed value has a type. We require that all list and sequences are homogeneous. The type of a value is then given by the following small inference system:

[2]Note that values are not a subset of expressions, mainly due to sequence values.

$\boxed{\vdash v : \sigma}$

$$\frac{}{\vdash a : \pi}\ a : \pi \qquad \text{(TV-Lit)}$$

$$\frac{}{\vdash \mathbf{()} : \mathbf{1}} \qquad \text{(TV-Unit)}$$

$$\frac{\vdash v_0 : \sigma_0 \qquad \vdash v_1 : \sigma_1}{\vdash \mathbf{(}v_0\mathbf{,}\ v_1\mathbf{)} : \sigma_0 * \sigma_1} \qquad \text{(TV-Pair)}$$

$$\frac{\vdash c_0 : \tau \qquad \cdots \qquad \vdash c_{l-1} : \tau}{\vdash \mathbf{[}c_0, \dots, c_{l-1}\mathbf{]} : \mathbf{[}\tau\mathbf{]}} \qquad \text{(TV-List)}$$

$$\frac{\vdash v_0 : \sigma \qquad \cdots \qquad \vdash v_{l-1} : \sigma}{\vdash \mathbf{\{}v_0, \dots, v_{l-1}\mathbf{\}} : \mathbf{\{}\sigma\mathbf{\}}} \qquad \text{(TV-Seq)}$$

We will now introduce meta-notation for sequences. The meta-annotation

$$(\Phi(i))_{i=n_0}^{n_1}$$

expands to the notation

$$\Phi(n_0), \cdots, \Phi(n_1)$$

where Φ is notation containing the meta-notational placeholder i, and $\Phi(n)$ is the substitution of n for i. We also define the simpler form $\overrightarrow{\Phi}^k$ which expands to $\Phi_0, \dots, \Phi_{k-1}$, i.e. subscripts.

E.g.

$$\frac{(e_i \downarrow v_i)_{i=0}^{2}}{e \downarrow \mathbf{\{}\overrightarrow{v}^3\mathbf{\}}}$$

expands to

$$\frac{e_0 \downarrow v_0 \qquad e_1 \downarrow v_1 \qquad e_2 \downarrow v_2}{e \downarrow \mathbf{\{}v_0, v_1, v_2\mathbf{\}}}$$

The semantics of the reference language is given as big-steps. The big-step semantics are short and clear, and therefore provides a good point of reference for other, more technical semantics that we will be given later. However, the big-step semantics do not directly admit any efficient implementations.

Define a value store

$$\rho ::= [x_0 \mapsto v_0, \dots, x_{k-1} \mapsto v_{k-1}]$$

The value semantics of a source language expression is then given by the following big step derivation rules:

$\boxed{\rho \vdash e \downarrow v}$

$$\rho \vdash \frac{}{\overline{a} \downarrow a} \tag{Lit}$$

$$\frac{[v_0/x]e_0 \downarrow v_0 \qquad \rho[x \mapsto v_0] \vdash e_1 \downarrow v_1}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ e_0\ \textbf{in}\ e_1 \downarrow v_1} \tag{Let}$$

$$\frac{}{\rho \vdash \textbf{()} \downarrow \textbf{()}} \tag{Unit}$$

$$\frac{\rho \vdash e_0 \downarrow v_0 \qquad \rho \vdash e_1 \downarrow v_1}{\rho \vdash \textbf{(}e_0\textbf{,}\ e_1\textbf{)} \downarrow \textbf{(}v_0\textbf{,}\ v_1\textbf{)}} \tag{Pair}$$

$$\frac{\rho \vdash e_0 \downarrow v_0}{\rho \vdash o\ e_0 \downarrow v}\ [\![o]\!]\, v_0 = v \tag{Op}$$

$$\frac{\rho \vdash e_1 \downarrow \textbf{\{}v_0, \ldots, v_{l-1}\textbf{\}} \qquad \left((\rho \upharpoonright \{x_1, \ldots, x_k\})[x \mapsto v_i] \vdash e_0 \downarrow v_i'\right)_{i=0}^{l}}{\rho \vdash \textbf{\{}e_0\ \textbf{:}\ x\ \textbf{in}\ e_1\ \textbf{using}\ x_1, \ldots, x_k\textbf{\}} \downarrow \textbf{\{}v_0', \ldots, v_{l-1}'\textbf{\}}} \tag{Over}$$

where $\rho \upharpoonright X$ is the restriction of ρ to the variables in X.

Assuming type correctness, the semantics of operations are as follows:

$[\![\mathbf{list}_{\tau,k}]\!]\,(c_0,\ldots,c_{k-1}) = [c_0,\ldots,c_{k-1}]$

$[\![\mathbf{iota}]\!]\,n = \{0,\ldots,n-1\} \qquad (n \geq 0)$

$\left[\!\!\left[\mathbf{elt}^{[]}_{\tau}\right]\!\!\right]([\overrightarrow{c}^{\,l}],\, n) = c_n \qquad (\leq n < l)$

$\left[\!\!\left[\mathbf{elt}^{\{\}}_{\sigma}\right]\!\!\right](\{\overrightarrow{v}^{\,l}\},\, n) = v_n \qquad (\leq n < l)$

$\left[\!\!\left[\mathbf{len}^{[]}_{\tau}\right]\!\!\right][c_0,\ldots,c_{l-1}] = l$

$\left[\!\!\left[\mathbf{len}^{\{\}}_{\sigma}\right]\!\!\right]\{v_0,\ldots,v_{l-1}\} = l$

$[\![\mathbf{concat}_{\sigma}]\!]\,\{\,\left(\{\overrightarrow{v_i}^{\,l_i}\}\right)_{i=0}^{l'}\} = \{\,\left(\overrightarrow{v_i}^{\,l_i}\right)_{i=0}^{l'}\}$

$[\![\mathbf{part}_{\sigma}]\!]\,(\{\overrightarrow{v}^{\,l}\},\, \{\,\left(\overrightarrow{\mathbf{f}}^{\,l'_i},\mathbf{t}\right)_{i=0}^{l''}\}) = \{\,\left(\{\,(v_k)_{k=\sum_{i=0}^{j-1} l'_i}^{\sum_{i=0}^{j} l'_i}\}\right)_{j=0}^{l''}\} \qquad (\bigwedge_{i=0}^{k-1}(n_i \geq 0) \wedge \sum_{i=0}^{k-1} n_i = l)$

$[\![\mathbf{zip}_{\sigma_0,\sigma_1}]\!]\,(\{\overrightarrow{v}^{\,l}\},\, \{\overrightarrow{v'}^{\,l'}\}) = \{\,((v_i,\, v'_i))_{i=0}^{l}\}$

$[\![\mathbf{scan}_{\otimes}]\!]\,\{a_0,\ldots,a_{l-1}\} = \{\,\bigotimes_{i=0}^{-1} a_i,\ldots,\bigotimes_{i=0}^{l-2} a_i\}$

$[\![\mathbf{reduce}_{\otimes}]\!]\,\{a_0,\ldots,a_{l-1}\} = \bigotimes_{i=0}^{l-1} a_i$

$[\![+_{\mathbf{int}}]\!]\,(n_0,\, n_1) = n_0 + n_1$

$[\![\mathbf{tab}_{\tau}]\!]\,\{\overrightarrow{c}^{\,l}\} = [\overrightarrow{c}^{\,l}]$

$[\![\mathbf{seq}_{\tau}]\!]\,[\overrightarrow{c}^{\,l}] = \{\overrightarrow{c}^{\,l}\}$

As can be observed above, non of the primitive operation use the type annotation their interpretation. The need to have ad-hoc polymorphism arises in the transformation to target language where the same operation with different type annotations are transformed to different target language expressions.

3.4.1 Properties

We want to show that an interpretation of a well-formed expression produces a value of the same type as the expression. In order to show this we need to define the type of a typing context:

$\boxed{\vdash \rho : \Gamma}$

$$\frac{\vdash v_0 : \tau_0 \qquad \cdots \qquad \vdash v_{k-1} : \tau_{k-1}}{\vdash [x_0 \mapsto v_0,\ldots,x_{k-1} \mapsto v_{k-1}] : [x_0 \mapsto \tau_0,\ldots,x_{k-1} \mapsto \tau_{k-1}]} \qquad \text{(T-Env)}$$

Type preservation Evaluation of an expression results in a value of the same type as the expression.

If $\Gamma \vdash e : \tau$
and $\vdash \rho : \Gamma$

and $\rho \vdash e \downarrow v$
then $\vdash v : \tau$.
The proof is a simple induction on the syntax of e.

3.5 Cost semantics

Since efficiency is a major concern, the meaning of a program is not confined to the value it produces through evaluation. The time that it takes to evaluate it, and how much space it requires is also of interest and is part of the meaning in some sense - We want to say that two expressions are intensionally different if they produce the same result, but one runs slower than the other or requires more space. In order to say anything about the time and the space of an evaluation, we first have to define a cost model, and in order for this thesis to have any value, it is important that the cost model is acceptable.

The actual time required when executing a program written in a high-level language is seldom possible to determine based on the program alone. Data parallelism means that the number of available processing units in the execution environment will vastly influence the running time. In order to give a machine-independent language-based cost model the time is quantified as two values: Total work W and steps D (also called depth). W is the total number of basic operations required to run a program. D is the number of steps where each data parallel operation is assumed to happen in a single step. Thus the work signifies the running time on a completely sequential machine, and the depth signifies the running time on a machine with an unbounded available parallel degree or at least as much as the potential parallel degree. This distinction is common for parallel algorithms.

We present a novel quantification of space in a similar way. We quantify space in two values, and we can them *Step space* N and *work space* M. N is the amount of space required for evaluating an expression assuming an unbounded degree of available parallelism. M is the space required assuming a completely sequential target machine. M and N are thus the space analogous of work and steps. M is the interesting quantity since this is the space complexity that should be obtainable by evaluating an expression with streaming. N gives the space complexity of usual evaluation without streaming, but at the same time, it is also the complexity in a streaming model if the available parallel degree exceeds the potential. In the following cost model, it is easy to see that work space M is always less then or equal to step space N, while on the other hand work W is always equal to or greater than the number of steps D. The notations of complexity quantities are summarized in the following table where P is the available parallel degree.

	$P = 1$		$P = \infty$
Time	W	$\geq$	D
Space	M	$\leq$	N

As demonstrated by Blelloch et. al., almost all operations can be implemented as a combination of the scalar vector operations, scan, vector gather and vector scatter which all have $O(\log P)$ depth complexity. It is therefore valid to assume that the step complexity of all operations are 1 and multiply the logarithmic term afterward [19]. The actual running can be derived

$$T = O(W/P + D \log P)$$

The actual space complexity is can be derived as

$$S = O(\min(PM, N))$$

We define a complexity quadruple as

$$\mathbf{Cx} \ni \omega = \langle W, D, M, N \rangle$$
$$W \in \mathbf{N} \qquad D \in \mathbf{N} \qquad M \in \mathbf{N} \qquad N \in \mathbf{N}$$

We also define two monoids $\boxdot$ and $\boxtimes$ on $\mathbf{Cx}$ both with the identity element $\langle 0, 0, 0, 0 \rangle$.

The first monoid $\boxdot$, is the sequential addition of complexities:

$$\boxdot : \mathbf{Cx} \times \mathbf{Cx} \to \mathbf{Cx}$$
$$\boxdot = (+) \times (+) \times \max \times \max$$

Sequential addition means that the computations follows each other linearly in time. Work and steps are therefore added, while space is maxed; we can safely assume that the space of the first complexity is deallocated before the next.

The second monoid $\boxtimes$ is parallel addition of complexities:

$$\boxtimes : \mathbf{Cx} \times \mathbf{Cx} \to \mathbf{Cx}$$
$$\boxtimes = (+) \times \max \times \max \times (+)$$

Parallel addition means that the computations are parallel in time. The steps are therefore maxed, and N is added, since the space is required at the same time. Work and M is indifferent compared to sequential addition, which is only natural considering that there is no parallelism to exploit in a completely sequential target machine. The definition of the two monoids complements the inverse relationship between space and time.

We also define and *add-space* operation on complexities as

$$\boxplus : \mathbf{Cx} \times (\mathbf{N} \times \mathbf{N}) \to \mathbf{Cx}$$
$$\langle W, D, M, N \rangle \boxplus \langle \Delta M, \Delta N \rangle = \langle W, D, M + \Delta M, N + \Delta N \rangle$$

The purpose is to require that a certain amount of space must be allocated.

Every value has a size defined by the size function $|\cdot|$. The size is given both in work space and step space. Most values have the same size in work and step space, but not sequences. The step space of sequence is large (sum), while the work space is small (max). If there is only one processor, only one element of a sequence should reside in memory at any given time, while if there is an unbounded number of processors, the entire sequence is allowed to reside in memory at the same time.

$$
\begin{aligned}
|()| &= \langle 0, 0 \rangle \\
|a| &= \langle 1, 1 \rangle \\
|(v_0, v_1)| &= \langle M_0 + M_1, N_0 + N_1 \rangle \\
&\quad \text{where } |v_0| = \langle M_0, N_0 \rangle \\
&\quad \phantom{\text{where }} |v_1| = \langle M_1, N_1 \rangle \\
|[\vec{c}^{\,k}]| &= \left\langle \textstyle\sum_{i=0}^{k-1} M_i, \sum_{i=0}^{k-1} N_i \right\rangle \\
&\quad \text{where } |v_i| = \langle M_i, N_i \rangle \quad \text{for } i \in \{0..k-1\} \\
|\{\vec{v}^{\,k}\}| &= \left\langle \max_{i=0}^{k-1} M_i, \textstyle\sum_{i=0}^{k-1} N_i \right\rangle \\
&\quad \text{where } |v_i| = \langle M_i, N_i \rangle \quad \text{for } i \in \{0..k-1\}
\end{aligned}
$$

The size of a value gives an asymptotic measure of how much space it requires.

We can now proceed to define a language-based cost model of the source language along side the value semantics by the judgment

$$\rho \vdash e \downarrow \langle v, \omega \rangle$$

The derivation rules are exactly the same as the big-step rules given in the previous section for values v, but we now include the complexity ω required to evaluate v. The space complexity in ω includes the size of the result v.

$$\boxed{\rho \vdash e \downarrow \langle v, \omega \rangle}$$

$$\frac{}{\rho \vdash \overline{a} \downarrow \langle a, \langle 0,0,1,1 \rangle \rangle} \qquad \text{(C-LIT)}$$

$$\frac{}{\rho \vdash x \downarrow \langle v, \langle 1,1,0,0 \rangle \rangle}\ \rho(x) = v \qquad \text{(C-VAR)}$$

$$\frac{\rho \vdash e_0 \downarrow \langle v_0, \omega_0 \rangle \qquad \rho[x \mapsto v_0] \vdash e_1 \downarrow \langle v_1, \omega_1 \rangle}{\rho \vdash \mathbf{let}\ x\ =\ e_0\ \mathbf{in}\ e_1 \downarrow \langle v_1, \omega_0 \boxdot (\omega_1 \boxplus |v_0|) \rangle} \qquad \text{(C-LET)}$$

$$\frac{}{\rho \vdash \mathbf{()} \downarrow \langle \mathbf{()}, \langle 0,0,0,0 \rangle \rangle} \qquad \text{(C-UNIT)}$$

$$\frac{\rho \vdash e_0 \downarrow \langle v_0, \omega_0 \rangle \qquad \rho \vdash e_1 \downarrow \langle v_1, \omega_1 \rangle}{\rho \vdash \mathbf{(}e_0\mathbf{,}\ e_1\mathbf{)} \downarrow \langle \mathbf{(}v_0\mathbf{,}\ v_1\mathbf{)}, (\omega_0 \boxplus |v_1|) \boxdot (\omega_1 \boxplus |v_0|) \rangle} \qquad \text{(C-PAIR)}$$

$$\frac{\rho \vdash e_0 \downarrow \langle v_0, \omega_0 \rangle}{\rho \vdash o\ e_0 \downarrow \langle v, \omega_0 \boxdot \langle W, 1, |v|, |v| \rangle \rangle}\ [\![o]\!]\ v_0 = \langle v, W \rangle \qquad \text{(C-OP)}$$

$$\frac{\rho \vdash e_1 \downarrow \left\langle \mathbf{\{}\overrightarrow{v}^{l}\mathbf{\}}, \omega_1 \right\rangle \qquad \left((\rho \upharpoonright \overrightarrow{x}^{k})[x \mapsto v_i] \vdash e_0 \downarrow \langle v'_i, \omega'_i \rangle\right)_{i=0}^{l}}{\rho \vdash \mathbf{\{}e_0\ \mathbf{:}\ x\ \mathbf{in}\ e_1\ \mathbf{using}\ \overrightarrow{x}^{k}\mathbf{\}}\tau \downarrow \left\langle \mathbf{\{}v'_0\mathbf{,} \ldots \mathbf{,}v'_{k-1}\mathbf{\}}, \omega_1 \boxdot \left(\boxtimes_{i=0}^{l-1} (\omega'_i \boxplus |v'_i|)\right) \right\rangle} \qquad \text{(C-OVER)}$$

The cost model of a let binding $\mathbf{let}\ x\ =\ e_0\ \mathbf{in}\ e_1$ where evaluation of e_0 yields v_0 with complexity $\omega_0 = \langle W_0, D_0, M_0, N_0 \rangle$ and evaluation of e_1 has complexity $\omega_1 = \langle W_1, D_1, M_1, N_1 \rangle$ gives the total complexity

$$\langle W, D, M, N \rangle = \omega_0 \boxdot (\omega_1 \boxplus |v_0|)$$

that is,

$$
\begin{aligned}
W &= W_0 + W_1 \\
D &= D_0 + D_1 \\
M &= \max(M_0, M_1 + |v_0|) \\
N &= \max(N_0, N_1 + |v_0|)
\end{aligned}
$$

In other words e_0 and e_1 are completely sequential as expected, and v_0 is live in the entire evaluation of e_1 which is not a preposterous assumption.

Pairs are in principle parallelizable since there is not dependencies between the two expression, but since the language is data parallel, we do not provide parallelism through the pair construct. Instead the programmer should use the apply-to-each construct. We define the evaluation of pairs to be completely sequential, and the cost model of a pair $\mathbf{(}e_0\mathbf{,}\ e_1\mathbf{)} \downarrow \langle\mathbf{(}c_0\mathbf{,}\ c_1\mathbf{)}, \omega\rangle$ is then similar to that of the let-binding, but with a slight difference - the two sub-expressions are symmetric:

$$\langle W, D, M, N\rangle = (\omega_0 \boxplus |v_1|) \boxdot (\omega_1 \boxplus |v_0|)$$

Expanding the complexities we get

$$
\begin{aligned}
W &= W_0 + W_1 \\
D &= D_0 + D_1 \\
M &= \max(M_0 + |v_1|, M_1 + |v_0|) \\
N &= \max(N_0 + |v_1|, N_1 + |v_0|)
\end{aligned}
$$

This is a slight overestimate in cases where the value v_1 (or v_0) may be stored in space deallocated after evaluation of e_0 (or v_1), but a more precise model would be more complicated and impose an ordering on the evaluation of pairs.

