Runtime Array Fusion for Data Parallelism

George Roldugin

UNSW
AUSTRALIA

School of Computer Science and Engineering
University of New South Wales

March 2015

A thesis in fulfilment of the requirements for the degree of

Master of Engineering

School of Computer Science and Engineering
The University of New South Wales
Australia

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Roldugin

First name: Georgy Other name/s:

Abbreviation for degree as given in the University calendar: Master of Engineering

School: Computer Science and Engineering Faculty: Engineering

Title: Runtime Array Fusion for Data Parallelism

Abstract 350 words maximum: (PLEASE TYPE)

The benefits of high level approach to parallel programming are well understood and are often desired in order to separate the domain view of the problem from the intricate implementation details. Yet, a naive execution of the resulting programs attracts unnecessary and even prohibitive performance costs.

One convenient way of expressing a program is by composing collective operations on large data structures. Even if these collective operations are implemented efficiently and provide a high degree of parallelism, the result of each operation must be fully computed and written into memory before the next operation can consume it as input. The cost of transferring these intermediate results to and from memory has a very noticeable impact on the performance of the algorithm and becomes a serious drawback of this high level approach.

Program optimisation which attempts to detect and eliminate the creation of intermediate results by combining multiple operations into one is known as fusion. While it is a well studied problem, there are unfilled gaps when it comes to fusing data parallel programs. In particular, I demonstrate solutions to the problems of fusion with multiple consumers as well as producing multiple results from one fused computation (tupling).

Through my research, I have designed and implemented an embedded domain specific language called LiveFusion that offers fusible combinators operating on flat and segmented arrays. To achieve fusion I propose a generic loop representation and use the concept of rates to guide fusion.

The results show that LiveFusion is considerably more effective at exploiting opportunities for fusion than previous systems. Specifically, the average performance increase of 3.2 for a non-trivial program indicates the attractiveness of the approach.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

Signature  Witness  Date 31.03.15

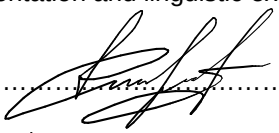
The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed 

Date March 31, 2015

Contents

List of Figures

Abstract

The benefits of high level approach to parallel programming are well understood and are often desired in order to separate the *domain view* of the problem from the intricate implementation details. Yet, a naive execution of the resulting programs attracts unnecessary and even prohibitive performance costs.

One convenient way of expressing a program is by composing collective operations on large data structures. Even if these collective operations are implemented efficiently and provide a high degree of parallelism, the result of each operation must be fully computed and written into memory before the next operation can consume it as input. The cost of transferring these intermediate results to and from memory has a very noticeable impact on the performance of the algorithm and becomes a serious drawback of this high level approach.

Program optimisation which attempts to detect and eliminate the creation of intermediate results by combining multiple operations into one is known as *fusion*. While it is a well studied problem, there are unfilled gaps when it comes to fusing data parallel programs. In particular, I demonstrate solutions to the problems of fusion with multiple consumers as well as producing multiple results from one fused computation (tupling).

Through my research, I have designed and implemented an embedded domain specific language called *LiveFusion* that offers fusible combinators operating on *flat* and *segmented* arrays. To achieve fusion I propose a generic loop representation and use the concept of *rates* to guide fusion.

The results show that *LiveFusion* is considerably more effective at exploiting opportunities for fusion than previous systems. Specifically, the average performance increase of 3.2 for a non-trivial program indicates the attractiveness of the approach.

Acknowledgements

I wish to express gratitude to my supervisors Manuel and Gabi for their guidance in the times of confusion and exploration of unknown territories.

I am glad to have been part of PLS group who strive to do great work at the intersection of engineering and science. I especially thank Ben Lippmeier for numerous impromptu lessons that were invaluable at the start of my degree.

I would also like to thank Prof. Ricardo Peña for his guidance during my stay at Universidad Complutense de Madrid.

I am grateful to my family and friends who offered their support throughout my degree.

I am indebted to the literary works of Stephen Covey and Brian Tracey among others, who motivated me to develop myself time and again.

Finally, this work would not have been possible without Nataly, her heroic patience, encouragement and love.

Chapter 1

Introduction

The past decade has seen a rise in development of increasingly sophisticated multi-core and multi-processor computer systems. From the early hyperthreaded solutions to the modern architectures comprising a number of independent processing cores, the horsepower for demanding applications is now available even in the most affordable consumer grade systems. To offer a large amount of parallelism, the high grade systems may have multiple multi-core processors, where each core can run a number of hardware scheduled threads.

While the hardware for running highly parallel computations is readily available and is improving at a high pace, the development of applications that make use of this capability is often a challenging task. The problems involved are finding an appropriate parallel implementation of the task at hand, implementing it so that the computations are evenly distributed across processing elements and synchronising parallel computations when necessary. The latter two require a substantial programmer's intervention to get the algorithm running fast.

The resulting application is a mixture of the actual algorithm and the implementation specific code, dealing with concurrency and parallelism. This obscures code clarity and may generally be error-prone.

One alternative to this practice of explicit parallelisation is to provide a common set of collective operations on large data structures. Programs implemented in terms of these operations would be automatically parallelised across available processing elements. This approach is successfully exercised by a multitude of frameworks covering many host languages, target architectures and suitable application domains [42, 25, 11, 1].

However, in the pursuit of a high level approach to programming a major inefficiency

is introduced. Having provided a number of collective operations, the problem of multiple traversals arises. In each program there is likely to be a number of such operations composed together in some way to compute the desired result. With each collective operation potentially traversing a large data structure, the memory traffic is considerably increased.

In a program written by hand without the use of collective operations the programmer would naturally recognise all the operation that could be performed in one pass over the data structure. The program would only traverse the data structure as few times as required, keeping the memory traffic to the necessary minimum and utilising cache correctly.