The cost model of an operation $o\ e_0$ where e_0 evaluates to v_0 with complexity $\omega_0 = \langle W_0, D_0, M_0, N_0\rangle$ and the meaning of the operation is $[\![o]\!]\ \ v_0 = \langle v_1, W_1\rangle$ is given by:

$$\langle W, D, M, N\rangle = \omega_0 \boxdot \langle W, 1, |v_1|, |v_1|\rangle$$

where

$$
\begin{aligned}
W &= W_0 + W_1 \\
D &= D_0 + 1 \\
M &= \max(M_0, |v_1|) \\
N &= \max(N_0, |v_1|)
\end{aligned}
$$

Every operation takes one step since the logarithmic term is multiplied in the derived complexity, and every operation require no additional space internally, which is asymptotically defensible, since either the input or output will pay for the space of the operation. In other words, all operations use asymptotic space linear in either the input or output size.

The cost model of an apply-to-each $\mathbf{\{}e_0 \mathbf{:}\ x\ \mathbf{in}\ e_1\ \mathbf{using}\ x_1, \ldots, x_k\mathbf{\}}$ where the evaluation of e_1 yields $\mathbf{\{}v'_0, \ldots, v'_{l-1}\mathbf{\}}$ with complexity $\omega_1 = \langle W_1, D_1, M_1, N_1\rangle$ and the evaluation of e_0 for $i = 0..l-1$ with x mapped to v'_i has the complexities $\omega'_i = \langle W'_i, D'_i, M'_i, N'_i\rangle$

gives the total complexity

$$\omega_1 \boxdot \left(\boxtimes_{i=0}^{l-1} (\omega'_i \boxplus |v'_i|) \right)$$

where

$$\begin{aligned} W &= W_1 + \textstyle\sum_i W'_i \\ D &= D_1 + \max_i W'_i \\ M &= \max(M_1, \max_i(M'_i + |v'_i|)) \\ N &= \max(M_1, \textstyle\sum_i (M'_i + |v'_i|)) \end{aligned}$$

In words, the work is summed up while the steps of the different evaluations of the body are maxed, all as expected. The more interesting part is the space complexities. Staring with M, consider the inner max over i of $M'_i + |v'_i|$. This is the space required to evaluate e_0 for different values of x on a sequential machine. The space is maxed which means that each v'_i is deallocated before the next v'_{i+1} enters memory. In order to satisfy the cost model e_1 must therefore be streamed. N is similar to M only the inner max is turned into a sum. All the space used for the evaluation of e_0 for $i = 0..l-1$ must be allocated at the same time.

The operation semantics

$$[\![o]\!]\, v_0 = \langle v, W \rangle \qquad (guard)$$

is given in the following table 2.

Notice that the work complexity on length and elt operation on sequences is l, where l is the length of the sequence. This differs from the list versions that are constant time operations. The reason is that we must process the entire sequence in order to get the length, and similarly in the worst case ($i = l - 1$), we must process the entire sequence to get element i.

This concludes the cost semantics of the source language. The cost model reflects the ideal costs that we want to obtain. The purpose of the high-level cost model is to be able to compare cost semantics of the eventual streaming execution semantics with an ideal cost model, such that we may show that every expression asymptotically uses the same amount of work, steps and space.

3.6 Examples

Iota-square-sum example

$$\sum_{i=0}^{l-1} i^2$$

can be expressed in the front end language as

$$\mathbf{sum(\{pow(}x\mathbf{,}\ \overline{2}\mathbf{)\ :}\ x\ \mathbf{in}\ \&\overline{l}\mathbf{\})}$$

Desugaring this expression yields

$$e = \mathbf{reduce_{sum}(\{pow(}x\mathbf{,}\ \overline{2}\mathbf{)\ :}\ x\ \mathbf{in\ iota(}\overline{l}\mathbf{)\ using\ \})}$$

o	v_0	*guard*	$[\![o]\!]\, v_0 = \langle v, W \rangle$	
			v	W
$\mathbf{list}_{\tau,k}$	$(c_0, \ldots, c_{k-1})$		$[c_0, \ldots, c_{k-1}]$	k
$\mathbf{iota}$	n	$n \geq 0$	$\{0, \ldots, n-1\}$	n
$\mathbf{elt}^{[]}{}_{\tau}$	$([\overrightarrow{c}^{\,l}],\, n)$	$0 \leq n < l$	c_n	1
$\mathbf{elt}^{\{\}}{}_{\sigma}$	$(\{\overrightarrow{v}^{\,l}\},\, n)$	$0 \leq n < l$	v_n	l
$\mathbf{len}^{[]}{}_{\tau}$	$[c_0, \ldots, c_{l-1}]$		l	1
$\mathbf{len}^{\{\}}{}_{\sigma}$	$\{v_0, \ldots, v_{l-1}\}$		l	l
$\mathbf{concat}_{\sigma}$	$\{\, \left(\{\overrightarrow{v_i}^{\,l_i}\}\right)_{i=0}^{l'} \}$		$\{\, \left(\overrightarrow{v_i}^{\,l_i}\right)_{i=0}^{l'} \}$	$\sum_{i=0}^{k-1} l_i$
$\mathbf{part}_{\sigma}$	$(\{\overrightarrow{v}^{\,l}\},\, \{\, \left(\overrightarrow{\mathbf{f}}^{\,l'_i}, \mathbf{t}\right)_{i=0}^{l''} \})$	$\bigwedge_{i=0}^{k-1} (n_i \geq 0) \wedge \sum_{i=0}^{k-1} n_i = l$	$\{\, \left(\{\, (v_k)_{k=\sum_{i=0}^{j-1} l'_i}^{\sum_{i=0}^{j} l'_i} \}\right)_{j=0}^{l''} \}$	l
$\mathbf{zip}_{\tau_0,\tau_1}$	$(\{\overrightarrow{v}^{\,l}\},\, \{\overrightarrow{v'}^{\,l'}\})$	$l = l'$	$\{\, ((v_i,\, v'_i))_{i=0}^{l} \}$	l
$\mathbf{scan}_{\otimes}$	$\{a_0, \ldots, a_{l-1}\}$		$\{\bigotimes_{i=0}^{-1} a_i, \ldots, \bigotimes_{i=0}^{l-2} a_i\}$	l
$\mathbf{reduce}_{\otimes}$	$\{a_0, \ldots, a_{l-1}\}$		$\bigotimes_{i=0}^{l-1} a_i$	l
$+$	$(n_0,\, n_1)$		$n_0 + n_1$	1
$\mathbf{tab}_{\tau}$	$\{\overrightarrow{c}^{\,l}\}$		$[\overrightarrow{c}^{\,l}]$	l
$\mathbf{seq}_{\tau}$	$[\overrightarrow{c}^{\,l}]$		$\{\overrightarrow{c}^{\,l}\}$	l

Table 2: The primitive operation semantics and costs.

The type of the expression is **int** in the empty context [], which can be seen by the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash \bar{l} : \mathbf{int}}}{[] \vdash \mathbf{iota}(\bar{l}) : \{\mathbf{int}\}}\ \mathbf{iota} : \mathbf{int} \text{ -> } \{\mathbf{int}\} \qquad \dfrac{\mathcal{T}_1}{[x \mapsto \mathbf{int}] \vdash \mathbf{pow}(x,\ \bar{2}) : \mathbf{int}}}{\{\mathbf{pow}(x,\ \bar{2})\ \mathbf{:}\ x\ \mathbf{in}\ \mathbf{iota}(\bar{l})\ \mathbf{using}\ \} : \{\mathbf{int}\}}}{[] \vdash \mathbf{reduce_{sum}}(\{\mathbf{pow}(x,\ \bar{2})\ \mathbf{:}\ x\ \mathbf{in}\ \mathbf{iota}(\bar{l})\ \mathbf{using}\ \}) : \mathbf{int}}\ \mathbf{reduce_{sum}} : \{\mathbf{int}\} \text{ -> } \mathbf{int}$$

where

$$\mathcal{T}_1 = \dfrac{\dfrac{\dfrac{}{[x \mapsto \mathbf{int}] \vdash x : \mathbf{int}} \quad \dfrac{}{[x \mapsto \mathbf{int}] \vdash \bar{2} : \mathbf{int}}}{[x \mapsto \mathbf{int}] \vdash (x,\ \bar{2}) : \mathbf{int} * \mathbf{int}}}{[x \mapsto \mathbf{int}] \vdash \mathbf{pow}(x,\ \bar{2}) : \mathbf{int}}\ \mathbf{pow} : \mathbf{int} * \mathbf{int} \text{ -> } \mathbf{int}$$

The value and cost semantics in the empty store [] are $\left\langle \sum_{i=0}^{l-1} i^2, \langle 4l, 4, 3, 3l \rangle \right\rangle$ which can be interpreted as: A sequential machine requires $O(l)$ time and $O(1)$ space, while an unbounded parallel machine requires $O(1)$ time and $O(l)$ space, the derived complexities for a concrete PRAM machine with P processors are

$$T = O(l/P + \log P)$$

for time and

$$S = O(\min(P, l))$$

for space. Which is exactly as desired. The the value and complexity is given by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[] \vdash \bar{l} \downarrow \langle l, \langle 0,0,1,1 \rangle \rangle}}{[] \vdash \mathbf{iota}(\bar{l}) \downarrow \langle \{0, \dots, l-1\}, \langle l, 1, l, l \rangle \rangle}(2) \qquad \left(\dfrac{\mathcal{E}_i}{[x \mapsto i] \vdash \mathbf{pow}(x,\ \bar{2}) \downarrow \langle i^2, \langle 2,2,2,2 \rangle \rangle} \right)_{i=0}^{l-1}}{\{\mathbf{pow}(x,\ \bar{2})\ \mathbf{:}\ x\ \mathbf{in}\ \mathbf{iota}(\bar{l})\ \mathbf{using}\ \} \downarrow \langle \{0^2, \dots (l-1)^2\}, \langle 3l, 3, 3, 3l \rangle \rangle}}{[] \vdash \mathbf{reduce_{sum}}(\{\mathbf{pow}(x,\ \bar{2})\ \mathbf{:}\ x\ \mathbf{in}\ \mathbf{iota}(\bar{l})\ \mathbf{using}\ \}) \downarrow \left\langle \sum_{i=0}^{l-1} i^2, \langle 4l, 4, 3, 3l \rangle \right\rangle}(1)$$

$$\mathcal{E}_i = \dfrac{\dfrac{\dfrac{}{[x \mapsto i] \vdash x \downarrow \langle i, \langle 1,1,0,0 \rangle \rangle} \quad \dfrac{}{[x \mapsto \mathbf{int}] \vdash \bar{2} \downarrow \langle 2, \langle 0,0,1,1 \rangle \rangle}}{[x \mapsto i] \vdash (x,\ \bar{2}) \downarrow \langle (i,\ 2), \langle 1,1,2,2 \rangle \rangle}}{[x \mapsto i] \vdash \mathbf{pow}(x,\ \bar{2}) \downarrow \langle i^2, \langle 2,2,2,2 \rangle \rangle}(3)$$

$$(1)\ [\![\mathbf{reduce_{sum}}]\!]\ \ \{0, \dots, (l-1)^2\} = \left\langle \sum_{i=0}^{l-1} i^2, l \right\rangle$$

$$(2)\ [\![\mathbf{iota}]\!]\ \ l = \langle \{0, \dots l-1\}, l \rangle$$

$$(3)\ [\![\mathbf{pow}]\!]\ \ (i,\ 2) = \left\langle i^2, 1 \right\rangle$$

Square matrix-matrix multiplication $n \times n$ square matrix-matrix multiplication can be written in the front end as

$$
\begin{array}{l}
\mathbf{\{} \\
\quad \mathbf{\{} \\
\qquad \mathbf{sum(\{} \\
\qquad A[i][k] \; \mathbf{*} \; B[k][j] \\
\qquad \mathbf{:}\; k \;\mathbf{in}\; \mathbf{\&}\overline{n} \\
\qquad \mathbf{\})} \\
\quad \mathbf{:}\; j \;\mathbf{in}\; \mathbf{\&}\overline{n} \\
\quad \mathbf{\}} \\
\mathbf{:}\; i \;\mathbf{in}\; \mathbf{\&}\overline{n} \\
\mathbf{\}}
\end{array}
$$

Desugared this becomes

$$
\begin{array}{l}
\mathbf{\{} \\
\quad \mathbf{\{} \\
\qquad \mathbf{sum(\{} \\
\qquad \mathbf{+(elt^{[]}_{int}(elt^{[]}_{[int]}(}A\mathbf{,}\; k\mathbf{),}\; i\mathbf{),}\; \mathbf{elt^{[]}_{int}(elt^{[]}_{[int]}(}B\mathbf{,}\; j\mathbf{),}\; k\mathbf{))} \\
\qquad \mathbf{:}\; k \;\mathbf{in\ iota}\; \overline{n} \;\mathbf{using}\; (A, B) \\
\qquad \mathbf{\})} \\
\quad \mathbf{:}\; j \;\mathbf{in\ iota}\; \overline{n} \;\mathbf{using}\; (A, B) \\
\quad \mathbf{\}} \\
\mathbf{:}\; i \;\mathbf{in\ iota}\; \overline{n} \;\mathbf{using}\; (A, B) \\
\mathbf{\}}
\end{array}
$$

The type system proves that the expression has the type $\mathbf{\{\{int\}\}}$ in the context

$$[A \mapsto \mathbf{[[int]]}, B \mapsto \mathbf{[[int]]}]$$

so the result is sequenced. We could tabulate it if we wanted to. We see that A and B cannot be sequences since they are part of the **using** term. This corresponds well to the intuition that it is not possible to stream the two matrices to form their product. It should however be possible to stream the rows one of the matrices, and indeed, if we reformulate the expression to map over the elements of one of the matrices, we obtain

$$
\begin{array}{l}
\mathbf{\{} \\
\quad \mathbf{\{} \\
\qquad \mathbf{sum(\{} \\
\qquad row[k] \,\mathbf{*}\; B[k][j] \\
\qquad \mathbf{:}\; k \;\mathbf{in}\; \mathbf{\&}\overline{n} \\
\qquad \mathbf{\})} \\
\quad \mathbf{:}\; j \;\mathbf{in}\; \mathbf{\&}\overline{n}\mathbf{;} \\
\quad \mathbf{\}} \\
\mathbf{:}\; row \;\mathbf{in}\; A' \\
\mathbf{\}}
\end{array}
$$

that will type check in the context

$$[A' \mapsto \mathbf{\{[int]\}}, B \mapsto \mathbf{[[int]]}]$$

yielding the same type as before. Evaluating the first expression in a context where A and B maps to two matrices of size $n \times n$ yields a $n \times n$ sequence for the value, and for the complexity $\omega = \langle W, D, M, N \rangle$ we have

$$\begin{aligned} W &= O(n^3) \\ D &= O(1) \\ M &= O(n) \\ N &= O(n^3) \end{aligned}$$

The derived complexities using P processors are then

$$T = O(n^3/P + \log P)$$

$$S = O(\min(nP, n^3))$$

This shows that given sufficiently few available processors P or sufficiently large matrices, we can compute matrix-matrix product in almost n space by streaming the rows of A. If we tabulated the result, we would get $O(n^2)$ space. Using squared space the obvious choice for a efficient low-level handwritten implementation, so this is fine, but it also shows that traditional NDP languages use cubic space regardless of P, which is definitely not fine for very large matrices.

3.7 Evaluation

We have presented a minimalistic NDP source language that is similar to traditional NDP languages, but with the introduced concept of sequences. We have given a cost model which in term of work and step complexities is similar to traditional NDP cost models, but with the addition of a novel space cost model for work space and step space, which is the space analogy of work and steps. We have tried the model on a couple of simple examples and demonstrated that some algorithms that performs bad in traditional NDP languages in terms of space complexity, performs good using sequences. We have also pointed out some flaws in the generality of sequences: Some well-typed sequence expression cannot be streamed and some sequence expression that will not type, can actually be streamed. We believe that a more complicated type system using size analysis and availability frames can help determine exactly what expressions can be streamed. The remainder of this thesis is devoted to showing that this ideal cost model is plausibly implementable.

4 Streaming language

The target language was originally intended to be combinator based where each primitive transition is a basic combinator, and larger combinators is build using composition $\circ$ and pair $\langle - , - \rangle$, i.e a Cartesian category. It is indeed possible to construct a combinator from a well-formed source language expression, but to provide a streaming interpretation of such a combinator turns out to be exceedingly difficult. The main problem is that some transitions does not consume (resp. produce) exactly the entire input (resp. output) buffer in one step. Keeping track of partial consumed (resp. produced) buffers while preserving sharing of sub-expressions is probably possible, but no satisfactory denotation, reduction or transition system using combinators was found.

Instead we present the proposed target language as a system of stream definitions. It is based on buffers as values and stream transformers as primitive operations. The stream definitions of a well-formed system forms a directed acyclic graph (DAG). A stream definition is of the form:

$$s \;\texttt{:=}\; t(s_1, \dots, s_k) \qquad \text{where } k \geq 0$$

and it can be read as "s is defined as the stream transformation t on the input streams $s_1, \dots, s_k$". s are unique stream identifiers, and streaming system is essentially a series of assignment forming a data-flow network.