On the other hand, using a straight-forward implementation of a high level library of collective operations would result in code that traverses data many times while potentially performing only a small change in each pass. For large data structures this may lead to poor cache utilisation and high memory traffic, noticeably reducing the overall performance.

Additionally, each operation may need to store its result in a new data structure, which is especially true for languages where values are immutable by default. Allocating temporary data structures to store the intermediate results leads to further memory and runtime penalties.

Optimising out the superfluous data structure traversals and allocations is collectively referred to as **Loop Fusion**. It allows a program expressed in terms of high level operations, also called **combinators**, to be transformed into a program that would be comparable to a handwritten one in operation and speed.

In this thesis I explore the problem of loop fusion in the context of purely functional data parallel programs. In particular I explore the challenges of fusing programs written for the *nested data parallel* frameworks such as *Data Parallel Haskell (DPH)*.

In the following I summarise the areas of my contribution, explicitly noting which are my individual work:

- We propose a novel system, *LiveFusion*, for fusing collective array operations at the run-time of the program in a purely functional setting (Chapter 4).

- I implement the proposed system as a embedded domain-specific language for the *Haskell* programming language which includes automatic recovery of shared terms

(Section 4.3.1), explicit tupling of results and a scalar language (Section 4.4).

- I devise a generic representation of loops as a more structured and composable alternative to `while` and `for` loops found in procedural languages. I implement the new looping construct as an intermediate language, *Loop*, inside *LiveFusion* (Chapter 5).

- I present the mapping of **flat** (Sections 5.3, 5.5) and **segmented** (Section 5.7)[1] array combinators onto the generic loop representation designed to enable fusion in the *Data Parallel Haskell* framework (Chapter 3).

- I implement a dynamic code generator which translates fused loops expressed in the intermediate *Loop* language into *Haskell* source code. I also implement dynamic code compilation and loading facilities enabling highly efficient execution of the generated code (Chapter 6).

- I present performance benchmarks evaluating *LiveFusion* against the types of programs resulting from the use of *nested data parallel* frameworks such as *Data Parallel Haskell* (Chapter 7).

The source code for *LiveFusion* is available at `https://github.com/roldugin/LiveFusion`. The source code for *Data Parallel Haskell* is available at `https://github.com/ghc/packages-dph`.

I shall begin by introducing the reader to the problem of fusion and the context of my work. In the next chapter I will outline the challenges of array fusion finding a solution to which has motivated my work in the prior years.

[1] The term **segmented** describes combinators that operate on nested arrays. It owes its name to the way such arrays are represented and is discussed in Section 3.2.1. **Flat** combinators operate on regular arrays.

Chapter 2

Array fusion

Suppose we have the following computation to perform:

```
sum (zipWith (*) xs ys)
```

A person familiar with the fundamentals of functional programming is likely to spot the computation of the dot product of two vectors in this snippet of *Haskell* code. Indeed, the `zipWith` list combinator will element-wise multiply the two vectors `xs` and `ys`. Quite expectedly the `sum` combinator will sum the elements of the resulting vector into a scalar value yielding the dot product of `xs` and `ys`.

This one-liner was an attempt to develop the motivation for the high-level view on numeric computations. It may seem reasonable to replace the lists with arrays and re-implement the same high-level interface in terms of traditional arrays to avoid random memory access penalties as in the case of lists.

Providing an instantly familiar interface without compromising performance has been one of the goals of the *Data Parallel Haskell (DPH)* project [42, 9] discussed in more detail in Section 3.1. The work described in this essay has been carried out in the context of this project.

However, even if we replaced the lists with arrays and gave efficient implementations to `sum` and `zipWith`, the resulting algorithm is still likely to be slower than one written by hand in a language such as *C*. The `zipWith` combinator *produces* an *intermediate array* containing the element-wise product of the two input vectors. It is immediately *consumed* by `sum`, yielding the final (scalar) value. Thus the algorithm performs two traversals and allocates another array of the size of the input arrays.

To illustrate the benefit of Array Fusion let us turn to the implementation of same algorithm expressed in *C*:

```
double dotProduct (double xs[], double ys[], int len) {
    // zipWith
    double* temp = malloc(len * sizeof(double));
    for(int i = 0; i < len; i++)
        temp[i] = xs[i] * ys[i];

    // sum
    double result = 0;
    for(int i = 0; i < len; i++)
        result += temp[i];

    return result;
}
```

We immediately notice that both loops iterate over the same *range* of indices and we could hence perform it as one loop. The two loops are said to have the same *rate* (the term used more recently in the context of loop fusion in *Haskell* [31]).

```
double* temp = malloc(len * sizeof(double));
double result = 0;
for(i = 0; i < len; i++) {
    temp[i] = xs[i] * ys[i];
    result += temp[i];
}
```

The process of finding and exploiting the opportunities for merging multiple loops into one is referred to as **loop fusion**.

However, this does not completely bypass the allocation of an intermediate array. Clearly, the intermediate array is redundant and the intermediate value `temp` could just be a *scalar* as in the following:

```
double temp; // temp has become scalar
double result = 0;
for(int i = 0; i < len; i++) {
    temp = xs[i] * ys[i];
    result += temp;
}
```

The optimisation that removes the need for temporary arrays by replacing them with scalars is called **scalarisation**.

It is a special case of **array contraction**. Array contraction optimisation attempts to remove a *dimension* from the array. In this particular case we contracted a one dimensional array to a scalar, hence *scalarisation*. However, in the examples with arrays of higher dimensions the dimension could just be reduced, and not eliminated entirely. We will see examples of that in the upcoming chapters when we discuss *segmented array combinators*. This concept is taken even further in multi-dimensional array systems such as *Repa* [25] and *Accelerate* [11].

Loop Fusion and **Array Contraction** optimisation are collectively referred to as **Array Fusion**.

Another term for this commonly encountered in literature is **deforestation**, first coined by Philip Wadler in [54].

In the remainder of this chapter I outline the reasons which led me to the line of work described in this thesis.

2.1 The problem statement

In this section I identify several types of fusion as well as the shortcomings of the previously available fusion systems which motivated the search for a fresh approach.

I will make many references to the *Stream Fusion* ([14], [15], [33]) framework in my discussion as one of the most comprehensive fusion frameworks that exist for *Haskell*. However, I also will implicitly or explicitly make references to *foldr/build* [17], *Functional Array Fusion* [10] since they all fall into the category of **equational fusion frameworks**. Such fusion systems rely on adjacent term rewriting (for example through the use of rewrite rules [41]) and subsequent compiler optimisations to produce efficient code.

Stream Fusion and *foldr/build* fusion are covered more thoroughly in Appendix A.

2.1.1 Simple pipeline fusion

The most basic form of optimisation that is attempted by most fusion systems is `map/map` fusion where a pipeline of `map` combinators is transformed into a single `map`:

$$\texttt{map g} \circ \texttt{map f} \mapsto \texttt{map (g} \circ \texttt{f)}$$

However, `map` is not the only combinator that can be chained together in a pipeline. `scan` and `filter`, like `map`, both *consume* one array and *produce* another.

In the simplest case, at the beginning of such a combinator pipeline there is a physical, or **manifest** array, materialised in memory. However, there may also be a **pure producer** or a **generator** instead. As such `replicate`, `enumerate` and several others, generate an array from scalars or functions according to some rule.

Just like a pipeline does not necessarily start with a physical array, it may be concluded with a computation such as `fold` yielding one scalar value.

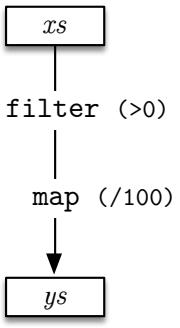
The combinators discussed so far can be combined into a pipeline of operations where an array output of one combinator is fed directly as input to the next. Additionally, each combinator in the pipeline is able to consume the elements one by one at the *rate* they are produced by the preceding combinator.

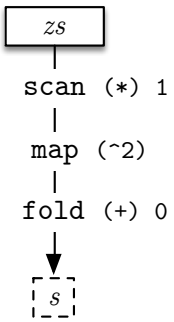
Such simple pipelines are generally handled well by most fusion frameworks. In particular the expressions presented in Figure 2.1 are all fusible by *Stream Fusion* [14] (and Appendix A.2) and *Functional Array Fusion* [10].

```
let ys = map (/100)
       $ filter (>0)
       $ xs
```

```
let s = fold (+) 0
      $ map (^2)
      $ scan (*) 1
      $ zs
```

```
let ws = filter odd
       $ map (+1)
       $ enumFromTo 1 10
```





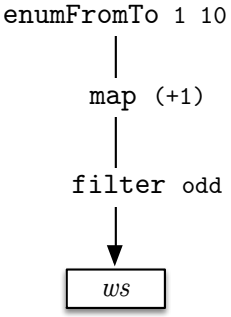


Figure 2.1: Examples of simple combinator pipelines: *Haskell* expressions (top) and their corresponding data flow graphs (bottom).

2.1.2 Fusing into multiple consumers

In the previous section we have looked at *uninterrupted* pipelines of combinators. Each intermediate array produced by a combinator was immediately consumed by *exactly one* other combinator.

However, in some cases the output of a combinator may be consumed at two or more places in the program. Consider the example on Figure 2.2, showing a program computing, for each element, the ratio of the sum of all preceding elements to the current element. Because computing the ratio involves division operator, the input array is first filtered to exclude any zero elements.

The output of the `filter` combinator is consumed by both the `scan` to compute the partial sums and the `zipWith` to give the final ratio. If one was to write this in *C*, the resulting program could be expressed in a single loop.

Nevertheless, this program will not be fused by an equational fusion system. Even though each pair of combinators could be fused individually (`zipWith`/`scan`, `zipWith`/`filter`, `scan`/`filter`) in this program the fusion will stop already after `filter`.

The reason for this is the following. If fused by *Stream Fusion* (the approach for other fusion frameworks is similar), the program will be first expanded to the following[1]:

[1]See Appendix A.2 for the explanation of this expansion.

```
let nz     = filter (≠ 0) xs
    psums  = scan (+) 0 nz
    ratios = zipWith (/) psums nz
in  ratios
```

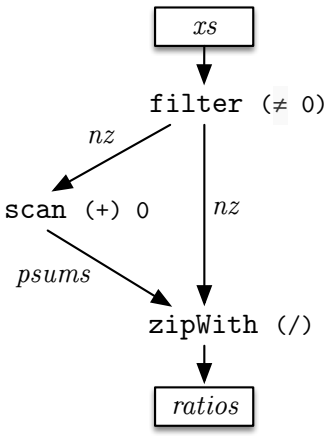


Figure 2.2: `ratios` program illustrating the problem of multiple consumers (left) and the corresponding data flow graph (right).

```
let nz     = unstream $ filterS (≠ 0) $ stream xs
    psums  = unstream $ scanS (+) 0    $ stream nz
    ratios = unstream $ zipWithS (/) (stream psums) (stream nz)
in  ratios
```

For fusion to take place the following rewrite rule needs to apply:

$\langle stream/unstream\ fusion \rangle \forall stream\,(unstream\,s) \mapsto s$

Unfortunately, because `nz` term is used in multiple places, *GHC* will not inline its definition into its use sites. This is done to prevent work duplication. This crucial step will prevent the fusion from occurring.

In the application of fusion to *DPH* we have discovered that this pattern of array consumption is recurring and if one of the principle factors contributing to reduced performance as far as fusion is concerned.