4.1 Syntax

The syntax of a streaming system is given by

$$s \in \textbf{SId}$$

$$\begin{array}{rcll}
\textbf{SSys} \ni G & ::= & \texttt{let } \Phi \texttt{ in } st & \\
\textbf{PIdT} \ni st & ::= & s \mid () \mid (st, st) & \\
\textbf{Eqs} \ni \Phi & ::= & \phi; \dots; \phi & \\
\textbf{Eq} \ni \phi & ::= & s \;\texttt{:=}\; t(s_1, \dots, s_k) & k \geq 0 \\
\textbf{Trans} \ni t & ::= & \mid \oplus\texttt{\^{}} & \\
 & & \mid \texttt{flagscan}_{\otimes} & \\
 & & \mid \texttt{pack}_{\pi} & \\
 & & \mid \texttt{segfetch}_{\pi} & \\
 & & \mid \texttt{flush}_{\pi} & \\
 & & \mid \texttt{flagdist}_{\pi} & \\
 & & \mid \texttt{2flags} & \\
 & & \mid \texttt{merge}_{k,\pi} & \\
 & & \mid \texttt{interleave} & \\
 & & \mid \texttt{reconstruct} & \\
\textbf{VOp} \ni \oplus\texttt{\^{}} & ::= & \texttt{+\^{}} \mid \texttt{*\^{}} \mid \cdots &
\end{array}$$

A streaming system

$$\texttt{let } \Phi \texttt{ in } st$$

is a list of stream definitions $\Phi \in \mathbf{Eqs}$ and a product (or tree) of stream identifiers representing the output $st \in \mathbf{PIdT}$ streams of the system. The output is defined as a tree in order to relate it to a high level value.

Each definition

$$s \; := t(s_1, \ldots, s_k)$$

defines one primitive stream in the system, and the transformer t determines how the stream is populated as a transformation from the input streams $s_1, \ldots, s_k$. All primitive transformers have efficient implementations on homogeneous data parallel machines such as GPGPUs.

The target language is meant to be interpreted by a runtime system that can perform scheduling dynamically, and dispatch the transformer of a stream definitions multiple times to process consecutive blocks.

4.2 Type system

The primitive types of the target language consists of primitive stream types of bounded or unbounded size:

$$\mathbf{VPTyp} \ni \mu ::= \{\pi\}^{\mathrm{b}} \mid \{\pi\}^{\mathrm{u}}$$

where $\{\pi\}^{\mathrm{b}}$ is a stream with a bounded buffer and $\{\pi\}^{\mathrm{u}}$ is a stream with an unbounded buffer. The type of a output st is then a tree of μ's:

$$\mathbf{VTyp} \ni \nu ::= \mathtt{1} \mid \mu \mid \nu * \nu$$

The type of a streaming system

$$\mathtt{let}\ \Phi\ \mathtt{in}\ st$$

is decomposed into the type of the definitions Φ which is a typing context

$$\Pi : \mathbf{SId} \rightharpoonup \mathbf{VPTyp}$$

assigning stream identifiers to primitive target types by the type system

$$\vdash \Phi : \Pi$$

and the type of the output streams st which is given a type $\nu \in \mathbf{VTyp}$ in a typing context Π by the type system

$$\Pi \vdash st : \nu$$

Every primitive transition is assigned a type of the form $\mu_1 * \cdots * \mu_k \rightarrow \mu$:

$$
\begin{array}{ll}
\texttt{2flags} & : \{\mathbf{int}\}^{\mathrm{b}} \rightarrow \{\mathbf{bool}\}^{\mathrm{b}} \\
\forall \otimes : \pi \,.\, \texttt{flagscan}_{\otimes} & : \{\pi\}^{\mathrm{b}} * \{\mathbf{bool}\}^{\mathrm{b}} \rightarrow \{\pi\}^{\mathrm{b}} \\
\forall \pi \,.\, \texttt{pack}_{\pi} & : \{\pi\}^{\mathrm{b}} * \{\mathbf{bool}\}^{\mathrm{b}} \rightarrow \{\pi\}^{\mathrm{b}} \\
\forall \pi \,.\, \texttt{segfetch}_{\pi} & : \{\pi\}^{\mathrm{u}} * \{\mathbf{int}\}^{\mathrm{b}} * \{\mathbf{int}\}^{\mathrm{b}} \rightarrow \{\pi\}^{\mathrm{b}} \\
\forall \pi \,.\, \texttt{flush}_{\pi} & : \{\pi\}^{\mathrm{b}} \rightarrow \{\pi\}^{\mathrm{u}} \\
\forall \pi \,.\, \texttt{flagdist}_{\pi} & : \{\pi\}^{\mathrm{b}} * \{\mathbf{bool}\}^{\mathrm{b}} \rightarrow \{\pi\}^{\mathrm{b}} \\
\forall \pi \,.\, \texttt{merge}_{k,\pi} & : \{\mathbf{int}\}^{\mathrm{b}} * (\{\pi\}^{\mathrm{b}})^{k} \rightarrow \{\pi\}^{\mathrm{b}} \\
\texttt{interleave} & : \{\mathbf{bool}\}^{\mathrm{b}} * \{\mathbf{bool}\}^{\mathrm{b}} \rightarrow \{\mathbf{bool}\}^{\mathrm{b}} \\
\texttt{reconstruct} & : \{\mathbf{bool}\}^{\mathrm{b}} * \{\mathbf{bool}\}^{\mathrm{b}} \rightarrow \{\mathbf{bool}\}^{\mathrm{b}} \\
\forall \oplus : \pi_1 * \cdots * \pi_k \rightarrow \pi \,.\, \oplus\text{^} & : \{\pi_1\}^{\mathrm{b}} * \cdots * \{\pi_k\}^{\mathrm{b}} \rightarrow \{\pi\}^{\mathrm{b}}
\end{array}
$$

The stream transformer are described informally in table 3. The type of an streaming system derived by typing one stream definition at a time, starting from the top, and building up the typing context Π incrementally. This ensures that no definitions use stream identifiers below them, thereby ensuring the system is indeed a DAG.

$\boxed{\vdash \Phi : \Pi}$

$$
\frac{}{\vdash \epsilon : []} \qquad \text{(T-Sys-0)}
$$

$$
\frac{\vdash \Phi : \Pi}{\vdash (s \ := t(s_1, \ldots, s_k)); \Phi : \Pi[s \mapsto \mu]} \; t : \Pi(s_0) * \cdots * \Pi(s_{k-1}) \rightarrow \mu \qquad \text{(T-Sys-1)}
$$

The type of the output streams st is then trivially given by looking up the type of each s in Π

Transition	Description	Example
$\texttt{flagscan}_{\otimes}$	Performs an exclusive prefix segmented scan on a scalar stream, given a stream of segment flags. The dummy values in the output stream will hold the corresponding reduced results.	$\texttt{flagscan}_{\mathbf{sum}}$ [3, 1, 1, _, 2, _] [**f**, **f**, **f**, **t**, **f**, **t**] = [0, 3, 4, 5, 0, 2]
$\texttt{pack}_{\pi}$	Packs a stream given a stream of flags.	$\texttt{pack}_{\mathbf{int}}$ [3, 8, 7, 3, 2, 1] [**f**, **f**, **f**, **t**, **f**, **t**] = [3, 1]
$\texttt{segftech}_{\pi}$	Takes an unbounded stream of values, and performs a segmented fetch. The segments are defined by two integer streams, the first defining segment starts, and the second defining segment lengths.	$\texttt{segfetch}_{\mathbf{int}}$ [10, 20, 30] [0, 0, 2, 2] [1, 3, 1, 1] = [10, 10, 20, 30, 30, 30]
$\texttt{flush}_{\pi}$	Flushes a bounded stream by writing the content to an unbounded stream. Informally this is essentially an identity function.	$\texttt{flush}_{\mathbf{int}}$ [1, 2, 3] = [1, 2, 3]
$\texttt{flagdist}_{\pi}$	Replicates each element of a stream over a given flag stream. Each element in the first stream is replicated for each consecutive zero in the flag stream.	$\texttt{flagdist}_{\mathbf{int}}$ [3, 8] [**f**, **f**, **f**, **t**, **f**, **t**] = [3, 3, 3, 3, 8, 8]
`2flags`	Converts an integer stream to an equivalent unary representation encoded as a boolean stream. This is the only operation that "creates" more parallelism. I.e. the output stream is longer then the input stream.	`2flags` [3, 1] = [**f**, **f**, **f**, **t**, **f**, **t**]
$\texttt{merge}_{k,\pi}$	Merges a number of streams of the same type by interleaving the elements. The interleaving is guided by a selector stream of indices.	$\texttt{merge}_{3,\mathbf{char}}$ [0, 1, 2, 1, 2, 0, 2] [a, b] [i, j] [x, y, z] = [a, i, x, j, y, b, z]
`interleave`	Merges two boolean streams by interleaving segments of the form **f**, ..., **f**, **t**.	`interleave` [**f**, **f**, **f**, **t**, **f**, **t**] [**f**, **f**, **t**, **t**] = [**f**, **f**, **f**, **t**, **f**, **f**, **t**, **f**, **t**, **t**]
`reconstruct`	Will be explained later. In short, it "unpacks" a boolean stream to match the length of another, in such a way that the **t**'s are in the same positions.	`reconstruct` [**f**, **f**, **t**] [**f**, **f**, **f**, **t**, **f**, **t**] = [**f**, **f**, **f**, **f**, **f**, **t**]

Table 3: The primitive stream transformers of the target language.

$\boxed{\Pi \vdash st : \nu}$

$$\frac{}{\Pi \vdash () : \mathtt{1}} \qquad \text{(T-ST-0)}$$

$$\frac{}{\Pi \vdash s : \mu}\ \Pi(s) = \mu \qquad \text{(T-ST-1)}$$

$$\frac{\Pi \vdash st_0 : \nu_0 \qquad \Pi \vdash st_1 : \nu_1}{\Pi \vdash (st_0, st_1) : \nu_0 * \nu_1} \qquad \text{(T-ST-2)}$$

4.3 Interpretation

The interpretation of a streaming system is a system

$$\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow_s \langle \Sigma', A', \chi' \rangle \ ! \ \varpi$$

which can be read as "Given the streaming definitions Φ and with a block size B, the buffer store Σ, accumulator store A and cursor map χ steps, by firing definition s, to the new buffer store Σ', new accumulator store A' and new cursor map χ' at the complexity costs ϖ".

Just like in the big-step semantics of the source language, runtime errors manifest themselves as undefined. I.e. there will be no derivation for target expression that results in runtime error.

We will now explain each component of this transition system in detail.

4.3.1 Buffers

A buffer is a *window* on a stream.

$$\textbf{Buffer} \ni buf ::= {}_{\ell}[a_0, \ldots, a_{k-1}]_{eos}$$

where $\ell \in \mathbf{N}$ is a cursor that indicates how far the window of the buffer is, and

$$\textbf{EOS} \ni eos ::= \checkmark \mid *$$

is and end-of-stream indicator, where $\checkmark$ marks that the window of the stream is at the end, and $*$ marks that there is more to come.

A buffer store is then a finite map from stream identifiers to buffers

$$\Sigma ::= [s_0 \mapsto buf_0, \ldots, s_{k-1} \mapsto buf_{k-1}]$$

We first define some simple operations on buffers

Append The append operation appends the values of two buffers.

$$(+\!\!+) : \textbf{Buffer} \times \textbf{Buffer} \rightharpoonup \textbf{Buffer}$$

$${}_{\ell}[a_0, \ldots, a_{k-1}]_{*} +\!\!+ {}_{0}[a'_0, \ldots, a'_{k'-1}]_{eos} = {}_{\ell}[a_0, \ldots, a_{k-1}, a'_0, \ldots, a'_{k'-1}]_{eos}$$

Read The read operation reads a buffer from a given cursor to the end of the buffer. Reading outside the window of a buffer is undefined.

$$[\cdot] : \textbf{Buffer} \times \textbf{Cursor} \rightharpoonup \textbf{Buffer}$$

$${}_{\ell}[a_0, \ldots, a_{k-1}]_{eos}[\ell_0] = {}_{\ell_0}[a_{\ell_0-\ell}, \ldots, a_{k-1}]_{eos} \qquad \ell \leq \ell_0 < \ell + k$$

Window size The window size of a buffer is defined as

$$|{}_{\ell}[a_0, \ldots, a_{k-1}]_{eos}| = k$$

this generalizes to buffer stores by summation

$$[s_0 \mapsto buf_0, \ldots, s_{k-1} \mapsto buf_{k-1}] = \sum_{i=0}^{k-1} |buf_i|$$

End of stream check We define a predicate on buffers which is true if the stream has ended

$$done({}_{\ell}[a_0, \ldots, a_{k-1}]_{eos}) \text{ if and only if } eos = \checkmark$$

Getting the data

$$data : \textbf{Buffer} \to \textbf{PVal}^*$$

$$data({}_{\ell}[a_0, \ldots, a_{k-1}]_{eos}) = a_0, \ldots, a_{k-1}$$

Initial buffer store All buffers are initially empty, starts at position zero, and have not ended yet, except for an initial control buffer that starts as ${}_0[\mathbf{f}, \mathbf{t}]_{\checkmark}$. The control buffer is used to control the parallel structure during execution, and it is an entry point for literals, as it is used to distribute them. Each $\mathbf{f}$ in the control stream indicates one sub-computation, and each $\mathbf{t}$ delimits the sub-computations in groups. The initial value of $[\mathbf{f}, \mathbf{t}]$ can be interpreted as a single group of parallel degree of 1. The exact usage of the control stream become more clear in the transformation section.

Given a type context Π, we can construct an initial buffer store:

$$\Sigma_{init} : (\textbf{SId} \rightharpoonup \textbf{VPTyp}) \to (\textbf{SId} \rightharpoonup \textbf{Buffer})$$

$$\begin{aligned} &\Sigma_{init}([s_0 \mapsto \mu_0, \ldots, s_{k-1} \mapsto \mu_{k-1}]) = \\ &\quad [s_{ctrl} \mapsto {}_0[\mathbf{f}, \mathbf{t}]_{\checkmark}, s_0 \mapsto {}_0[]_{*}, \ldots, s_{k-1} \mapsto {}_0[]_{*}] \end{aligned}$$

4.3.2 Accumulator store

Each stream definition has a stream transformer t. Some transformers have an accumulator associated with it:

$$acc \in \textbf{Acc} = \textbf{PVal}$$

The accumulators are stored in a finite map from stream identifiers to accumulated values:

$$A : \textbf{SId} \rightharpoonup \textbf{PVal}$$

$$A ::= [s_0 \mapsto acc_0, \ldots, s_{k-1} \mapsto acc_{k-1}]$$

Not all transformers have an accumulator, so the domain of the accumulator map is often smaller than the domain of a corresponding type context. The value of the initial accumulator depends on the transformer. We can construct an initial accumulator map given a list of definitions Φ like this:

$$A_{init} : \mathbf{SSys} \to (\mathbf{SId} \rightharpoonup \mathbf{PVal})$$

$$A_{init}(\epsilon) = []$$

$$A_{init}(s := t(s_1, \ldots, s_k); \Phi) = \begin{cases} A_{init}(\Phi)[s \mapsto 0] & t = \texttt{2flags} \\ A_{init}(\Phi)[s \mapsto a_0] & t = \texttt{flagscan}_{\otimes} \quad a_0 \text{ is the identity element of } \otimes \\ A_{init}(\Phi)[s \mapsto 0] & t = \texttt{segfetch}_{\pi} \\ A_{init}(\Phi)[s \mapsto \mathbf{f}] & t = \texttt{interleave} \\ A_{init}(\Phi)[s \mapsto \mathbf{f}] & t = \texttt{reconstruct} \\ A_{init}(\Phi) & \text{otherwise} \end{cases}$$

4.3.3 Cursor map

The cursor map implements precise reference counting on stream elements. Each stream has a number of streams reading it, and each reader must have a separate cursor to the stream that is read. The cursor map forms a sparse matrix, that is triangular due to the DAG structure of definitions. We define it as a finite map from pairs of stream identifiers to cursors:

$$\chi : \mathbf{SId} \times \mathbf{SId} \rightharpoonup \mathbf{Cursor}$$

$$\mathbf{CursorMap} \ni \chi ::= [(s_0, s'_0) \mapsto \ell_0, \ldots, (s_{k-1}, s'_{k-1}) \mapsto \ell_{k-1}]$$

$\chi(s)$ defines the cursors on s, and $\chi(s_0)(s_1) = \ell$ means that s_1 is defined as a transformation of s_0 and possibly other streams, and in order to fire the definition, we must read the values in s_0 at position ℓ.

In order to keep values in the output buffers of a system, we add a phantom reader, that will actually never read the stream. The unique stream identifier s_{ctrl} serves this purpose.

All cursors are initially zero. We can construct an initial cursor map given a list of definitions and a stream identifier tree:

$$\chi_{init} : \mathbf{SSys} \times \mathbf{PIdT} \to \mathbf{CursorMap}$$

$$\chi_{init}(\epsilon, st) = []$$

$$\chi_{init}(s := t(s_1, \ldots, s_k); \Phi, st) = \chi_{init}(\Phi) \cup initCursors(s, \Phi, st)$$

$$initCursors : \mathbf{SId} \times \mathbf{SSys} \times \mathbf{PIdT} \to \mathbf{StreamCursors}$$

$$initCursors(s, \epsilon, st) = \begin{cases} [(s, s_{ctrl}) \mapsto \infty] & s \notin st \\ [(s, s_{ctrl}) \mapsto 0] & s \in st \end{cases}$$

$$initCursors(s, (s' := t(s_1, \ldots, s_k)); \Phi, st) = \begin{cases} initCursors(\Phi, st)[(s, s') \mapsto 0] & \exists i \in 1..k \,.\, s_i = s \\ initCursors(\Phi, st) & \text{otherwise} \end{cases}$$

The quadratic complexity is acceptable since the construction of the initial cursor map can happen statically.

4.3.4 Target complexity

The target language complexities are similar to the complexity quadruple $\omega \in \mathbf{Cx}$ from the source language, but there is no notion of step space in the target language. Target language complexities are therefore merely triples $\mathbf{TCx} \ni \varpi = \langle W, D, M \rangle$. In addition, there is no high-level notion of parallel sub-computations in the low level language, so there is no parallel composition of complexities either, and we only define a sequential composition monoid for target language complexities

$$\boxdot^{\mathrm{t}} : \mathbf{TCx} \times \mathbf{TCx} \to \mathbf{TCx}$$

$$\boxdot^{\mathrm{t}} = + \times + \times \max$$

with identity element

$$\langle 0, 0, 0 \rangle$$

4.3.5 Transformer semantics

The streaming semantics of primitive transformers are defined as a function parametrized by B with the following type signature:

$$[\![\cdot]\!]_B : \mathbf{Trans} \to \mathbf{TState} \times \mathbf{Buffer}^k \rightharpoonup (\mathbf{TState} \times \mathbf{Buffer} \times \mathbf{Cursor}^k)$$

It takes a transformer t and defines the meaning of it $[\![t]\!]_B$ (using block size B) as a function from a an accumulator acc and k input buffer buf_i to a new accumulator acc', an output buf update and k cursor updates ℓ_i:

$$[\![t]\!]_B \; acc \; (buf_1, \dots, buf_k) = (acc', buf, (\ell_1, \dots, \ell_k))$$

The input buffers can in principle contain any number of elements, but the output buffer update must invariably contain *at most* B elements. This restriction is a necessity in order to unsure that there is room in the output buffer for the update.