In particular the `filterMax` program on Figure 2.3 is a simplified excerpt of the *DPH* program computing convex hull of a set of points using *QuickHull* algorithm (thoroughly covered in Chapter 7). Given an array of `points` it first computes the distance from each point to some line (not shown) by mapping `dist` function. Then it finds all distances corresponding to points `above` the line as well as the `farthest` distance.

Again, in *C* this whole computation could be expressed by one loop, traversing the `points` array exactly once. However, `distances` array is shared between `filter` and `fold1` consumers, preventing them from being fused by equational fusion frameworks into a single computation. In this example three array traversals would be performed instead of one, increasing memory traffic and memory footprint of the program.

```
let dist (x,y) = ...
    distances  = map dist points
    above      = filter (> 0) distances
    farthest   = fold1 max distances
in  (above, farthest)
```

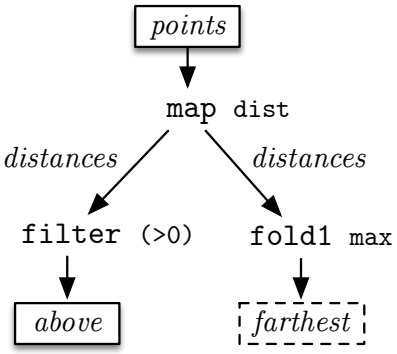


Figure 2.3: `filterMax` program illustrating the problem of multiple consumers in DPH (left) and the corresponding data flow graph (right).

The inability to fuse branched data flow graphs where a single intermediate array is consumed by multiple combinators has proven to be critical to the application of fusion in *DPH* and motivated the search for alternative approaches satisfying this requirement.

In the following section I discuss an additional problem with *Stream Fusion* which introduces runtime overhead in successfully fused programs.

2.1.3 Duplicated counters

In the beginning of this chapter the `zipWith` combinator was used for point-wise multiplication of two vectors. Later it was used to compute point-wise ratios of two arrays (Figure 2.2).

While *Stream Fusion* is able to fuse `zipWith`N combinators where the input arrays are produced by *independent*[2] pipelines of combinators the generated code is not optimal.

For example, consider the following expression:

```
zipWith3 (λ x y z → (x + y) * z) xs ys zs
```

Assuming that `xs`, `ys` and `zs` are physical arrays in memory, *Stream Fusion* results in the following *GHC Core* code (adapted from [31]):

[2] As discussed in the previous section (2.1.2).

```
1  loop = λ i j k l s →
2    case ≥# i len_xs of
3      True  → (# s, I# n #)
4      False →
5        case indexIntArray# xs i of x →
6          case ≥# j len_ys of
7            True  → (# s, I# n# #)
8            False →
9              case indexIntArray# ys j of y →
10               case ≥# k len_zs of
11                 True  → (# s, I# n# #)
12                 False →
13                   case indexIntArray# zs k of z →
14                     loop (+# i 1) (+# j 1) (+# k 1) (+# l 1)
15                       (writeIntArray# rs l (*# (+# x y) z) s)
```

Even though the arrays are consumed in lockstep, *Stream Fusion* introduces a separate counter for each input array (`i`, `j` and `k`) as well as the output index variable `l`. *Stream Fusion* is unable to replace all four with one counter based on the fact that `i = j = k = l`. Not only this introduces extra bounds checks on lines 2, 6 and 10 and increments on line 14, but also increases register pressure. At present *GHC* compiler is unable to identify the invariant `i = j = k = l` and remove the redundant variables either.

In *DPH* it is not uncommon to to encounter point-wise processing of a much larger number of arrays. For example, the *QuickHull* example mentioned in the previous section and discussed more thoroughly in Chapter 7, processes as many as 6 arrays using the `zipWith6` combinator.

Functional Array Fusion is able to fuse `zipWith`N combinators without introducing repeated counter variable but is overall a more fragile system than *Stream Fusion* as it relies on more complex rewrite to function. *foldr/build* fusion is unable to fuse point-wise operations altogether.

The problem of duplicated loop counters was an important problem solved by the proposed *LiveFusion* system to avoid the overhead of performing extra arithmetic operations in the tight loop and reduce register pressure.

2.1.4 Summary

In summary, the problem of multiple consumers and duplicated loop counters are recurring patterns in programs generated by *Data Parallel Haskell* and are pressing performance

problems. In the following chapter I introduce the reader to the inner workings of *DPH* demonstrating the reasons for the ubiquity of such patterns.

Chapter 3

Nested data parallelism

In this chapter I introduce the reader to the context of my work – the *Data Parallel Haskell* (*DPH*) project.

DPH has very specific requirements for the fusion system to satisfy in order to be effective. While the fusion concepts I discuss in later chapters are widely applicable to areas of high performance computing beyond nested data parallelism, *DPH* had a heavy influence on the fusion system I have developed.

3.1 Data Parallel Haskell

Data Parallel Haskell (DPH) is a library for and an extension to the *Haskell* programming language, providing high level access to *nested data parallelism*.

Nested data parallelism is a type of *SPMD* parallelism (single program, multiple data) which operates on *irregular* data structures. Two examples of irregular data structures are *sparse matrices* and *unbalanced trees*.

In a nested data parallel program, parallel computations can call further data parallel computations. For example when traversing an unbalanced tree each node may process each of its children in parallel. Without statically knowing the branching of the tree, it may be difficult to adequately parallelise the process. *DPH* offers an elegant solution through its vectorisation process which will be discussed in Section 3.2.

DPH takes an active part in the optimisation and the compilation of its client code. It guides the compilation process in many ways including vectorisation [42], choosing the optimal data representation [7] as well as applying fusion [9] and rewrite rules [41] to

improve performance.

In essence *DPH* reimplements the familiar list interface in terms of seamlessly parallelised arrays. Listing 3.1 shows some of the most important functions in the *DPH* interface. Since *DPH* has language support, the bracket notation for parallel arrays closely resembles that for *Haskell* lists: `[:a:]` denotes a parallel array of type `a`, while `[:[:a:]:]` is a nested parallel array. The counterparts of *Haskell*'s list comprehensions are also present and is called *parallel array comprehensions*.

```
(!:)          :: [:a:] → Int → a
sliceP        :: [:a:] → (Int,Int) → [:a:]
replicateP    :: Int → a → [:a:]
mapP          :: (a → b) → [:a:] → [:b:]
zipP          :: [:a:] → [:b:] → [:(a,b):]
zipWithP      :: (a → b → c) → [:a:] → [:b:] → [:c:]
filterP       :: (a → Bool) → [:a:] → [:a:]

concatP       :: [:[:a:]:] → [:a:]
concatMapP    :: (a → [:b:]) → [:a:] → [:b:]
unconcatMapP  :: [:[:a:]:] → [:b:] → [:[:b:]:]
transposeP    :: [:[:a:]:] → [:[:a:]:]
expandP       :: [:[:a:]:] → [:b:] → [:b:]

combineP      :: [:Bool:] → [:a:] → [:a:] → [:a:]
splitP        :: [:Bool:] → [:a:] → ([:a:], [:a:])
```

Listing 3.1: Type signatures for parallel array operations.

Collective operations on parallel arrays are executed in parallel when the hardware supports it. The framework is targeting shared memory architectures and uses *Haskell* threads [40] to achieve parallelism.

By design the computation is split evenly across the available processing elements[1] which includes highly irregular parallel programs. The next sections will cover the basics of vectorisation and describe the current stages of fusion in *DPH*.

[1] **Processing element** is a general term referring to processors, processor cores or hardware threads.

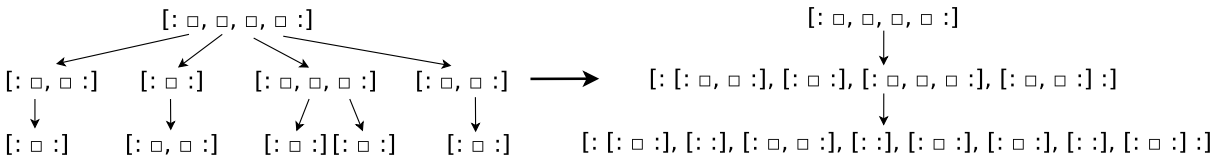


Figure 3.1: Value of type `[:Tree:]` (left) and its flattened representation (right). Empty subtrees are omitted from the conceptual representation.

3.2 Vectorisation

The principle idea behind *DPH* was to allow the programmer write parallel programs without the additional effort of parallelising, scheduling, load balancing and low level optimisation.

The work on *DPH* was originally inspired by Blelloch's pioneering work on *NESL* [4], a research language designed to explore the new approach to nested data parallelism.

At the core of *DPH* is the vectoriser which is implemented in the *Glasgow Haskell Compiler*[2] (*GHC*).

Vectoriser transforms **nested** data parallelism to **flat** data parallelism by means of **flattening transform** on *data* and **lifting transform** on *functions*.

The two transforms are described in the following sections.

3.2.1 Flattening transform

Suppose we wanted to store trees of an arbitrary shape. The following definition of a tree using parallel arrays is not unlike those a Haskell programmer would write using lists:

```
data Tree = Tree Int [:Tree:]
```

A sample tree is given on the left of Figure 3.1. Just by looking at the top level of the tree it would not be possible to know what the branching is.

Now suppose we wanted to apply some function `f` to every node of the tree:

```
visit :: (Int → Int) → Tree → Tree
visit f (Tree v children) = Tree (f v) children'
  where children' = mapP (visit f) children
```

Since the array of `children` may be small, we are unlikely to gain any performance by parallelising the call to `mapP`. Additionally every `visit` to a child node would potentially

[2]Glasgow Haskell Compiler: http://www.haskell.org/ghc

spawn more tiny parallel computations.

The **flattening transform** translates *nested data* in user programs to *flat data representation*.

The right of Figure 3.1 shows the same tree with each of its levels placed into a **nested parallel array**.

At runtime a *nested parallel array* (e.g. each level of a tree) is represented by a **data array** and a **segment descriptor**. The flat data array stores all the nodes' values, while the segment descriptor defines the partitioning to recreate the original nesting.

The following nested parallel array may have been the deepest level of the discussed tree:

```
[: [:5:], [::], [:4,2:], [::], [:3:], [:17:], [::], [:11:] :]
```

At runtime it would be stored as an array of data values together with the segment descriptor containing the lengths and the starting index positions of the contained arrays (offsets into the data array):

```
[ 5, 4, 2, 3, 17, 11 ]       -- data

[ 1, 0, 2, 0, 1, 1, 0, 1 ]   -- lengths (segment descriptor)
```

As a result every level of a tree can be efficiently processed in parallel.

The combinators that operate on the segmented array representation are called **segmented combinators**. They will be discussed is Section 3.5 towards the end of this chapter.

In summary, flattening transform allows the user to define data which is then automatically aggregated into parallel arrays.

In the following section we discuss the lifting transform which vectorises the functions that can be applied to the flattened data.

3.2.2 Lifting transform

We have discussed the way the data is represented in *DPH* but said nothing about how the vectoriser adapts the functions to fit the new data representation.

The lifting transform turns functions on values (`a`) into functions on arrays (`[:a:]`). Similarly, and functions on arrays (`[:a:]`) into functions on nested arrays (`[:[:a:]:]`).

It is not necessary to delve into too much detail here, since by the time the lifted code is compiled in backend of *DPH* (where the fusion takes place) it is no longer a concern as to how it is done.