For all transformers, we can always infer the end-of-stream indicator from the input buffers, and the resulting cursor updates - the output buffer update is marked as ended $\checkmark$ if and only if all the input buffers has ended $\checkmark$, and all elements were used in the transformation. Since the resulting buffer is an update buffer, we set the cursor to 0. We can therefore make do with a simpler semantic function:

$$[\![\cdot]\!]'_B : \mathbf{Trans} \to \mathbf{TState} \times (\mathbf{PVal}^*)^k \rightharpoonup (\mathbf{TState} \times \mathbf{PVal}^* \times \mathbf{Cursor}^k)$$

that returns a takes and output lists of primitive values $\mathbf{PVal}^*$ instead of a buffers **Buffer**, and we can then decorate the output list with a window:

$$\begin{array}{l} [\![t]\!]_B \; acc \; (buf_1, \dots, buf_k) = \\ \quad \text{let} \;\; ([\overrightarrow{a}^l], acc', (\ell_1, \dots, \ell_k)) = [\![t]\!]'_B \; acc \; (data(buf_1), \dots, data(buf_k)) \\ \quad \text{in} \;\; ({}_0[\overrightarrow{a}^l]_{eos}, acc', (\ell_1, \dots, \ell_k)) \\ \qquad \text{where } eos = \begin{cases} \checkmark & |buf_i| = \ell_i \wedge done(buf_i) \text{ for i in } 1..k \\ * & \text{otherwise} \end{cases} \end{array}$$

Defining the real semantic function $[\![\cdot]\!]_B$ on buffers instead of lists of primitive values is a technicality that allows us to use cleaner notation in the transition system.

We will now define the formal semantics of each of the transformers, and argue that each of them has an efficient parallel implementation, that is always logarithmic in the buffer size.

Vectorized scalar operations All scalar operations have semantics on the following form.

$$[\![\oplus\hat{}\,]\!]'_B \; () \; ([n_0, \ldots, n_{l_0-1}], [m_0, \ldots, m_{l_1-1}]) = ((), [n_0 \oplus m_0, \ldots, n_{l-1} \oplus n_{l-1}], (l, l))$$

$$\text{where } l = \min(B, l_0, l_1)$$

They obviously all have efficient parallel implementation, also in the context of streaming.

Flag scan Flag scan is known to have an efficient parallel implementation (see [20], or [19] page 45). The question is if it also has an efficient implementation in a streaming model. The only difference between flag scan in this model and a regular flag scan, is that we have an accumulator that must be the first element of the result and we must carry a new accumulator on to the next dispatch. We can add the accumulator to the first element (using the monoid associative operator). If its the initial accumulator (the identity element) it does nothing, but if it is an accumulated value, the value will propagate through the scan as we would expect. As for the new accumulator, we can simply read it of the last element in scan. Both operations are constant time, so they don not affect the overall performance of the algorithm.

We can describe the semantic formally as a recursive definition. We first assume a cons notation for lists of values $(a : as)$ and $[]$.

$$[\![\texttt{flagscan}_{\textbf{sum}}]\!] \; n_{acc} \; (ns, bs) = (n'_{acc}, ns', (\ell', \ell'))$$

$$\text{where } (n'_{acc}, ns', \ell') = \mathit{flagscan}_{\textbf{sum}} \; n_{acc} \; ns \; bs \; B$$

$$\mathit{flagscan}_{\textbf{sum}} \; n_{acc} \; ns \; bs \; k = (n_{acc}, [], 0) \quad (ns = [] \text{ or } bs = [] \text{ or } k = 0)$$

$$\begin{aligned}&\mathit{flagscan}_{\textbf{sum}} \; n_{acc} \; (n : ns) \; (\mathbf{f} : bs) \; k = (n'_{acc}, n_{acc} : ns', \ell' + 1)\\ &\quad \text{where } (n'_{acc}, ns', \ell') = \mathit{flagscan}_{\textbf{sum}} \; (n + n_{acc}) \; ns \; bs \; (k - 1)\end{aligned}$$

$$\begin{aligned}&\mathit{flagscan}_{\textbf{sum}} \; n_{acc} \; (n : ns) \; (\mathbf{t} : bs) \; k = (n'_{acc}, n_{acc} : ns', \ell' + 1)\\ &\quad \text{where } (n'_{acc}, ns', \ell') = \mathit{flagscan}_{\textbf{sum}} \; 0 \; ns \; bs \; (k - 1)\end{aligned}$$

Pack Pack has an efficient implementation using scans and permutation. The operation has an efficient implementation is described in [19] page 44, where the elements that a filtered away are permuted to the back of the buffer. By counting the **t**'s in the flag stream (by a reduction), we can find the exact size of the output buffer, and we can simply clamp the buffer to that size. Pack therefore has an efficient implementation in streaming context. The formal semantics are:

$$[\![\texttt{pack}_\pi]\!] \; () \; (as, bs) = ((), as', (\ell', \ell'))$$

$$\text{where } (as', \ell') = \mathit{pack}_\pi \; as \; bs \; B$$

$$pack_\pi \; () \; as \; bs \; k = ([\,], 0) \quad (as = [\,] \text{ or } bs = [\,] \text{ or } k = 0)$$

$$\begin{aligned} &pack_\pi \; () \; (a : as) \; (\mathbf{f} : bs) \; k = (as', \ell' + 1) \\ &\quad \text{where } (as', \ell') = pack_\pi \; () \; as \; bs \; (k-1) \end{aligned}$$

$$\begin{aligned} &pack_\pi \; () \; (a : as) \; (\mathbf{t} : bs) \; k = (a : as', \ell' + 1) \\ &\quad \text{where } (as', \ell') = pack_\pi \; () \; as \; bs \; (k-1) \end{aligned}$$

To flags An example of `2flags`:

$$\begin{aligned} &\texttt{2flags} \\ &[3, 1, 2] \\ = \; &[0, 0, 0, 1, 0, 1, 0, 0, 1] \end{aligned}$$

`2flags` can be implemented by initializing the output buffer to zeroes and writing the 1's at the correct positions. We can get the positions by adding one to all elements of the input buffer and performing and inclusive +-scan. In the context of streaming, we must use an integer accumulator to carry over the number of zeroes that was generated at the end of the output buffer.

$$[\![\texttt{2flags}]\!] \; n_{acc} \; ns = \mathit{2flags} \; as \; bs \; B$$

$$\mathit{2flags} \; n_{acc} \; ns \; k = (n_{acc}, [\,], 0) \quad (ns = [\,] \text{ or } k = 0)$$

$$\mathit{2flags} \; n_{acc} \; (n : ns) \; k = (n'_{acc}, \mathbf{t} : bs', \ell' + 1) \quad (n_{acc} = n)$$

$$\text{where } (n'_{acc}, bs', \ell') = \mathit{2flags} \; 0 \; ns \; (k-1)$$

$$\mathit{2flags} \; n_{acc} \; (n : ns) \; k = (n'_{acc}, \mathbf{f} : bs', \ell') \quad (n_{acc} \neq n)$$

$$\text{where } (n'_{acc}, bs', \ell') = \mathit{2flags} \; (n_{acc} + 1) \; (n : ns) \; (k-1)$$

Flag distribute Example of flag distribute

$$\begin{aligned} &\texttt{flagdist}_{\textbf{int}} \\ &[3, 8] \\ &[0, 0, 0, 1, 0, 1] \\ = \; &[3, 3, 3, 3, 8, 8] \end{aligned}$$

Flag requires no accumulator to be implemented. We can figure out how much was read from the first input buffer by a +-reduction of the second. We conclude that flag distribute has en efficient implementation in the context of streaming, and the formal semantics are:

$$[\![\texttt{flagdist}_\pi]\!] \; () \; (as, bs) = ((), as', (\ell'_0, \ell'_1))$$

$$\text{where } (as', \ell'_0, \ell'_1) = \mathit{flagdist}_\pi \; as \; bs \; B$$

$$\mathit{flagdist}_\pi\ ()\ as\ bs\ k = ([], 0, 0) \quad (as = []\text{ or }bs = []\text{ or }k = 0)$$

$$\mathit{flagdist}_\pi\ ()\ (a : as)\ (\mathbf{f} : bs)\ k = (a : as', \ell'_0, \ell'_1 + 1)$$
$$\text{where } (as', \ell'_0, \ell'_1) = \mathit{flagdist}_\pi\ ()\ (a : as)\ bs\ (k - 1)$$

$$\mathit{flagdist}_\pi\ ()\ (a : as)\ (\mathbf{t} : bs)\ k = (a : as', \ell'_0 + 1, \ell'_1 + 1)$$
$$\text{where } (as', \ell'_0, \ell'_1) = \mathit{flagdist}_\pi\ ()\ as\ bs\ (k - 1)$$

Segmented fetch

$$\begin{aligned} & \texttt{segfetch}_{\mathbf{int}} \\ & [10, 20, 30] \\ & [0, 0, 2, 2] \\ & [1, 3, 1, 1] \\ = \ & [10, 10, 20, 30, 30, 30] \end{aligned}$$

The segmented fetch is actually not a very primitive operation. It can be implemented by `2flags`, `flagdist`, scans, and a gather operation, but for the purpose of moving cursors in unbounded buffers we provide this compound operation as a primitive. During the streaming of a segmented fetch, we require as an invariant that segment start indices arrive in monotonic increasing order. This allows us to move the cursor to the last known segment start.

Segmented fetch must use an integer accumulator in order to handle segments overlapping window boundaries.

For the purpose of getting an element from an arbitrary position in an unbounded buffer we define the operations $as[i]$ as

$$[a_0, \ldots, a_{l-1}][i] = a_i \qquad 0 \le i < l$$

We can the define segfetch as

$$[\![\texttt{segfetch}_\pi]\!]\ n_{acc}\ (as, ns_0, ns_1) = (n'_{acc}, ns', (\ell'', \ell', \ell'))$$

$$\text{where } (n'_{acc}, as', \ell') = \mathit{segfetch}_\pi\ n_{acc}\ as\ ns_0\ ns_1\ B$$

$$\mathit{segfetch}_\pi\ n_{acc}\ as\ ns_0\ ns_1\ k = (n_{acc}, [], 0) \quad (as = []\text{ or }ns_0 = []\text{ or }ns_1 = []\text{ or }k = 0)$$

$$\mathit{segfetch}_\pi\ n_{acc}\ as\ (n_0 : ns_0)\ (n_1 : ns_1)\ k = (n'_{acc}, as', \ell' + 1) \quad (n_{acc} = n_1)$$
$$\text{where } (n'_{acc}, as', \ell') = \mathit{segfetch}_\pi\ 0\ as\ ns_0\ ns_1\ k$$

$$\mathit{segfetch}_\pi\ n_{acc}\ as\ (n_0 : ns_0)\ (n_1 : ns_1)\ k = (n'_{acc}, as[n_0 + n_{acc}] : as', \ell' + 1) \quad (n_{acc} < n_1)$$
$$\text{where } (n'_{acc}, as', \ell') = \mathit{segfetch}_\pi\ (n_{acc} + 1)\ as\ (n_0 : ns_0)\ (n_1 : ns_1)\ (k - 1)$$

Flush In terms of transformer semantics, flush is (almost) simply the identity function:

$$[\![\texttt{flush}]\!]_B\ ()\ [a_0, \ldots, a_{l-1}] = ((), [a_0, \ldots, a_{l'-1}], \ell')$$
$$\text{where } \ell' = \min(B, \ell)$$

Merge Example:

$$\begin{aligned} & \texttt{merge}_{3,\textbf{char}} \\ & \quad [0,1,2,1,2,0,2] \\ & \quad [\texttt{a},\texttt{b}] \\ & \quad [\texttt{i},\texttt{j}] \\ & \quad [\texttt{x},\texttt{y},\texttt{z}] \\ = & \quad [\texttt{a},\texttt{i},\texttt{x},\texttt{j},\texttt{y},\texttt{b},\texttt{z}] \end{aligned}$$

Merge has the following semantics:

$$\begin{aligned} & [\![\texttt{merge}_{k,\pi}]\!] \ () \ (ns, as_1, \dots, as_k) = ((), a', (\ell', \ell'_1, \dots, \ell'_k)) \\ & \quad \text{where } (a', \ell', (\ell'_1, \dots, \ell'_k)) = merge_{k,\pi} \ ns \ (as_1, \dots, as_k) \ B \end{aligned}$$

$$merge_{k,\pi} \ ns \ (as_1, \dots, as_k) = ([], 0, (0, \dots, 0)) \quad (k = 0 \text{ or } ns = [] \text{ or } \exists i \ . \ as_i = [])$$

$$\begin{aligned} & merge_{k,\pi} \ (n : ns) \ (as_1, \dots, as_k) = (a_n : as', \ell' + 1, (\ell'_1, \dots, \ell'_n + 1, \dots, \ell'_k)) \\ & \quad \text{where } (as', \ell', (\ell'_1, \dots, \ell'_k)) = merge_{k,\pi} \ (ns) \ (as_1, \dots, as'_n \dots, as_k) \\ & \qquad as_n = (a_n : as'_n) \end{aligned}$$

4.3.6 Transition system

The semantics of the target language is defined in a single transition rule:

$$\boxed{\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow_s \langle \Sigma', A', \chi' \rangle \ ! \ \varpi}$$

$$\frac{}{\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow_s \langle \Sigma', A', \chi' \rangle \ ! \ \langle |buf|, 1, |\Sigma'| \rangle}(*) \qquad ()$$

$$\begin{array}{lllll} (*) & 1. & \Phi(s) & = & t \ (s_1, \dots, s_k) \\ & 2. & acc & = & A(s) \\ & 3. & buf_1 & = & \Sigma(s_1)[\chi(s_1, s)] \\ & & \dots & & \\ & & buf_k & = & \Sigma(s_k)[\chi(s_k, s)] \\ & 4. & \multicolumn{3}{l}{(acc', buf, (\ell_1, \dots, \ell_k)) = [\![t]\!]_B \ acc \ (buf_1, \dots, buf_k)} \\ & 5. & A' & = & A[s \mapsto acc'] \\ & 6. & \chi' & = & \chi[(s_1, s) \mapsto \chi(s_1, s) + \ell_1, \dots, (s_k, s) \mapsto \chi(s_k, s) + \ell_k] \\ & 7. & \Sigma'' & = & \Sigma'[\ (s_i \mapsto \Sigma(s_i)[\min_x(\chi'(s_i, x))])\ _{i=1}^{k}] \\ & 8. & \Sigma' & = & \Sigma''[s \mapsto \Sigma(s) +\!\!+ \ buf] \end{array}$$

Explanation:

1. Load the stream definition $s := t(s_1, \dots, s_k)$.

2. Load the transformer accumulator for s.

3. Read the input buffers at the cursors of s.
4. Fire the transformer t on the accumulator and input buffers, yielding a new accumulator, an output buffer update and k cursor updates for the input buffers.
5. Update new accumulator.
6. Update new cursors of s on s_1 to s_k.
7. Trim input buffer windows to the new cursors. I.e. move the window of s_i to the smallest cursor on s_i.
8. Write output buffer update to output buffer.

A string of stream identifiers defines a schedule

$$ss ::= \epsilon \mid s; ss$$

We can define a transitive version of the transition rule given a schedule, which corresponds to evaluation a target language program:

$$\boxed{\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow_s s^* \langle \Sigma', A', \chi' \rangle \; ! \; \varpi}$$

$$\frac{\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow_s \langle \Sigma'', A'', \chi'' \rangle \; ! \; \varpi'' \qquad \Phi \vdash_B \langle \Sigma'', A'', \chi'' \rangle \rightarrow^*_{ss} \langle \Sigma', A', \chi' \rangle \; ! \; \varpi'}{\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow^*_{s;ss} \langle \Sigma', A', \chi' \rangle \; ! \; \varpi'' \boxdot^{\mathrm{t}} \varpi'} \quad ()$$

$$\frac{}{\Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow^*_{\epsilon} \langle \Sigma, A, \chi \rangle \; ! \; \langle 0, 0, 0 \rangle} \quad ()$$

In practice, the schedule ss is determined dynamically at runtime.

4.4 Derived complexities

We will now take a closer look at the cost model. Each transition has the cost

$$\langle |buf|, 1, |\Sigma'| \rangle$$

where buf is the output update buffer produced by the transformation of the transition and Σ' is the resulting buffer store.

Space in the target language is explicit, we always measure the size of the buffer store at each step. The accumulator store and cursor map only use $O(1)$ space if we assume the size of expressions are constant. We can therefore derive the actual space requirements directly from M:

$$S = O(M)$$

The derived time complexity is the same as in the source language. We record work and steps and assume that transformers use one step, even though most of them are actually logarithmic, but we will then account for this in the derived time just as in the source language:

$$T = O(W/P + D \log P)$$

If we choose the block size parameter B carefully, we can do better. If we use the invariant that output update buffers are always at most B in size, we can show by induction on $\rightarrow^*_{ss}$ that:

$$\begin{gathered} \forall \Phi \forall \Sigma \forall \chi \,. \\ \text{if } \Phi \vdash_B \langle \Sigma, A, \chi \rangle \rightarrow^*_{s;ss} \langle \Sigma', A', \chi' \rangle \; ! \; \langle W, D, M \rangle \\ \text{then } W \leq B \cdot D \end{gathered}$$

In other words $W = O(BD)$. If we choose a block size $B = O(P)$, we have $W = O(PD)$. We can then improve the derived time complexity as:

$$\begin{aligned} T &= O(W/P + D \log P) \\ &= O(PD/P + D \log P) \\ &= O(D + D \log P) \\ &= O(D \log P) \end{aligned}$$

In other words, we can completely remove the work term from the derived complexity, since the total work is be "remembered" in the steps.

4.4.1 Representation

Given a source language type σ, we can define the high-level meaning of a product of buffers as a representation of some source language value of type σ. More formally we define a product on buffers as

$$\mathbf{Buffer} \subset \mathbf{Places} \ni pt ::= () \mid (ps\,,\; ps) \mid p$$

and a *representation-of* function

$$\left\langle \cdot \right\rangle_\sigma : \mathbf{Places} \rightarrow \mathbf{Val}^*$$

We assume the existence of the higher order functions $zip : \mathbf{Val}^* \times \mathbf{Val}^* \rightarrow \mathbf{Val}^*$, $map : (\mathbf{Val} \rightarrow \mathbf{Val}) \rightarrow \mathbf{Val}^* \rightarrow \mathbf{Val}^*$ and $scan_\otimes : \mathbf{Val}^* \rightarrow \mathbf{Val}^*$ defined in the obvius way.

The representation-of function is then defined by induction on the type subscript as

$$\left\langle () \right\rangle_{\mathbf{1}} = \mathbf{()}$$

$$\left\langle [\,\overrightarrow{a}\,] \right\rangle_\pi = [\,\overrightarrow{a}\,] \qquad a_i : \pi$$

$$\left\langle (ps_0\,,\; ps_1) \right\rangle_{\sigma_0 * \sigma_1} = zip\; [\left\langle ps_0 \right\rangle_{\sigma_0}]\; [\left\langle ps_1 \right\rangle_{\sigma_1}]$$

$$\begin{aligned} \left\langle (([\,\overrightarrow{n}^{l}]\,,\; [\overrightarrow{m}^{l}])\,,\; ps_0) \right\rangle_{[\tau]} &= map_f\; (zip\; [\,\overrightarrow{n}^{\,l}]\; [\overrightarrow{m}^{l})] \\ &\quad \text{where } f\; \mathbf{(}n,\, m\mathbf{)} = \mathbf{[}\left\langle ps_0 \right\rangle_\tau [n..n + m - 1]\mathbf{]} \end{aligned}$$

$$\begin{aligned} \left\langle ([\, \left(\overrightarrow{\mathbf{f}}^{\,l'_i}, \mathbf{t} \right)_{i=0}^{l}]\,,\; ps_0) \right\rangle_{\{\sigma_0\}} &= map_f\; (zip\; (scan_+\; [\,\overrightarrow{l'}^{\,l}])\; [\,\overrightarrow{l'}^{\,l}]) \\ &\quad \text{where } f\; \mathbf{(}n,\, m\mathbf{)} = \mathbf{\{}\left\langle ps_0 \right\rangle_\tau [n..n + m - q]\mathbf{\}} \end{aligned}$$

4.5 Examples

If we return to the iota-square-sum example

$$\sum_{i=0}^{l-1} i^2$$

we can implement it in the target language as the system

$$\texttt{let}\ \Phi\ \texttt{in}\ s$$

where

$$\begin{aligned}
\Phi = \quad & s_0 := \textbf{const}_l\texttt{\^{}}(s_{ctrl}) \\
& s_1 := \texttt{2flags}(s_0) \\
& s_2 := \textbf{const}_1\texttt{\^{}}(s_{ctrl}) \\
& s_3 := \textbf{const}_2\texttt{\^{}}(s_{ctrl}) \\
& s_4 := \texttt{flagdist}_{\textbf{int}}(s_2, s_1) \\
& s_5 := \texttt{flagscan}_{\textbf{sum}}(s_4, s_1) \\
& s_6 := \texttt{flagdist}_{\textbf{int}}(s_3, s_1) \\
& s_7 := \textbf{pow}\texttt{\^{}}(s_5, s_6) \\
& s_8 := \texttt{flagscan}_{\textbf{sum}}(s_7, s_1) \\
& s := \texttt{pack}_{\textbf{int}}(s_8, s_1)
\end{aligned}$$

which the type system proofs to have typing context

$$\begin{aligned}
\Pi = [\quad & s_0 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_1 \mapsto \{\textbf{bool}\}^{\text{b}} \\
, \quad & s_2 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_3 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_4 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_5 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_6 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_7 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s_8 \mapsto \{\textbf{int}\}^{\text{b}} \\
, \quad & s \mapsto \{\textbf{int}\}^{\text{b}} \\
]
\end{aligned}$$

and thus the output type $\Pi(s) = \{\textbf{int}\}^{\text{b}}$.

First, we let the block size $B = l + 1$. We get the initial buffer store $\Sigma_{init}(\Pi) = \Sigma_0$, initial accumulator store $A_{init}(\Phi) = A_0$, and initial cursor map $\chi_{init}(\Phi) = \chi_0$ which expands to

$$A_0 = [s_1 \mapsto 0, s_5 \mapsto 0, s_8 \mapsto 0]$$

$$
\begin{array}{rlll}
\Sigma_0 = & [s_{ctrl} & \mapsto & {}_0[0,1]_{\checkmark} \\
& s_0 & \mapsto & {}_0[\,]_{*} \\
& ,s_1 & \mapsto & {}_0[\,]_{*} \\
& ,s_2 & \mapsto & {}_0[\,]_{*} \\
& ,s_3 & \mapsto & {}_0[\,]_{*} \\
& ,s_4 & \mapsto & {}_0[\,]_{*} \\
& ,s_5 & \mapsto & {}_0[\,]_{*} \\
& ,s_6 & \mapsto & {}_0[\,]_{*} \\
& ,s_7 & \mapsto & {}_0[\,]_{*} \\
& ,s_8 & \mapsto & {}_0[\,]_{*} \\
& ,s & \mapsto & {}_0[\,]_{*} \\
&] & &
\end{array}
$$

χ_0	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s	s_{ctrl}
s_0	−	0	−	−	−	−	−	−	−	−	∞
s_1	−	−	−	−	0	0	0	−	0	0	∞
s_2	−	−	−	−	0	−	−	−	−	−	∞
s_3	−	−	−	−	−	−	0	−	−	−	∞
s_4	−	−	−	−	−	0	−	−	−	−	∞
s_5	−	−	−	−	−	−	−	0	−	−	∞
s_6	−	−	−	−	−	−	−	0	−	−	∞
s_7	−	−	−	−	−	−	−	−	0	−	∞
s_8	−	−	−	−	−	−	−	−	−	0	∞
s	−	−	−	−	−	−	−	−	−	−	0

We can then dispatch the constant streams s_0, s_2 and s_3 and obtain

$$\Phi \vdash_{l+1} \langle \Sigma_0, A_0, \chi_0 \rangle \rightarrow^{*}_{s_0;s_2;s_3} \langle \Sigma_1, A_0, \chi_0 \rangle \ ! \ \varpi_1$$

where Σ_1 expands to

$$
\begin{array}{rlll}
\Sigma_1 = \Sigma_0 & [s_0 & \mapsto & {}_0[l,l]_{\checkmark} \\
& ,s_2 & \mapsto & {}_0[1,1]_{\checkmark} \\
& ,s_3 & \mapsto & {}_0[2,2]_{\checkmark} \\
&] & &
\end{array}
$$

and $\varpi_1 = \langle 3,3,3 \rangle$. We can now fire s_1 yielding

$$\Phi \vdash_{l+1} \langle \Sigma_1, A_0, \chi_0 \rangle \rightarrow_{s_1} \langle \Sigma_2, A_0, \chi_2 \rangle \ ! \ \varpi_2$$

$$\Sigma_2 = \Sigma_1[s_0 \mapsto {}_1[\,]_{\checkmark}, s_1 \mapsto {}_0[\overbrace{0,\ldots,0}^{l},1]_{\checkmark}]$$
$$\chi_2 = \chi_0[(s_0, s_1) \mapsto 1)]$$

$$\varpi_2 = \langle 1 + l, 1, l + 3 \rangle$$

This opens up the possibility to fire $s_4; s_6$:

$$\Phi \vdash_{l+1} \langle \Sigma_2, A_0, \chi_2 \rangle \rightarrow^*_{s_4;s_6} \langle \Sigma_3, A_0, \chi_3 \rangle \; ! \; \varpi_3$$

$$\begin{aligned} \Sigma_3 = \Sigma_2 \; [s_2 \; &\mapsto \; {}_1[]_\checkmark \\ s_3 \; &\mapsto \; {}_1[]_\checkmark \\ , s_4 \; &\mapsto \; {}_0[\overbrace{1, \ldots, 1}^{l+1}]_\checkmark \\ , s_6 \; &\mapsto \; {}_0[\overbrace{2, \ldots, 2}^{l+1}]_\checkmark \\ ] & \end{aligned}$$

$$\chi_3 = \chi_2[(s_2, s_4) \mapsto 1, (s_3, s_6) \mapsto 1]$$

$$\varpi_3 = \langle 2 \cdot (l + 1), 2, 3l + 5 \rangle$$

We begin to see a pattern. Each transition costs $O(l)$ work, 1 step and $O(l)$ space, so the constant-sized schedule $s_5; s_7; s_8; s$ yields the result in s with complexity $\varpi' = \langle W, D, M \rangle$

$$W = O(l)$$
$$D = O(1)$$
$$M = O(l)$$

If we have that the number of processors $P = l + 1$, we can derive the complexities

$$T = O(1 \log(l + 1)) = O(\log l)$$

$$S = l$$

We recall the cost model for this example from the high level language, we can derive the equivalent high level complexities

$$T' = O(l/(l + 1) + \log(l + 1)) = O(\log l)$$

$$S' = O(\min(P, l)) = O(l)$$

Which are seen to be the same as the low level complexities. It is therefore, as expected, possible to express the iota-square-sum example in the target language with ideal complexity given more than enough processors. We now turn our attention to the more interesting case where $P = 1$. We let the block size $B = 1$ so that $B = O(P)$ and try again. If we choose the right schedule we get

$$W = O(l)$$

$$D = O(l)$$

$$M = O(1)$$

as expected, but another schedule would yield

$$M = O(n)$$

which is definitely not desirable. Scheduling is crucial for the correctness of the cost model of the source language.

4.6 Recursion

If we had recursion in the source language, we would need named sub-routines containing in the target language with the interpretation that we can dynamically allocate and deallocate more buffers. We can simulate recursion in the source language by statically unfolding the recursive function. It is almost always the case that recursion depth is logarithmic for data parallel algorithms [5], and assuming this that, a recursive function requires a logarithmic increase in the number of buffers we allocate, which potentially increases the space complexity by a logarithmic factor. If there are no unbounded buffer types in the sub-routine, we can quickly conclude that the space increase is constant and therefore irrelevant.

4.7 Evaluation

We have described a target language syntax, types and semantics. Target language expressions are a list of stream definitions, where each stream is defined as a transformation of 0 or more input streams, with no cyclic definitions. All types are either bounded or unbounded buffers of primitive values. The semantic of an expression is a transition system on buffer stores, where each transition fires a stream transformer, transforming blocks of the input stream to a block in the output stream. The target language is data parallel, where the parallelism is exposed by the block size. The semantics is augmented with a cost model, that is hopefully believable and allows the example

$$\sum_{i=0}^{l-1} i^2$$

to be expressed and interpreted with ideal space and time complexities. We therefore deem the target language as a suitable backend for the source language, and what is left is to provide an actual translation of source to target language expressions.

5 Transformation

The transformation from source language to target language given in this section involves many technical details that are important for the correctness of the transformation, and therefore an important part of the thesis results. However, since we do provide any proofs, it is not so important for the reader to understand all the details. In particular the section about operation transformation may safely be skimmed when reading the thesis as a whole.

5.1 Value representation

There are many ways to define a representation for sequences and lists. There is no notion of nested sequences or lists in a vectorized language, so we must define an accurate representation of high-level values as streams, before we can define an accurate vectorization transformation on expression. This section is devoted to analyzing the different choices and selecting the most promising ones.

The choice of value representation plays a huge part on how the rest of the transformation is shaped. We cannot use the usual segment descriptor representation as presented in section 1.2.1 in the context of sequences, since the length of a sequence are not known until the sequence has been used.

Scalars Primitive values in the target language do not exist by themselves. They arise as a transformation of the control stream. A scalar value is therefore represented as a primitive stream given a bounded buffer. We call such a stream a scalar stream, because it is the representation of one or more source language scalar values. The scalar a in the source language will be represented as a stream s_0 of a's, which can be visualized as:

$$s_0 = a \quad a \quad a \quad a \quad \cdots \quad a \quad _$$

where the last element $_$ is a dummy element, the meaning of which we will explain when we describe the representation of sequences. This is a visualization of the entire stream. It is important to note that only a part of the entire stream is stored in memory at any given time. We think of time as moving from left to right when visualizing streams.

The concrete number of a's in the stream depends on the parallel degree, defined by the control stream. Thus if a literal $\overline{a}$ appears inside a map

$$\{\overline{n} : x \textbf{ in } \&10\}$$

The stream of $\overline{n}$ will hold 11 n's - 10 from the parallel degree and 1 additional dummy value in the end.

Lists The vectorization of lists is an interesting subject on its own with many possibilities, but it is not the real focus of this thesis. We use a virtually segmented representation as defined in [16], since it facilitates provably work-preserving vectorization, except for vectorization of the list constructor. In essence, every list is vectorized to a pair (st_0, st_1) where st_0 is referred to as the segment descriptor and st_1 is referred to as the data streams. st_0 is itself a pair (s_2, s_3) where s_2 is a stream of segment starts, and s_1 is a stream of segment lengths. These two streams defines a segmentation on the streams st_1, which holds the actual elements of the list. This definition

is recursive in that st_1 mast itself be a pair of segment descriptor and data vector (st_4, st_5) in which case st_0 defines a segmentation on the segment descriptor st_4, and st_4 defines a segmentation of st_5. We can visualize one of the many representation of $[[\texttt{a}, \texttt{b}], [\texttt{c}, \texttt{d}, \texttt{e}], [\texttt{e}, \texttt{f}, \texttt{h}]]$ as

$$
\begin{array}{llllllll}
s_0 = 0 & & & & & & & _ \\
s_1 = 3 & & & & & & & _ \\
s_2 = 0 & & 2 & & & 5 & & \\
s_3 = 2 & & 3 & & & 3 & & \\
s_4 = \texttt{a} & \texttt{b} & \texttt{c} & \texttt{d} & \texttt{e} & \texttt{f} & \texttt{g} & \texttt{h}
\end{array}
$$

Since we use virtual segment descriptors, there are multiple representations of each source language lists. This representation is said to be *normalized*, because the segments appear directly after each other. The structure of the streams is $((s_0, s_1), ((s_2, s_3), s_4))$. (s_0, s_1) is the top-level segment descriptor defining a single segment starting and 0 and with length 3. (s_2, s_3) is the second segment descriptors, and it defines 3 segments in s_4. In contrast to sequences, lists are always fully allocated in memory. This affects our choice of representation of lists in that the buffers of s_2, s_3 and s_4 are unbounded. However, the top most segment descriptor is bounded just like scalar streams. In fact, s_2 and s_3 can be thought of as scalar streams for all purposes.

The number of elements in the top-most segment descriptor must always be the same as the number of elements in the control stream.

Sequences At the very basics, a primitive sequence $\{\pi\}$ is simply a primitive stream. Reading a file into a stream of characters, or generating the numbers from 0 to n are examples of primitive streams. Because we may have multiple primitive sequences in parallel sub-computations, we must delimit the streams they represent. For this purpose we add a special symbol to the primitive type that represent a delimiter, so now we need to represent a stream of $\pi + 1$. Since we do not have sum types, we can use an equivalent representation by tagging each element in the stream with a boolean flag indicating left or right injection, and in the case of right injection (a delimiter), we can simply ignore the left projection. It is a dummy value. A primitive sequence is therefore represented as a primitive stream, and a boolean stream. For example, the value

$$\{0, 1, 2\}$$

is represented as

$$
\begin{array}{llll}
s_0 = \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} \\
s_1 = 0 & 1 & 2 & _
\end{array}
$$

And in the case of parallel computations, we could for example have the following three sequences in parallel sub-computations:

$$\{0, 1, 2\} \quad \{\} \quad \{10, 20\}$$

which would be represented by the streams

$$
\begin{array}{llllllll}
s_0 = \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{t} \\
s_1 = 0 & 1 & 2 & _ & _ & 10 & 20 & _
\end{array}
$$

When we have a nested sequences things get a little more complicated. Reading a file into a stream of lines, where each line is streamed is an example of a nested

sequence. We need to delimit nested sequences just like we need to delimit primitive sequences. The most intuitive way to define delimiters on nested sequences is to delimit the delimiter stream of the sub-sequences. For instance, the two parallel nested sequences

$$\{\{\texttt{a,b}\},\{\texttt{c,d,e}\},\{\texttt{f,g,h}\}\}$$
and
$$\{\{\texttt{i,j,k,l}\},\{\texttt{m,n}\}\}$$

could be represented as

$s_0 =$	f	f	f	f	f	f	f	f	f	f	t	f	f	f	f	f	f	f	t
$s_1 =$	f	f	t	f	f	f	t	f	f	f	_	f	f	f	f	t	f	f	_
$s_3 =$	a	b	_	c	d	e	_	f	g	h	_	i	j	k	l	_	m	n	_

This representation has the nice property that it can easily be thought of as a sequence of **char** + **1** + **1**, but it has the unfortunate property that the sub-sequences of a sequence of pairs, i.e. $\{\sigma_0 * \sigma_1\}$ do not necessarily share delimiters. The number of **f**'s in a segment of the form $\mathbf{f}, \ldots, \mathbf{f}, \mathbf{t}$ depends on the number of elements stored in the bottom-most stream of the stack, and the two stacks of σ_0 and σ_1 may be very different. The implication is that we must lift all pair types to the top-level of the type outside any sequence, and remember that they are actually zipped, and at what level. Every projection will then have it's own complete stack of delimiter streams. This is doable, but leads to overly complex representation and subsequent vectorization of expressions. The following is a better representation:

$s'_0 =$	f			f				f			t	f					f		t
$s_1 =$	f	f	t	f	f	f	t	f	f	f	_	f	f	f	f	t	f	f	_
$s_2 =$	a	b	_	c	d	e	_	f	g	h	_	i	j	k	l	_	m	n	_

If we only stream the first **f** on top of each underlying segment, the delimitation is unique no matter the size of the segments in the underlying stream stack. In this way we can share the delimiters of $\{\sigma_0 * \sigma_1\}$ in one boolean stream for all σ_0 and σ_1. Sometimes we really need the original long top-level delimiter stream that matches the size of the one below. This is why we have the `reconstruct` transformer. In the example, we can "reconstruct" s_0 from s'_0 and s_1 by the definition s_0 `:=` `reconstruct`(s'_0, s_1):

$s'_0 =$	f			f				f			t	f					f		t
$s_1 =$	f	f	t	f	f	f	t	f	f	f	_	f	f	f	f	t	f	f	_
$s_0 =$	f	f	f	f	f	f	f	f	f	f	t	f	f	f	f	f	t	f	t

5.2 Type transformation

We will begin by describing a vectorization of types.