In order to illustrate the basics of the lifting transform we will use the following function as an example:

```
f :: Float → Float
f x = x * x + 1
```

For every such function the vectoriser generates its *lifted* version `f↑` replacing all inner functions by their *lifted* counterparts and all constants by calls to `replicateP`:

```
f↑ :: [:Float:] → [:Float:]
f↑ x = (x *↑ x) +↑ (replicateP n 1)
  where n = lengthP x
```

This new definition obeys the rule `f↑ = mapP f`, thus it is possible to replace `(mapP f)` with `f↑`.

In the generated code[3] `f↑` becomes the following[4]:

```
f↑ :: Array Float → Array Float
f↑ x = zipWith (+) (zipWith (*) x x)
                   (replicate n 1)
  where n = length x
```

The above code is an example of the code which is compiled against the backend library of flat array combinators. One will notice that this code would produce two intermediate arrays. *DPH* relies on subsequent fusion optimisation to remove these intermediate results and compute `f↑` in one traversal of the input array.

Vectorisation is more thoroughly covered in [42]. However, since much of the code generated by the vectoriser is particularly idiomatic, I will come back to the topic of vectorisation at various points of this thesis.

[3]The code generated by the vectoriser to be used with the backend library of array combinators.

[4]This is no longer true for functions as simple as `f` [26] but it is still a valid example indicative of *DPH* inner workings

3.3 Data representation

Haskell allows for a very high-level of expressiveness when it comes to user-defined types. *DPH* attempts to support this expressiveness and allow the programmer to create and compose types to be used in *DPH* programs. We have already seen an example of a recursive tree type in Section 3.2.1 which is fully supported by *DPH*.

Over the next several pages I will briefly outline how *DPH* chooses the data representation.

3.3.1 Unboxed values

Regular *Haskell* values, such as `Int`s and `Float`s are stored as boxed values on the heap. This means that an array of such values would be stored as array of pointers to the heap allocated values which would be a major hit on performance. It would incur boxing/unboxing costs as well as make it impossible to cache multiple consecutive elements at a time.

Instead, *DPH* uses arrays of unboxed values in its backend which will be referred to as *unboxed arrays*.

3.3.2 Product types

Haskell's product types, or the tuples of two of more heterogeneous elements, are allocated as boxed values on the heap. As discussed, dereferencing heap allocated values considerably affects performance. Instead, an array of pairs `[:(a,b):]` is stored as a pair of arrays `([:a:],[:b:])`. Similarly an array of larger n-tuples becomes an n-tuple of arrays.

This parametric representation is made possible through the use of families as described in [7, 8].

3.3.3 Sum types

One distinctive feature of *DPH* is that is allows one to use user defined data types. An example presenting a `Tree` data type with *one* constructor was given in Section 3.2.1. Likewise one could define a new data type `Shape` with *two* constructors:

```
data Shape = Circle Float
           | Rectangle Int Int
```

In *DPH* the values constructed by the *same constructor* are stored in the same array. To store an array `[:Shape:]` the following arrays are used:

- An array `[:Float:]` that stores radii of all circles
- A pair of arrays `([:Int:],[:Int:])` that sore dimensions of all rectangles
- A *selector* array `[:Int:]` which determines which of the two constructors has been used. The selector array allows to recreate the original interleaving of values constructed with different constructors.

Having multiple arrays, each storing only the relevant values wrapped by the same constructor, not only avoids the cost of unboxing but also avoids the overhead of traditional pattern matching on constructors.

3.4 Backend and Fusion

By the time the backend is reached in the compilation pipeline the nested data parallelism in the original user program has been completely transformed to flat data parallelism:

- The program is now expressed in terms of flat and *segmented array combinators* that take the segment descriptor as an additional argument.
- All data is now represented in terms of flat unboxed data arrays and segment descriptors.
- As discussed in Section 3.3 the vectoriser has also stripped out the product and sum types including those defined by the user and conveniently arranged them in flat arrays.

Thus the backend, or the library of array combinators, only needs to support the arrays of primitive types (`Int`, `Double`, `Bool`, etc. and tuples of these). This is where both the fusion and parallelisation occur.

3.4.1 Previous fusion systems

DPH system originally introduced fusion on two different levels: one removed the superfluous thread synchronisation points introduced after every combinator, while the other, *Stream Fusion*, for fusing the array processing code of each combinator.

3.4.1.1 Removing synchronisation points

Due to the use of *Haskell* threads [40], the code responsible for distributing the processing across the processing elements introduces many thread `fork/join` points. Most collective array operations are individually surrounded by thread `fork` and `join`. This prohibits the application of array fusion discussed in the previous chapter.

Thus if there is no processing to be done between the `join` and the next `fork`, these are fused together using rewrite rules [41].

The handling of synchronisation points in *DPH* as well as the process parallelisation by using *distributed types* and *sequential array combinators* is described in [9].

3.4.1.2 Stream Fusion

Stream Fusion receives the most attention as far as the fusion in *DPH* goes. It is able to statically fuse many types of array combinators while only relying on a single rewrite rule and generic compiler optimisations already available in *GHC* ([43], [37]). A description of *Stream Fusion* is given in Appendix A.2. Just like the fusion that removes superfluous synchronisation points, *Stream Fusion* also heavily relies on inlining [39] and rewriting [41] to work.

Unfortunately, the common point of failure of all of the above fusion types is that they rely on the consecutive operations being adjacent to each other for the rewrite rules to fire.

In particular, the required rewriting will not take place if the result of a combinator is consumed by multiple combinators as discussed in Section 2.1. The ability to fuse heavily branched graphs of combinators is key to achieving high performance in *DPH* programs as we will see in Chapter 7. The work described in this thesis completely replaces *Stream Fusion* to address this and other issues.