Any given source type $\sigma \in \mathbf{Typ}$ can be transformed to a target language type ν by type vectorization. Vectorization of primitive types in the source language is done by mapping the primitive type π to a buffer type μ. If it is bounded or unbounded depends on whether or not the primitive should be tabulated. Types should be tabulated if they exist inside a vector type $[\tau]$. Streams are vectorized by separately vectorizing the element type, and attaching a boolean bounded buffer to it representing stream delimiters. Vectors are vectorized by separately vectorizing the element type, and attaching a pair of integer buffers to it representing virtual segmentation by segment

offsets and segment lengths. The integer buffers are unbounded if the type is within a vector type, and bounded otherwise.

$$\mathcal{V}\langle\cdot\rangle : \textbf{Typ} \to \textbf{VTyp}$$

$$\mathcal{V}_c\langle\cdot\rangle : \textbf{CTyp} \to \textbf{VTyp}$$

$$\begin{aligned}
&\mathcal{V}\langle\sigma_0 * \sigma_1\rangle = \mathcal{V}\langle\sigma_0\rangle * \mathcal{V}\langle\sigma_1\rangle\\
&\mathcal{V}\langle\mathbf{1}\rangle = \mathtt{1}\\
&\mathcal{V}\langle\{\sigma\}\rangle = \{\textbf{bool}\}^{\mathrm{b}} * \mathcal{V}\langle\sigma\rangle\\
&\mathcal{V}\langle\pi\rangle = \{\pi\}^{\mathrm{b}}\\
&\mathcal{V}\langle[\tau]\rangle = \{\textbf{int}\}^{\mathrm{b}} * \{\textbf{int}\}^{\mathrm{b}} * \mathcal{V}_c\langle\tau\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{V}_c\langle\tau_0 * \tau_1\rangle = \mathcal{V}_c\langle\tau_0\rangle * \mathcal{V}_c\langle\tau_1\rangle\\
&\mathcal{V}_c\langle\mathbf{1}\rangle = \mathtt{1}\\
&\mathcal{V}_c\langle\pi\rangle = [\pi]\\
&\mathcal{V}_c\langle[\tau]\rangle = \{\textbf{int}\}^{\mathrm{u}} * \{\textbf{int}\}^{\mathrm{u}} * \mathcal{V}_c\langle\tau\rangle
\end{aligned}$$

5.3 Expression transformation

For the purpose of translating source language expressions to streaming systems, we define a monad for building up a lists of stream definitions compositionaly. Since we need to generate unique stream identifiers, we take **SId** to be isomorphic to the natural numbers

$$\textbf{SId} = \{s_n \mid n \in \mathbf{N}\}$$

and define the length of a list of definitions

$$|\epsilon| = 0$$

$$|\phi;\Phi| = 1 + |\Phi|$$

We then define a monoid on definition lists. Two lists are naturally composed by appending the definitions of one list with the other.

$$\begin{aligned}
\epsilon +\!\!+\, \Phi_1 &= \Phi_1\\
(\phi;\Phi_0) +\!\!+\, \Phi_1 &= \phi;(\Phi_0 +\!\!+\, \Phi_1)
\end{aligned}$$

Append is clearly a monoid with identity element ϵ. We can now define a monad:

$$\textbf{Gen } X = \textbf{Eqs} \to (X \times \textbf{Eqs})$$

where **Gen** X is a function from a (big) list of definitions to X and a list of new definitions to add. We have the two natural transformations return $\eta_{\textbf{Gen}}$ and bind $\star_{\textbf{Gen}}$ defined as

$$\eta_{\textbf{Gen}} : X \to \textbf{Gen } X$$

$$\eta_{\textbf{Gen}}\ x = \lambda\Phi.(x,\epsilon)$$

$$\star_{\mathbf{Gen}} : \mathbf{Gen}\ X \to (X \to \mathbf{Gen}\ Y) \to \mathbf{Gen}\ Y$$

$$\begin{aligned} q \star_{\mathbf{Gen}} f = \lambda\Phi.\text{let}\ & (x, \Phi_0) = q\ \Phi \\ & (y, \Phi_1) = f\ x\ (\Phi + \Phi_0) \\ \text{in}\ & (y, \Phi_0 + \Phi_1) \end{aligned}$$

We can then define the function *emit* that adds a stream definition defining a fresh stream identifier as, and returns the identifier for further use:

$$emit : \mathbf{Trans} \to \mathbf{SId}^k \to \mathbf{Gen\ SId}$$

$$emit\ t\ (s_0, \ldots, s_{k-1}) = \lambda\Phi.(s', s' := t(s_0, \ldots s_{k-1}) \quad \text{where } s' = s_{|\Phi|})$$

The function *defs*q extracts the list of definitions from the monad

$$defs = \lambda\Phi.(\Phi, \epsilon)$$

We can then build lists of definitions incrementally by repeated application of $\star_{\mathbf{Gen}}$, $\eta_{\mathbf{Gen}}$, and *emit*. For example:

$$emit\ t_0\ () \star_{\mathbf{Gen}} (\lambda s_0.emit\ t_1\ (s_0) \star_{\mathbf{Gen}} (\lambda s_1.emit\ t_2\ (s_0, s_1) \star_{\mathbf{Gen}} defs))$$

When applied to $(\epsilon, 0)$ yields Φ where Φ is the following list of definitions:

$$\begin{aligned} s_0 &:= t_0(); \\ s_1 &:= t_1(s_0); \\ s_2 &:= t_2(s_0, s_1) \end{aligned}$$

Writing definition lists by $\star_{\mathbf{Gen}}$, *emit* and anonymous functions is tedious, so we adopt a notation similar to Haskell's do notations:

$$\begin{aligned} \text{do}\ \ & s_0 \leftarrow emit\ t_0\ (); \\ & s_1 \leftarrow emit\ t_1\ (s_0); \\ & emit\ t_2\ (s_0, s_1) \end{aligned}$$

which translates into

$$emit\ t_0\ () \star_{\mathbf{Gen}} (\lambda s_0.emit\ t_1\ (s_0) \star_{\mathbf{Gen}} (\lambda s_1.emit\ t_2\ (s_0, s_1)))$$

The transformation from source language to target language is given by the transformation function $\langle\!\langle \cdot \rangle\!\rangle$ parametrized by a source language typing context Γ. The typing context is used when transforming apply-to-each. The promoted variables in the **using** term must be distributed over the sequence. This require that we know their type. The transformation function on expressions is defined as

$$\langle\!\langle \cdot \rangle\!\rangle_\Gamma : \mathbf{Exp} \to \mathbf{SSys}$$

$$\begin{aligned} \langle\!\langle e \rangle\!\rangle_\Gamma = q\ & (\epsilon, 0) \\ & \text{where } q = \text{do}\ \ st \leftarrow \mathcal{E}\langle\!\langle e \rangle\!\rangle_\Gamma\ []\ s \\ & \qquad\qquad\quad \Phi \leftarrow defs \\ & \qquad\qquad\quad \eta_{\mathbf{Gen}}\ (\texttt{let}\ \Phi\ \texttt{in}\ st) \end{aligned}$$

The type invariant is:
If

$$\Gamma \vdash e : \sigma$$

then

$$\langle\!\langle e \rangle\!\rangle_\Gamma = \texttt{let}\ \Phi\ \texttt{in}\ st$$

s.t.

$$\vdash \Phi : \Pi$$

and

$$\Pi \vdash st : \mathcal{V}\langle \sigma \rangle$$

Which can be proved by induction on the syntax of e.

The function $\mathcal{E}\langle\!\langle \cdot \rangle\!\rangle_\Gamma$ builds up the definition list Φ and returns the resulting stream identifier tree st, that holds the output streams of the system.

$$\mathcal{E}\langle\!\langle \cdot \rangle\!\rangle_\Gamma : \mathbf{Exp} \to (\mathbf{VarId} \rightharpoonup \mathbf{PIdT}) \to \mathbf{SId} \to \mathbf{Q\ PIdT}$$

where $\mathcal{E}\langle\!\langle e \rangle\!\rangle_\Gamma\ \delta\ s_{ctrl} = q$ is the transformation of e given a finite map from source language variables to stream identifier trees to a stream identifier tree inside a **Gen** monad. In other words, it may emit stream definitions.

$$\begin{array}{rcl}
\mathcal{E}\langle\!\langle \overline{a} \rangle\!\rangle_\Gamma\ \delta\ s & = & const\ s\ a \\
\mathcal{E}\langle\!\langle o\ e \rangle\!\rangle_\Gamma\ \delta\ s & = & \text{do}\ \ st \leftarrow \mathcal{E}\langle\!\langle e \rangle\!\rangle_\Gamma \delta\ \ s \\
 & & \qquad \mathcal{O}\langle\!\langle o \rangle\!\rangle\ s\ st \\
\mathcal{E}\langle\!\langle \mathbf{()} \rangle\!\rangle_\Gamma\ \delta\ s & = & \eta_{\mathbf{Gen}}\ \texttt{()} \\
\mathcal{E}\langle\!\langle \mathbf{(}e_0\mathbf{,}\ e_1\mathbf{)} \rangle\!\rangle_\Gamma\ \delta\ s & = & \text{do}\ \ st_0 \leftarrow \mathcal{E}\langle\!\langle e_0 \rangle\!\rangle_\Gamma\ \delta\ s \\
 & & \qquad st_1 \leftarrow \mathcal{E}\langle\!\langle e_1 \rangle\!\rangle_\Gamma\ \delta\ s \\
 & & \qquad \eta_{\mathbf{Gen}}\ \texttt{(}st_0\texttt{,}\ st_1\texttt{)} \\
\mathcal{E}\langle\!\langle \mathbf{fst}\ e \rangle\!\rangle_\Gamma\ \delta\ s & = & \text{do}\ \ \texttt{(}st_0\texttt{,}\ st_1\texttt{)} \leftarrow \mathcal{E}\langle\!\langle e \rangle\!\rangle_\Gamma\ \delta\ s \\
 & & \qquad \eta_{\mathbf{Gen}}\ st_0 \\
\mathcal{E}\langle\!\langle \mathbf{snd}\ e \rangle\!\rangle_\Gamma\ \delta\ s & = & \text{do}\ \ \texttt{(}st_0\texttt{,}\ st_1\texttt{)} \leftarrow \mathcal{E}\langle\!\langle e \rangle\!\rangle_\Gamma\ \delta\ s \\
 & & \qquad \eta_{\mathbf{Gen}}\ st_1 \\
\mathcal{E}\langle\!\langle \mathbf{let}\ x\ \mathbf{=}\ e_0\ \mathbf{in}\ e_1 \rangle\!\rangle_\Gamma\ s & = & \text{do}\ \ st_0 \leftarrow \mathcal{E}\langle\!\langle e_0 \rangle\!\rangle_\Gamma\ \delta\ s \\
 & & \qquad \mathcal{E}\langle\!\langle e_1 \rangle\!\rangle_\Gamma\ \delta[x \mapsto st_0]\ s
\end{array}$$

$$\mathcal{E}\langle\!\langle \mathbf{\{}e_0\ \mathbf{:}\ x\ \mathbf{in}\ e_1\ \mathbf{using}\ x_1, \ldots, x_k\mathbf{\}} \rangle\!\rangle_\Gamma\ \delta\ s\ =$$

$$\begin{array}{ll}
\text{do} & \texttt{(}s'\texttt{,}\ st_1\texttt{)} \leftarrow \mathcal{E}\langle\!\langle e_1 \rangle\!\rangle_\Gamma\ \delta\ s \\
 & st_2 \leftarrow \mathcal{E}\langle\!\langle e_2 \rangle\!\rangle_\Gamma\ s \\
 & st'_1 \leftarrow dist_{\tau_1}\ \delta(x_1)\ s' \qquad (\Gamma(x_1) = \tau_1) \\
 & \quad \vdots \\
 & st'_k \leftarrow dist_{\tau_k}\ \delta(x_k)\ s' \qquad (\Gamma(x_k) = \tau_k) \\
 & st_0 \leftarrow \mathcal{E}\langle\!\langle e_0 \rangle\!\rangle_\Gamma\ \delta[x \mapsto st_1, x_1 \mapsto st'_1, \ldots, x_k \mapsto st'_k]\ s' \\
 & \eta_{\mathbf{Gen}}\ \texttt{(}s'\texttt{,}\ st_0\texttt{)}
\end{array}$$

For the purpose of generating constant literals at any parallel degree, the constant in the target language is actually a scalar operation

$$\oplus ::= \cdots \mid \mathbf{const}_a$$

with the type and semantic

$$\forall a : \pi_1 \ . \ \mathbf{const}_a : \mathbf{bool} \rightarrow \pi$$

$$[\![\mathbf{const}_a]\!] \ _ = a$$

const is then a function that generates a constant stream distributed over to a given control stream:

$$const\ s\ a = emit\ \mathbf{const}_a\text{\^{}}\ s$$

$dist_\tau$ is a function that distributes non-delimited streams over a control stream guided by a high-level type τ. It distributes the simple stream if $\tau = \pi$, and it distributes the top-level segment descriptor streams if $\tau = [\tau_0]$. Since lists are virtually segmented, we don't need to distribute the remainder of the list.

$$dist_\tau : \mathbf{PIdT} \to \mathbf{SId} \to \mathbf{W\ PIdT}$$

$$\begin{aligned}
dist_{\tau_0 * \tau_1}\ (st_0,\ st_1)\ s &= \text{do}\ \ st'_0 \leftarrow dist_{\tau_0}\ st_0\ s \\
&\qquad\ \ \ st'_1 \leftarrow dist_{\tau_1}\ st_1\ s \\
&\qquad\ \ \ \eta_{\mathbf{Gen}}\ (st'_0,\ st'_1) \\
dist_{\mathbf{1}}\ ()\ s &= \eta_{\mathbf{Gen}}\ () \\
dist_{[\tau]}\ (st_0,\ st_1)\ s &= \text{do}\ \ st'_0 \leftarrow dist_{\mathbf{int*int}}\ st_0\ s \\
&\qquad\ \ \ \eta_{\mathbf{Gen}}\ (st'_0,\ st_1) \\
dist_\pi\ s_0\ s &= emit\ \texttt{flagdist}_\pi\ s_0\ s
\end{aligned}$$

The functions $\mathcal{E}\langle\!\langle \cdot \rangle\!\rangle_\Gamma$ and $dist_\tau$ is only the tip of iceberg when vectorizing source language expressions. The trickiest part is to vectorize the primitive operations **Op**. Each individual operation needs to be vectorized with great care in order to maintain correct alignment of delimiters and data, and to avoid impossible situations with respect to streaming. The operation vectorization function has the signature

$$\mathcal{O}\langle\!\langle \cdot \rangle\!\rangle : \mathbf{Op} \to \mathbf{SId} \to \mathbf{PIdT} \to \mathbf{Gen\ PIdT}$$

and we will give its definition with explanation one operation at a time.

Primitive operations Most scalar operations are trivially vectorized to their vector-versions

$$\mathcal{O}\langle\!\langle + \rangle\!\rangle\ s\ (s_0,\ s_1) = emit\ (+\text{\^{}})\ s_0\ s_1$$

$$\mathcal{O}\langle\!\langle * \rangle\!\rangle\ s\ (s_0,\ s_1) = emit\ (*\text{\^{}})\ s_0\ s_1$$

But for some scalar operations, mainly division, we must make sure not to apply the operation on unknown dummy values. Such an application could result in unexpected runtime failure. In order to deal with this problem, we introduce a new scalar operation *conditional write*

$$\oplus ::= \cdots \mid \mathbf{condw}_a$$

with the type and semantics

$$\forall a : \pi \ . \ \mathbf{condw}_a : \pi * \mathbf{bool} \rightarrow \pi$$

$$[\![\mathbf{condw}_a]\!]\ (a_0, b_1) = \begin{cases} a & b_1 = 1 \\ a_0 & b_1 = 0 \end{cases}$$

We can then use this operation to write safe values to the dummy value places on a per-operation basis. In the case of division, we must make sure to write a non-zero value, so:

$$\begin{aligned} \mathcal{O}\langle\!\langle \mathbf{div} \rangle\!\rangle\ s\ (s_0,\ s_1) = \text{do}\ \ & s_2 \leftarrow \mathit{emit}\ \mathbf{condw}_1^{\wedge}\ (s_1, s) \\ & \mathit{emit}\ (\mathbf{div}^{\wedge})\ (s_0, s_2) \end{aligned}$$

Scan and reduction The scan operation is also pretty straight-forward. Since scan is already a primitive in the target language, we can easily define vectorization of scan as

$$\mathcal{O}\langle\!\langle \mathbf{scan}_{\otimes} \rangle\!\rangle\ s\ (s_0,\ s_1) = \mathit{emit}\ \mathtt{flagscan}_{\otimes}\ (s_0, s_1)$$

Reduction is not a primitive on the other hand, but it is easily vectorized by simply packing the result of a scan operation at the position of the segment flags

$$\mathcal{O}\langle\!\langle \mathbf{reduce}_{\otimes} \rangle\!\rangle\ s\ (s_0,\ s_1) = \mathit{flagreduce}_{\otimes}\ (s_0, s_1)$$

where

$$\begin{aligned} \mathit{flagreduce}_{\otimes}\ s_0\ s_1 = \text{do}\ \ & s_2 \leftarrow \mathit{emit}\ \mathtt{flagscan}_{\otimes}\ (s_0, s_1) \\ & \mathit{emit}\ \mathtt{pack}_{\pi}\ s_2\ s_1 \qquad \text{where } \otimes : \pi \end{aligned}$$

For later purposes we also define a non-segmented scan operation

$$\begin{aligned} \mathit{scan}_{\otimes}\ s_0 = \text{do}\ \ & s_1 \leftarrow \mathit{const}\ 0\ s_0 \\ & \mathit{emit}\ \mathtt{flagscan}_{\mathbf{int}}\ (s_0, s_1) \end{aligned}$$

Iota Similar to division, where we must take care not to divide with a zero when the zero is a dummy value, for iota, we must take care not to generate arbitrary long iota sequences from non-zero dummy values. In fact, every single dummy values must produce a single dummy value as a result, and the only input to iota that satisfies this property is zero, so we will write zeroes on all dummy positions. Iota itself is vectorized into a segmented sum-scan of 1's.

$$\begin{aligned} \mathcal{O}\langle\!\langle \mathbf{iota} \rangle\!\rangle\ s\ s_0 = \text{do}\ \ & s_1 \leftarrow \mathit{emit}\ \mathbf{condw}_0^{\wedge}\ (s_0, s) \\ & s_2 \leftarrow \mathit{emit}\ \mathtt{2flags}\ s_1 \\ & s_3 \leftarrow \mathit{flagiota}\ s\ s_2 \\ & \eta_{\mathbf{Gen}}\ (s_2,\ s_3) \end{aligned}$$

where *flagiota* is a composite transition that generates an iota sequence from a flag sequence.

$$\begin{aligned} \mathtt{flagiota}\ s\ s_0 = \text{do}\ \ & s_1 \leftarrow \mathit{const}\ 1\ s \\ & \mathit{emit}\ \mathtt{flagscan}_{\mathbf{int}}\ (s_1, s_0) \end{aligned}$$

Length The length of a concrete list is given by its length stream in its segment descriptor. So

$$\mathcal{O}\langle\!\langle \mathbf{len}_{\tau}^{[]} \rangle\!\rangle\ s\ ((_,\ s_0),\ _) = s_0$$

The length of a sequence is given by counting zeroes in its delimiter stream. This can be done by a segmented sum of 1's. We can then define the vectorization of $\mathbf{len}^{\{\}}_{\sigma}$ as

$$\mathcal{O}\langle\!\langle \mathbf{len}^{\{\}}_{\sigma} \rangle\!\rangle\ s\ \texttt{(}s_0\texttt{,}\ _\texttt{)} = \mathit{2lens}\ s_0$$

where

$$\begin{aligned} \mathit{2lens}\ s_0 = \text{do}\ \ & s_1 \leftarrow \mathbf{const}_1\texttt{\^{}}\ s_0 \\ & \mathit{flagreduce}_{\mathbf{sum}}\ (s_1, s_0) \end{aligned}$$

Zip Zipping two streams is a simple matter of choosing one of the delimiter streams and attach it two both of them:

$$\mathcal{O}\langle\!\langle \mathbf{zip}_{\sigma_0,\sigma_1} \rangle\!\rangle\ s\ \texttt{((}s_0\texttt{,}\ st_0\texttt{),}\ \texttt{(}s_1\texttt{,}\ st_1\texttt{))} = \eta_{\mathbf{Gen}}\ \texttt{(}s_0\texttt{,}\ \texttt{(}st_0\texttt{,}\ st_1\texttt{))}$$

Of course this does not account for runtime error if the delimiters are different. In order to catch the error we could extend the language with a primitive transition `assert` that reads a stream of bools and fails if any of the elements are zero. We then feed the vector version of scalar equality operator on s_0 and s_1 to this transition.