3.5 *DPH* combinators

In Chapter 2 on array fusion we discussed fusion as applied to simple combinators operating on flat arrays. This section introduces *segmented combinators* as well as more advanced flat array combinators which consume multiple arrays at once.

3.6 Segmented combinators

With the introduction of segmented array representation by the *Flattening transform* (Section 3.2.1) there is the need for more complex **segmented array combinators**.

All nested arrays in the *DPH* backend are represented by a flat data array and a *segment descriptor* defining the original nesting.

Conceptually *segmented combinators* resemble nested loops in procedural languages, where for each element taken from the segment descriptor array, that many elements of the data array get processed.

We can identify two types of segmented combinators:

1. Combinators that *produce* elements at the *rate* of the segment descriptor, i.e. one element per segment. For example:

   ```
   fold_s max 0 [3,1,2] [1,4,2, 5, 6,8] = [4,5,8]
   ```

2. Combinators that *produce* elements at the *rate* of the data array. For example:

   ```
   scan_s (+) 0 [3,1,2] [1,4,2, 5, 6,8] = [0,1,5, 0, 0,6]
   ```

As seen in the above examples `fold_s` and `scan_s` behave identically to their *Prelude* counterparts as applied to individual segments. Due to the way the parallelism is achieved in *DPH*, the reduction operation f is required to be associative with the neutral initial value z. This is the reason for omitting `l/r` suffix from the combinator names.

In particular, the following must hold:

$f\ (f\ m\ n)\ k) = f\ m\ (f\ n\ k)$

$f\ z\ n = n$

There is also a type of segmented combinators which are a segmented equivalent of *generators*. They produce segmented arrays given certain input arrays. *Segmented replicate*, *enumerators* and *index generator* are all examples of these:

```
replicate_s [2,1,3] [10,20,30] = [10,10, 20, 30,30,30]

enumFromStepLenEach [10,40,60] [1,2,3] [2,4,3]
  = [10,11, 40,42,44,46, 60,63,66]

indices_s [5,3,4] = [0,1,2,3,4, 0,1,2, 0,1,2,3]
```

These combinators roughly fit in the second category (data rate) even though they don't consume a data array as such.

As with the reduction combinators, they behave similarly to their corresponding flat combinators except they are applied across multiple segments.

3.6.1 Advanced multiarray consumers

3.6.1.1 Point-wise consumers

The most obvious and widely used combinator that consumes multiple arrays is `zipWith` (and its `zipWith`N counterparts). It has already been used for point-wise multiplication of two vectors in the beginning of this chapter.

The term **point-wise** describes the manner of array processing where multiple arrays are consumed in lockstep.

Examples of point-wise consumers from the previous section on segmented combinators include *segmented replicate* and *enumerators*:

```
replicate_s [2,1,3] [10,20,30] = [10,10, 20, 30,30,30]

enumFromStepLenEach [10,40,60] [1,2,3] [2,4,3]
  = [10,11, 40,42,44,46, 60,63,66]
```

However, point-wise consumption is not the only pattern seen in *DPH*. Other recurring patterns of consumption of multiple arrays include **interleaved**, **random access** and **consecutive** which we discuss next.

3.6.1.2 Interleaved consumers

Interleaved consumption presumes picking elements from one array at a time but potentially changing the array to pick from from iteration to iteration. *Interleave* and *segmented append* are examples of these. Expectedly interleave picks one element from each array in order, while *segmented append* interleaves the whole segments of arrays:

```
interleave [1,3,5,7,9] [2,4,6] = [1,2,3,4,5,6,7,9]

append_s ([2,1,1],[10,20, 30, 40]) ([1,2,2],[50, 60,70, 80,90])
  = [10,20, 50, 30, 60,70, 40, 80,90]
```

3.6.1.3 Random access consumers

Random access pattern is useful for combinators performing *permutations* on an array given an array of indexes. We distinguish *backwards* and *forward* permutations.

In `bpermute` (back permute) the indexes specify which elements must be picked from the source array and that array must be randomly accessible:

```
bpermute [0,10,20,30,40,50] [3,4,5,1] = [30,40,50,10]
```

This is different from *forward permutation*, where the index array specifies the *destination* index for a particular index of the source array.

```
permute [30,40,50,10,20,0] [3,4,5,1,2,0] = [0,10,20,30,40,50]
```

This pattern allows both arrays to be consumed in a lockstep. Each array may potentially be a result of a fused pipeline of operations. However, `permute` will likely be unable to fuse into its consumer because of out-of-order output generation.

3.6.1.4 Consecutive consumers

This type of consumers is a result of two or more independent array traversals, with one happening after the other. `append` is currently the only combinator which possesses this property:

```
append [1,2,3,4,5] [6,7,8] = [1,2,3,4,5, 6,7,8]
```

3.7 Conclusion

Over the following several chapters I will present my own replacement backend for *Data Parallel Haskell* which enables the fusion in more cases than was possible with *Stream Fusion*.

Chapter 4

Delaying array computations

This chapter describes *LiveFusion* – a new array language that I designed. It offers a library of fusible array combinators and is able to exploit more fusion opportunities than equational fusion frameworks (see Appendix A)

In its essence *LiveFusion* takes the approach of *deeply embedded domain specific languages* (EDSLs).

4.1 On embedded domain-specific languages

The term **domain-specific language** (DSL) describes a computer language designed for solving problems in a particular application domain.

Examples of DSLs include *HTML* for webpages markup [22], *SQL* for relational database queries [21] and *Verilog* for hardware description [19].

In contrast, *Haskell* [38] is a *general purpose language* and is more widely applicable across domains.

On the other hand, an **embedded domain-specific language** (EDSL) is a domain-specific language which is designed to be used from within another language, called *host* language. It is implemented as a library for the host language and can reuse its syntax, compiler and the runtime system.