Elt We can use the $\texttt{segfetch}_{\pi}$ transformer to gather elements from a vectorized list.

$$\begin{aligned} & \mathcal{O}\langle\!\langle \mathbf{elt}^{[]}_{\tau} \rangle\!\rangle\ s\ \texttt{(((}s_0\texttt{,}\ s_1\texttt{),}\ st_0\texttt{),}\ s_3\texttt{)} = \\ & \quad \text{do}\ \ s_4 \leftarrow \mathit{emit}\ \mathbf{condw}_0\texttt{\^{}}\ s_3\ s \\ & \qquad\ \ s_5 \leftarrow \mathit{emit}\ \texttt{(+\^{})}\ s_0\ s_4 \\ & \qquad\ \ s_6 \leftarrow \mathit{const}\ s\ 1 \\ & \qquad\ \ \mathit{segfetch}_{\tau}\ st_0\ s_5\ s_6 \end{aligned}$$

$$\begin{aligned} \mathit{segfetch}_{\pi}\ s_0\ s_1\ s_2 &= \mathit{emit}\ \texttt{segfetch}_{\pi}\ (s_0, s_1, s_2) \\ \mathit{segfetch}_{[\tau]}\ \texttt{(}st_0\texttt{,}\ st_1\texttt{)}\ s_1\ s_2 &= \text{do}\ \ st_0' \leftarrow \mathit{segfetch}_{\mathbf{int*int}}\ st_0\ s_1\ s_2 \\ &\qquad\ \ \eta_{\mathbf{Gen}}\ \texttt{(}st_0'\texttt{,}\ st_1\texttt{)} \\ \mathit{segfetch}_{\mathbf{1}}\ \texttt{()}\ s_1\ s_2 &= \eta_{\mathbf{Gen}}\ \texttt{()} \\ \mathit{segfetch}_{\tau_0*\tau_1}\ \texttt{(}st_0\texttt{,}\ st_1\texttt{)}\ s_1\ s_2 &= \text{do}\ \ st_0' \leftarrow \mathit{segfetch}_{\tau_0}\ st_0\ s_1\ s_2 \\ &\qquad\ \ st_1' \leftarrow \mathit{segfetch}_{\tau_1}\ st_1\ s_1\ s_2 \\ &\qquad\ \ \eta_{\mathbf{Gen}}\ \texttt{(}st_0'\texttt{,}\ st_1'\texttt{)} \end{aligned}$$

For a safer solution, we should assert that the indices in s_4 are greater than zero and element-wise less then the lengths in s_1.

Element retrieval from a nested stream boils down to removing a level of delimiters and packing the remaining stream(s). It is in some sense similar to gather, but we must perform the pack

$$\begin{aligned} & \mathcal{O}\langle\!\langle \mathbf{elt}^{\{\}}_{\sigma} \rangle\!\rangle\ s\ \texttt{((}s_0\texttt{,}\ st_0\texttt{),}\ s_1\texttt{)} = \\ & \quad \text{do}\ \ s_2 \leftarrow \mathit{emit}\ \texttt{flagdist}_{\mathbf{int}}\ (s_1, s_0) \\ & \qquad\ \ s_3 \leftarrow \mathit{flagiota}\ s\ s_0 \\ & \qquad\ \ s_4 \leftarrow \mathit{emit}\ \texttt{(=\^{})}\ s_2\ s_3 \\ & \qquad\ \ \mathit{deeppack}_{\sigma}\ st_0\ s_4 \end{aligned}$$

where

$$
\begin{array}{rcl}
\mathit{deeppack}_{\pi}\ s_0\ s & = & \mathit{emit}\ \mathtt{pack}_{\pi}\ s_0\ s \\
\mathit{deeppack}_{[\tau]}\ (st_0,\ st_1)\ s & = & \text{do}\ \ st_0' \leftarrow \mathit{deeppack}_{\mathbf{int*int}}\ st_0\ s \\
& & \quad\ \ \eta_{\mathbf{Gen}}\ (st_0',\ st_1) \\
\mathit{deeppack}_{\{\tau\}}\ (s_0,\ st_1)\ s & = & \text{do}\ \ st_1' \leftarrow \mathit{deeppack}_{\tau}\ st_1\ s \\
& & \quad\ \ s_1\ \ \leftarrow \mathit{deeppack}_{\mathbf{bool}}\ s_0\ s \\
& & \quad\ \ \eta_{\mathbf{Gen}}\ (s_1,\ st_1') \\
\mathit{deeppack}_{\{\{\sigma\}\}}\ (s_0,\ (s_1,\ st_1))\ s & = & \text{do}\ \ s'\ \ \leftarrow \mathit{emit}\ \mathtt{reconstruct}_{\mathbf{bool}}\ s\ s_1 \\
& & \quad\ \ st_1' \leftarrow \mathit{deeppack}_{\{\sigma\}}\ (s_1,\ st_1)\ s' \\
& & \quad\ \ s_1\ \ \leftarrow \mathit{emit}\ \mathtt{pack}_{\mathbf{bool}}\ s_0\ s \\
& & \quad\ \ \eta_{\mathbf{Gen}}\ (s_1,\ st_1') \\
\mathit{deeppack}_{\mathbf{1}}\ ()\ s & = & \eta_{\mathbf{Gen}}\ () \\
\mathit{deeppack}_{\sigma_0 * \sigma_1}\ (st_0,\ st_1)\ s & = & \text{do}\ \ st_0' \leftarrow \mathit{deeppack}_{\sigma_0}\ st_0\ s \\
& & \quad\ \ st_1' \leftarrow \mathit{deeppack}_{\sigma_1}\ st_1\ s \\
& & \quad\ \ \eta_{\mathbf{Gen}}\ (st_0',\ st_1')
\end{array}
$$

Concat and partition Concatenation is done by removing delimiters in the second delimiter stream from the top. As an example, consider concating the stream stack

$$
\begin{array}{rcccccccccccccccccccc}
s_0 = & \mathbf{f} & & & & & & & \mathbf{f} & & & \mathbf{t} & \mathbf{f} & & & & & & \mathbf{f} & & \mathbf{t} \\
s_1 = & \mathbf{f} & & & \mathbf{f} & & & \mathbf{t} & \mathbf{f} & & & \mathbf{t} & \mathbf{f} & \mathbf{f} & & & & \mathbf{t} & \mathbf{f} & & \mathbf{t} \\
s_2 = & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{t} \\
s_3 = & \mathtt{a} & \mathtt{b} & _ & \mathtt{c} & \mathtt{d} & \mathtt{e} & _ & \mathtt{f} & \mathtt{g} & \mathtt{h} & _ & _ & \mathtt{i} & \mathtt{j} & \mathtt{k} & \mathtt{l} & _ & \mathtt{m} & \mathtt{n} & _
\end{array}
$$

Removing the **t**'s in s_1 where there is no **t**'s in s_0 (reconstructed), and removing s_0 from the stack yields

$$
\begin{array}{rcccccccccccccccccccc}
s_1' = & \mathbf{f} & & & \mathbf{f} & & & & \mathbf{f} & & & \mathbf{t} & \mathbf{f} & \mathbf{f} & & & & & \mathbf{f} & & \mathbf{t} \\
s_2 = & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & \mathbf{f} & \mathbf{f} & \mathbf{t} \\
s_3 = & \mathtt{a} & \mathtt{b} & _ & \mathtt{c} & \mathtt{d} & \mathtt{e} & _ & \mathtt{f} & \mathtt{g} & \mathtt{h} & _ & _ & \mathtt{i} & \mathtt{j} & \mathtt{k} & \mathtt{l} & _ & \mathtt{m} & \mathtt{n} & _
\end{array}
$$

which gives the desired result. If we concat again, using the same technique we get

$$
\begin{array}{rcccccccccccccccccccc}
s_2' = & \mathbf{f} & \mathbf{f} & & \mathbf{f} & \mathbf{f} & \mathbf{f} & & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{t} & & \mathbf{f} & \mathbf{f} & \mathbf{f} & \mathbf{f} & & \mathbf{f} & \mathbf{f} & \mathbf{t} \\
s_3 = & \mathtt{a} & \mathtt{b} & _ & \mathtt{c} & \mathtt{d} & \mathtt{e} & _ & \mathtt{f} & \mathtt{g} & \mathtt{h} & _ & _ & \mathtt{i} & \mathtt{j} & \mathtt{k} & \mathtt{l} & _ & \mathtt{m} & \mathtt{n} & _
\end{array}
$$

And we see that there are dangling dummy values in s_3 with no corresponding flag in the delimiter stream s_2. We must therefore pack out these values.

$$
\begin{array}{rl}
\mathcal{O}\langle\!\langle \mathbf{concat}_{\sigma} \rangle\!\rangle\ s\ (s_0,\ (s_1,\ st)) = \text{do} & s_2\ \ \leftarrow \mathit{emit}\ \mathtt{reconstruct}_{\mathbf{bool}}\ s_0\ s_1 \\
& s_3\ \ \leftarrow \mathit{emit}\ \mathbf{xor}\hat{}\ s_1\ s_2 \\
& s_4\ \ \leftarrow \mathit{emit}\ \mathbf{not}\hat{}\ s_3 \\
& s_5\ \ \leftarrow \mathit{emit}\ \mathtt{pack}_{\mathbf{bool}}\ s_0\ s_4 \\
& st' \leftarrow \mathit{shallowpack}_{\sigma}\ st\ s_4 \\
& \eta_{\mathbf{Gen}}\ (s_5,\ st')
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{shallowpack}_{\pi}\ s_0\ s & = & \mathit{emit}\ \texttt{pack}_{\pi}\ s_0\ s \\
\mathit{shallowpack}_{[\tau]}\ (st_0,\ st_1)\ s & = & \text{do}\ \ st'_0 \leftarrow \mathit{shallowpack}_{\mathbf{int*int}}\ st_0\ s \\
 & & \qquad \eta_{\mathbf{Gen}}\ (st'_0,\ st_1) \\
\mathit{shallowpack}_{\{\sigma\}}\ st_0\ s & = & \eta_{\mathbf{Gen}}\ st_0 \\
\mathit{shallowpack}_{\mathbf{1}}\ ()\ s & = & \eta_{\mathbf{Gen}}\ () \\
\mathit{shallowpack}_{\sigma_0 * \sigma_1}\ (st_0,\ st_1)\ s & = & \text{do}\ \ st'_0 \leftarrow \mathit{shallowpack}_{\sigma_0}\ st_0\ s \\
 & & \qquad st'_1 \leftarrow \mathit{shallowpack}_{\sigma_1}\ st_1\ s \\
 & & \qquad \eta_{\mathbf{Gen}}\ (st'_0,\ st'_1)
\end{array}
$$

We have now packed s_3. Partition should be able to reverse this process given the correct boolean sequence. The correct boolean sequence in this case is given by the streams $s_5 = s_2$ delimited by some stream s_4 equal to s'_1 reconstructed by s_2:

$s_4 =$	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**t**
$s_5 =$	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	_	**t**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	_

We can partition (s'_2, s_3) by writing zeroes to the dummies in s_5, not'ing it and flag distribute s'_3 over this stream:

flag distribute:

$s'_5 =$	**t**	**t**	**f**	**t**	**t**	**t**	**f**	**t**	**t**	**t**	**t**	**f**	**t**	**t**	**t**	**t**	**f**	**t**	**t**	**t**
$s'_3 =$	a	b		c	d	e		f	g	h	_		i	j	k	l		m	n	_

yields

$s''_3 =$	a	b	c	c	d	e	f	f	g	h	_	i	i	j	k	l	m	m	n	_

The new delimiter is s_5 with **t**'s for the dummy value:

$s''_5 =$	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**t**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**t**
$s''_3 =$	a	b	_	c	d	e	_	f	g	h	_	_	i	j	k	l	_	m	n	_

Now all that is left is to attach a top-most delimiter. s_4 is a good candidate, but it needs to be converted into reconstructed form. This is done by right-shifting s''_5, or'ing with s_4 and packing s_4 with that (right shifting is necessary to make the **t** arrive at the start of segments instead of the end).

Packing

$s_4 =$	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**f**	**t**

with

$s'''_5 =$	**t**	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**t**	**t**	**t**	**f**	**f**	**f**	**f**	**t**	**f**	**t**

yields

$s'_4 =$	**f**			**f**				**f**			**t**	**f**	**f**					**f**		**t**

which we can put on top of the stack to produce the final result.

$s'_4 =$	**f**			**f**				**f**			**t**	**f**	**f**					**f**		**t**
$s''_5 =$	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**t**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**t**
$s''_3 =$	a	b	_	c	d	e	_	f	g	h	_	_	i	j	k	l	_	m	n	_

If we partition once again with the flag stream that corresponds to the original $s_{\mathbf{t}}$, we should obtain the original result

$s_6 =$	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**f**	**t**
$s_7 =$	**f**	**f**	**t**	**f**	_	**f**	**f**	**t**	**f**	_

We can see that s_6 is the same as the original s_1, so we do not need to flag distribute it. In fact, we just need to replace the top-most segment descriptor with the flag streams, where we have reconstruct the delimiter stream s_5 to s_5':

$s_5' =$	**f**							**f**			**t**	**f**	**f**					**f**		**t**
$s_7 =$	**f**			**f**			**t**	**f**			**f**	**f**	**f**				**t**	**f**		**f**
$s_5'' =$	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**t**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**t**
$s_3'' =$	`a`	`b`	_	`c`	`d`	`e`	_	`f`	`g`	`h`	_	_	`i`	`j`	`k`	`l`	_	`m`	`n`	_

Which is the same as the original streams:

$s_0 =$	**f**							**f**			**t**	**f**						**f**		**t**
$s_1 =$	**f**			**f**			**t**	**f**			**t**	**f**	**f**				**t**	**f**		**t**
$s_2 =$	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**t**	**t**	**f**	**f**	**f**	**f**	**t**	**f**	**f**	**t**
$s_3 =$	`a`	`b`	_	`c`	`d`	`e`	_	`f`	`g`	`h`	_	_	`i`	`j`	`k`	`l`	_	`m`	`n`	_

$$
\begin{aligned}
\mathcal{O}\langle\!\langle \mathbf{part}_\sigma \rangle\!\rangle\ s\ ((s_0,\ st_0),\ (s_1,\ s_2)) = \\
\text{do}\ \ & s_3 \leftarrow \mathit{emit}\ \mathbf{condw}_1^{\wedge}\ s_2\ s_1 \\
& s_4 \leftarrow \mathit{scan}_{>>}\ s_3 \\
& s_5 \leftarrow \mathit{emit}\ \mathbf{or}^{\wedge}\ s_4\ s_1 \\
& s_5 \leftarrow \mathit{emit}\ \mathtt{pack}_{\mathbf{bool}}\ s_2\ s_5 \\
& s_6 \leftarrow \mathit{emit}\ \mathbf{condw}_0^{\wedge}\ s_2\ s_1 \\
& s_7 \leftarrow \mathit{emit}\ \mathbf{not}^{\wedge}\ s_6 \\
& st_1 \leftarrow \mathit{shallowflagdist}_\sigma\ st_0\ s_7 \\
& \eta_{\mathbf{Gen}}\ (s_5,\ (s_3,\ st_1))
\end{aligned}
$$

where

$$\otimes ::= \cdots \mid >>$$

is the right-shift monoid on boolean streams with 1 as identity element. It has the associative operator

$$x + y = y$$

and

$$
\begin{aligned}
\mathit{shallowflagdist}_\pi\ s_0\ s &= \mathit{emit}\ \mathtt{flagdist}_\pi\ s_0\ s \\
\mathit{shallowflagdist}_{[\tau]}\ (st_0,\ st_1)\ s &= \text{do}\ \ st_0' \leftarrow \mathit{shallowflagdist}_{\mathbf{int*int}}\ st_0\ s \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (st_0',\ st_1) \\
\mathit{shallowflagdist}_{\{\sigma\}}\ st_0\ s &= \eta_{\mathbf{Gen}}\ st_0 \\
\mathit{shallowflagdist}_{\mathbf{1}}\ ()\ s &= \eta_{\mathbf{Gen}}\ () \\
\mathit{shallowflagdist}_{\sigma_0 * \sigma_1}\ (st_0,\ st_1)\ s &= \text{do}\ \ st_0' \leftarrow \mathit{shallowflagdist}_{\sigma_0}\ st_0\ s \\
&\qquad\ \ st_1' \leftarrow \mathit{shallowflagdist}_{\sigma_1}\ st_1\ s \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (st_0',\ st_1')
\end{aligned}
$$

List constructor

$$\mathcal{O}\langle\!\langle \mathbf{list}_{k,\tau} \rangle\!\rangle\ s\ (st_1,\ldots,st_k) = \text{do}\ \begin{array}{l} s_0 \leftarrow \mathit{const}\ s\ k \\ s_0' \leftarrow \mathit{emit}\ \mathbf{condw}_0\texttt{\^{}}\ s_0 \\ s_1 \leftarrow \mathit{scan}_{\mathbf{sum}}\ s_0' \\ s_2 \leftarrow \mathit{emit}\ \texttt{2flags}\ s_0' \\ s_3 \leftarrow \mathit{flagiota}\ s\ s_2 \\ st' \leftarrow \mathit{list}_{k,\tau}\ s_3\ (st_1,\ldots,st_k) \\ \eta_{\mathbf{Gen}}\ ((s_1,\ s_0),\ st') \end{array}$$

$$\begin{array}{l} \mathit{normalize}_\tau\ s\ ((s_0,\ s_1),\ st) = \\ \quad \text{do}\ \begin{array}{l} s_2 \leftarrow \mathit{emit}\ \mathbf{condw}_0\texttt{\^{}}\ s_1\ s \\ s_3 \leftarrow \mathit{emit}\ \texttt{2flags}\ s_2 \\ s_4 \leftarrow \mathit{emit}\ \texttt{flagdist}_{\mathbf{int}}\ s_0\ s_3 \\ s_5 \leftarrow \mathit{flagiota}\ s_3 \\ s_6 \leftarrow \mathit{emit}\ (\texttt{+\^{}})\ s_4\ s_5 \\ st' \leftarrow \mathit{gather}_\tau\ st\ s_6 \\ s_7 \leftarrow \mathit{scan}_{\mathbf{sum}}\ s_2 \\ \eta_{\mathbf{Gen}}\ ((s_7,\ s_2),\ st') \end{array} \end{array}$$

$$\begin{array}{rcl} \mathit{list}_{k,\mathbf{1}}\ s\ ((),\ldots,()) & = & \eta_{\mathbf{Gen}}\ () \\ \mathit{list}_{k,\pi}\ s\ (s_1,\ldots,s_k) & = & \text{do}\ \begin{array}{l} s' \leftarrow \mathit{emit}\ \texttt{merge}_{k,\pi}\ s\ (s_1,\ldots,s_k) \\ \mathit{emit}\ \texttt{flush}_\pi\ s' \end{array} \end{array}$$

$$\begin{array}{l} \mathit{list}_{k,\tau_0 * \tau_1}\ s\ ((st_1,\ st_1'),\ldots,(st_k,\ st_k')) = \\ \quad \text{do}\ \begin{array}{l} st \leftarrow \mathit{list}_{k,\tau_0}\ s\ (st_1,\ldots,st_k) \\ st' \leftarrow \mathit{list}_{k,\tau_1}\ s\ (st_1',\ldots,st_k') \\ \eta_{\mathbf{Gen}}\ (st,\ st') \end{array} \end{array}$$

Append Appending two sequences in the high level language corresponds to interleaving segments between two stream stacks in the target language. We use the transformer interleave for his purpose

$$\begin{array}{l} \mathcal{O}\langle\!\langle \mathbf{append}_\sigma \rangle\!\rangle\ s\ ((s_0,\ st_0),\ (s_1,\ st_1)) = \\ \quad \text{do}\ \begin{array}{l} s_2 \leftarrow \mathit{emit}\ \texttt{interleave}\ s_0\ s_1 \\ s_3 \leftarrow \mathit{emit}\ \texttt{scan}_{\mathbf{xor}}\ s_2 \\ s_4 \leftarrow \mathit{emit}\ \mathbf{b2i}\texttt{\^{}}\ s_3 \\ st_2 \leftarrow \mathit{deepmerge}_\sigma\ st_0\ st_1\ s_4 \\ \eta_{\mathbf{Gen}}\ (s_2,\ st_2) \end{array} \end{array}$$

$$
\begin{aligned}
\mathit{deepmerge}_{\pi}\ s_0\ s_1\ s &= \mathit{emit}\ \mathtt{merge}_{\pi}\ s\ s_0\ s_1 \\
\mathit{deepmerge}_{[\tau]}\ (st_0,\ st_1)\ (st_2,\ st_3)\ s &= \text{do}\ \ st'_0 \leftarrow \mathit{deepmerge}_{\mathbf{int*int}}\ st_0\ st_2\ s \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (st'_0,\ (st_1,\ st_2)) \\
\mathit{deepmerge}_{\{\sigma\}}\ (s_0,\ st_1)\ s &= \text{do}\ \ st'_1 \leftarrow \mathit{deepmerge}_{\sigma}\ st_1\ s \\
&\qquad\ \ s_1 \leftarrow \mathit{deepmerge}_{\mathbf{bool}}\ s_0\ s \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (s_1,\ st'_1) \\
\mathit{deepmerge}_{\mathbf{1}}\ ()\ ()\ s &= \eta_{\mathbf{Gen}}\ () \\
\mathit{deepmerge}_{\sigma_0*\sigma_1}\ (st_0,\ st_1)\ (st_2,\ st_3)\ s &= \text{do}\ \ st'_0 \leftarrow \mathit{deepmerge}_{\sigma_0}\ st_0\ st_2\ s \\
&\qquad\ \ st'_1 \leftarrow \mathit{deepmerge}_{\sigma_1}\ st_1\ st_3\ s \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (st'_0,\ st'_1)
\end{aligned}
$$

where

$$\otimes ::= \cdots \mid \mathbf{xor}$$

is the xor monoid on boolean streams with 1 as identity element.

Tabulate and stream Tabulation is done by turning a delimiter stream into a length stream using the *2lens* function. The starts are calculated as as scan of the lengths and the result is a virtually segmented list. We have to insert dummy values in both the starts and the lengths to maintain our choice of representation:

$$
\begin{aligned}
\mathcal{O}\langle\!\langle \mathtt{tabulate}_{\tau} \rangle\!\rangle\ s\ (s_0, st_0,\ =)\text{do}\ \ & s_1 \leftarrow \mathit{2lens}\ s_0 \\
& s_2 \leftarrow \mathit{emit}\ \mathbf{not\hat{}}\ s \\
& s_3 \leftarrow \mathit{emit}\ \mathbf{b2i\hat{}}\ s_2 \\
& s_4 \leftarrow \mathit{emit}\ \mathtt{flagdist}_{\mathbf{int}}\ s_1\ s_3 \\
& s_5 \leftarrow \mathit{flagscan}_{\mathbf{sum}}\ s_4\ s \\
& st_1 \leftarrow \mathit{flush}_{\tau} \\
& \eta_{\mathbf{Gen}}\ ((s_1,\ s_2),\ st_1)
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{flush}_{\mathbf{1}}\ () &= \eta_{\mathbf{Gen}}\ () \\
\mathit{flush}_{\pi}\ s &= \mathtt{flush}_{\pi}\ s \\
\mathit{flush}_{\tau_0*\tau_1}\ (st_0,\ st_1) &= \text{do}\ \ st_2 \leftarrow \mathit{flush}_{\tau_0}\ st_0 \\
&\qquad\ \ st_3 \leftarrow \mathit{flush}_{\tau_1}\ st_1 \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (st_2,\ st_3) \\
\mathit{flush}_{[\tau_0]}\ (st_0,\ st_1) &= \text{do}\ \ st_2 \leftarrow \mathit{flush}_{\mathbf{int*int}}\ st_0 \\
&\qquad\ \ \eta_{\mathbf{Gen}}\ (st_2,\ st_1)
\end{aligned}
$$

Streaming a tabulated vector is done by the gather function

$$
\begin{aligned}
& \mathit{stream}_\tau\ s\ \texttt{((}s_0\texttt{, }s_1\texttt{), }st\texttt{) =} \\
& \quad \text{do}\ \ s_2 \leftarrow \mathit{emit}\ \mathbf{condw}_0\texttt{\^{}}\ s_1\ s \\
& \qquad\ \ \ s_3 \leftarrow \mathit{emit}\ \texttt{2flags}\ s_2 \\
& \qquad\ \ \ s_4 \leftarrow \mathit{emit}\ \texttt{flagdist}_{\mathbf{int}}\ s_0\ s_3 \\
& \qquad\ \ \ s_5 \leftarrow \mathit{flagiota}\ s_3 \\
& \qquad\ \ \ s_6 \leftarrow \mathit{emit}\ \texttt{(+\^{})}\ s_4\ s_5 \\
& \qquad\ \ \ st' \leftarrow \mathit{gather}_\tau\ st\ s_6 \\
& \qquad\ \ \ \eta_{\mathbf{Gen}}\ \texttt{(}s_3\texttt{, }st'\texttt{)}
\end{aligned}
$$

5.4 Value and cost preservation

This concludes the description of the transformation from source language to target language. We have provided a transformation for *any* well-typed source language expression using an update monad to generate a list of stream definitions. We have also described how we can go back, by providing an interpretation of the output streams of a streaming system as a high-level value. We conclude this section with a conjecture, that seems plausible based on intuition and a few examples.

Conjecture *(Value and cost preservation)*
$\forall e \forall B > 0 \exists K > 0$ such that,
if $[] \vdash e : \sigma$ and $[] \vdash e \downarrow \langle v, \langle W, D, M, N \rangle \rangle$ and e *can be streamed*[3] then

(i) $\langle\!\langle e \rangle\!\rangle = \texttt{let}\ \Phi\ \texttt{in}\ st$

(ii) $\vdash \Phi : \Pi$

(iii) $\exists ss \in \mathbf{SId}^*$ such that

$$\Phi \vdash_B \langle \Sigma_{init}(\Pi), A_{init}(\Phi), \chi_{init}(\Phi, st) \rangle \rightarrow_{ss} s^* \langle \Sigma', A', \chi' \rangle \ !\ \langle W', D', M' \rangle$$

(iv) $\left\langle \Sigma'(st) \right\rangle_\sigma = v$

(v) $D' \log B \leq K * (W/B + D \log B)$ and $M' \leq K * \min(M, B)$

[3]Can be streamed means that it will not get stuck. The set of expressions that can be streamed is a non-trivial subset of all expression. An example of an expression that cannot be streamed is an expression that adds the length of a sequence to all its elements. If an expression can be streamed or not is supposedly discovered by appropriate static analysis, outside the scope of this thesis.

6 Implementation

The proposed languages and transformation have been implemented. The implementation is divided in four major parts:

- A front end with a parser and type checker for the source language.
- A runtime system for the target language.
- A compiler from source to target language.
- Primitive stream transformers written in a back-end data parallel language.

The output evaluating an expression in the implementation is the computed value, as well as the work, step and space required as defined by the cost model. We do not an measure the actual space and time used, and we don not expect the implementation to run efficiently since it is merely a proof of concept.

Frontend The front end provides an interface to the programmer. We have designed the front end as an interactive prompt embedded in Haskell, where the programmer may write source language expressions directly or load an expression from a source file. The expressions are parsed, desugared, compiled to the target language and executed in the runtime system of the target language. The programmer can control the parameter B, the block size, from the front end for experimental purposes, though in a more realistic implementation, the block size should be solely determined by the runtime system of the target language.

6.1 Runtime system

For simplicity, the runtime system of the target language has been written in Haskell as well. The runtime system implements a simple scheduler that selects the next transformer to fire by the first stream definition that has at least B elements in all it's input buffers and there is space to write at least B elements to it's output buffer. This scheduler is close to naive, since it potentially scans the entire list of definitions to find a transformer to fire. Since we consider the size of expressions as constants, this does not lead to asymptotically wrong performance, but it is certainly noticeable. A more efficient scheduler would manage a set or priority queue of definitions that a known to be fireable, so that the next definition to fire can be selected in almost constant time with regard to the size of the expression list.

The runtime system has a debug mode with an interactive prompt where execution pauses on each transition and dumps the content of all buffers to the prompt. Work, steps and space as defined in the cost model of the target language is computed along with execution.

6.2 Compiler

The transformation from source language to target language is implemented in Haskell as described in the transformation section using a generator monad.

6.3 Back end

The primitive data parallel transformers are also implemented in Haskell. This implies that they are *not* efficient parallel implementations as they should be, if the complexity preservation conjecture have any chance of being correct, except for a block size of 1.

In principle, there should not be any major difficulties in replacing them with foreign function calls to pre-compiled kernels, written in CUDA for instance.

6.4 Evaluation

The implementation has been tested by running a number of examples and checking if they compute the correct value, and if the bounded buffers respects their bounds. The examples includes matrix-matrix multiplication, which is a good example, since it contains segmented reductions and three levels of nested apply-to-each constructs. We have also positively tested more irregular examples.

We evaluate the implementation by implementing and running the iota-square-sum example

```
sum({ x^2 : x in &l})
```

We know asymptotically how much work, steps and space it should require during execution, namely

$$W = O(l)$$

$$D = O(l/P)$$

$$M = O(\min(l, P))$$

so by running the example with different values of B and l we get the following complexities :

Work W

$l \setminus B$	1	10	10^2	10^3
10	94	90	89	89
10^2	859	859	810	809
10^3	8509	8509	8509	8010

Steps D

$l \setminus B$	1	10	10^2	10^3
10	84	17	9	9
10^2	759	84	17	9
10^3	7509	759	84	17

Space M

$l \setminus B$	1	10	10^2	10^3
10	17	143	1403	14003
10^2	17	143	1403	14003
10^3	17	143	1403	14003

We can see that the work grows linearly with l, the number of steps grows linearly in l and linearly in B^{-1}, and the space grows linearly in B:

$$W = O(l)$$

$$D = O(l/B)$$

$$M = O(B)$$

For the sake of comparison assume that $B = O(P)$ for some imaginary machine with parallel degree P. We see that our implementation almost computes the correct complexities. The work and steps are identical, but the space is slightly worse, namely $O(P)$ instead of $O(\min(l, P))$. The difference is subtle, and it arises because our implementation allocates buffers entirely, when only a portion is needed. All in all we conclude that the implementation behaves as expected with some minor issues.

7 Conclusion

While we have not been able to demonstrate without a doubt, that high-level platform-independent can indeed be as efficient as equivalent low-level languages, the thesis certainly sheds some light on the subject, and it indicates the possibility of a positive answer. We have taken a traditional NDP point of view, identified a small but crucial example program with worse than desired space complexity and searched for possible solutions. We have argued that some form of machine-dependent and problem-size specific sequentialization of parallelism is a necessity in order to achieve the desired performance, and to achieve this in a high-level machine-independent language, we have necessarily turned our attention to a target execution environment.

We have demonstrated that an execution model that combines streaming with dynamic scheduling on a host processor and flat data parallelism on a homogeneous collection of device processors has the potential to solve the problem. We have not however, provided any formal guarantees as we had hoped, but we have formalized a framework for doing so - by presenting the languages using formal syntax, types and semantics and augmenting the semantics with cost models. We have designed the cost models to be ideal in the sense that they give the performance one would expect from an equivalent low-level hand-written implementation.

The contributions of the thesis that can be regarded as novel are:

1. A description and proof-of-concept implementation of a runtime environment that combines streaming with flat data parallelism, and has an intuitive cost model of work, steps and time.

2. A high-level concept of sequences that reflects the behavior of streamed lists.

3. A space cost model in an NDP language with sequences given in two extreme cases analogous to work and steps, with a formula for deriving expected space cost on a concrete machine.

4. A vectorizing transformation in this context, that presumable preserves value and complexity semantics.

The concept of sequences and the space cost model on the source language are useful results in themselves, as they formally capture where traditional NDP languages go wrong. With these tools, it should now be possible to design a provably space-efficient language, which is a big step towards answering the question if high-level machine-independent languages can ever be as efficient as low-level languages.

The target language and transformation demonstrates an example of how such a provably space-efficient language could be designed. They are not hard results as such, since we have not given a theorem with a proof, but instead we have given a proof-of-concept implementation as well as a conjecture that we believe is plausible, implicitly delaying a potential proof to future work.

References

[1] "NVIDIA GPU Computing Documentation." http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[2] "NVIDIA GPU Computing Documentation." http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[3] L. Bergstrom and J. Reppy, "Nested data-parallelism on the GPU," Sept. 2012.

[4] M. Chakravarty, G. Keller, S. Lee, T. McDonell, and V. Grover, "Accelerating Haskell array codes with multicore GPUs," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, (New York, NY, USA), pp. 3–14, ACM, 2011.

[5] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, pp. 85–97, March 1996.

[6] G. E. Blelloch and G. W. Sabot, "Compiling collection-oriented languages onto massively parallel computers," *J. Parallel Distrib. Comput.*, vol. 8, pp. 119–134, February 1990.

[7] G. E. Blelloch, *Vector models for data-parallel computing.* Cambridge, MA, USA: MIT Press, 1990.

[8] G. Blelloch, "NESL: A nested data-parallel language (3.1)," tech. rep., 1995. CMU-CS-95-170.

[9] J. F. Prins and D. W. Palmer, "Transforming high-level data-parallel programs into vector operations," in *Proceedings Of The Fourth ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, (New York, N. Y., USA), pp. 119–128, ACM, 1993.

[10] J. W. Riely, J. Prins, and S. P. Iyer, "Provably correct vectorization of nested-parallel programs," in *Proceedings of the Conference on Programming Models for Massively Parallel Computers*, PMMP '95, (Washington, DC, USA), pp. 213–222, IEEE Computer Society, 1995.

[11] D. W. Palmer, J. F. Prins, and S. Westfold, "Work-efficient nested data-parallelism," in *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, FRONTIERS '95, (Washington, DC, USA), pp. 186–, IEEE Computer Society, 1995.

[12] G. Keller and M. Simons, "A calculational approach to flattening nested data parallelism in functional languages," in *Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*, ASIAN '96, (London, UK), pp. 234–243, Springer-Verlag, 1996.

[13] S. Peyton Jones, "Harnessing the multicores: Nested data parallelism in Haskell," in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, (Berlin, Heidelberg), pp. 138–150, Springer-Verlag, 2008.

[14] M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow, "Data parallel Haskell: A status report," in *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, (New York, NY, USA), pp. 10–18, ACM, 2007.

[15] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pp. 1–12, ACM, 1995.

[16] F. M. Madsen, "Flattening nested data parallelism." Master's project, February 2012.

[17] G. Keller, M. M. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones, "Vectorisation avoidance," in *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, (New York, NY, USA), pp. 37–48, ACM, 2012.

[18] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons, *Space profiling for parallel functional programs*, vol. 43. ACM, 2008.

[19] G. E. Blelloch, *Vector models for data-parallel computing*, vol. 75. MIT press Cambridge, MA, 1990.

[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware: Proceedings of the 22 nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, vol. 4, pp. 97–106, 2007.