The added benefit of the approach is that the EDSL code can interact with the rest of the code in the host language. This way the programmers do not have to leave the familiar environment and can write EDSL programs which are seamlessly integrated with their programs.

Two types of embedding exist:

Shallow embedding

Shallowly embedded DSLs directly translate their constructs to expressions in the host language. They offer a fixed interpretation of their operations.

Parsec[1] library of parsing combinators is an example of a shallow EDSL for *Haskell*.

Deep embedding

The principle idea behind **deeply embedded DSLs** is that all operations of such an EDSL construct a tree of operations, or an **abstract syntax tree** (AST). This new AST is a data structure in the host language. It is separate from the host language's AST and is rather presented as a tree-like structure in the host language's runtime.

The difference from the ASTs found in compilers is that it allows for analysis, optimisation and compilation (or interpretation) of the EDSL's AST at runtime. The complete process is taken care by the library written in the host language. This avoids the difficulties of writing a complete standalone optimising compiler.

Accelerate language for functional programming on GPUs [11] is a vivid example of a deeply embedded DSL.

LiveFusion EDSL falls under the category of deeply embedded domain-specific languages. In this chapter I describe the process of embedding array computations.

Figure 4.1 presents the overview of the *LiveFusion* language and the way it fits into the *Data Parallel Haskell* framework. While many of the concepts in the figure have not yet been discussed, it can serve as reference to the reader in future chapters.

4.2 *LiveFusion* EDSL by example

Figure 4.2 presents a program that can be written using *LiveFusion* library as well as the corresponding data flow diagram.

The function uses the familiar `map`, `filter` and `fold` combinators. Operators and functions of `Num` type class (`+`, `-`, `abs`...) and number literals (0, 0.5, 1...) are supported directly. Versions of `Ord` and `Eq` operators are available postfixed with a dot (`>.`, `==.` ...).

[1] parsec: Monadic parser combinators - http://hackage.haskell.org/package/parsec

§ p.17

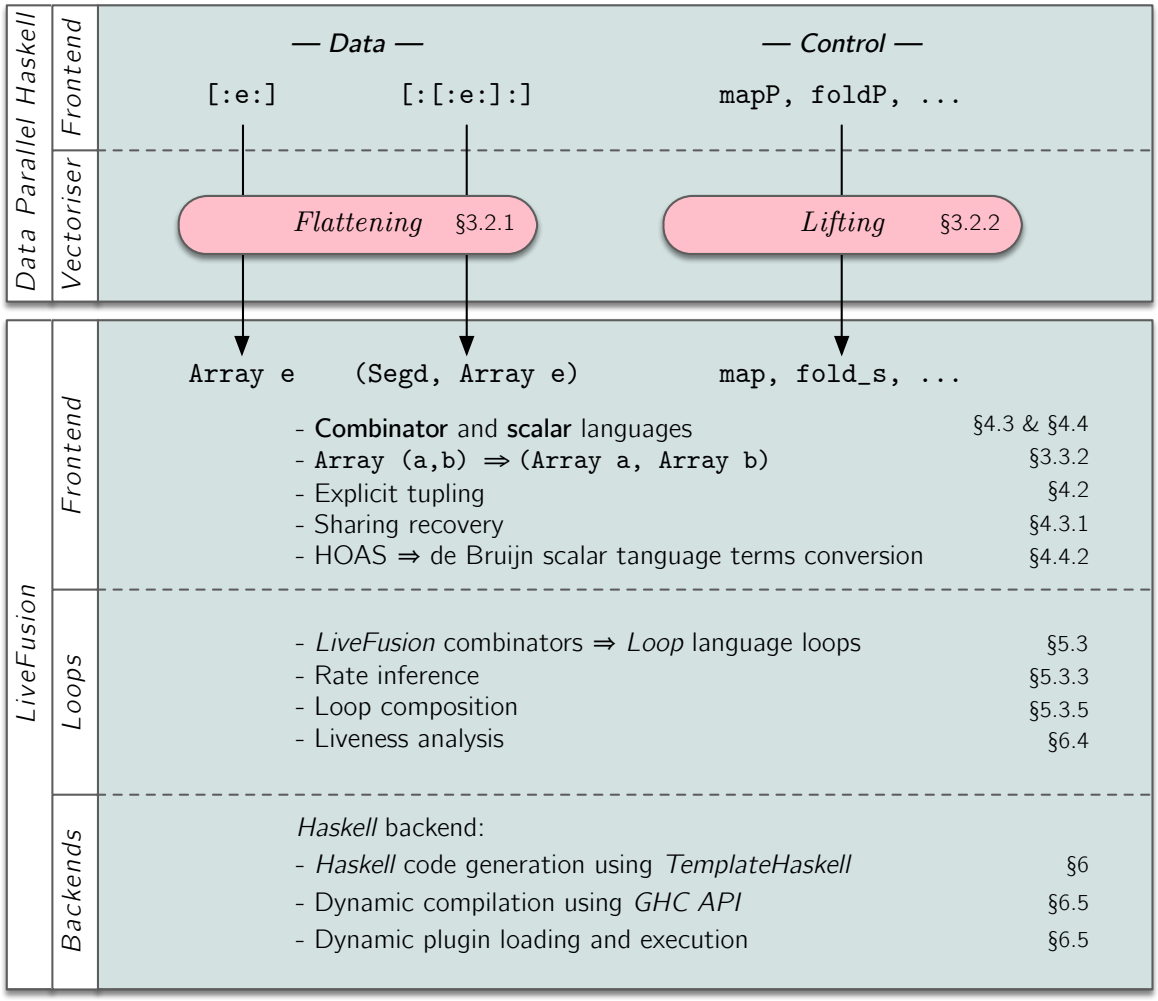


Figure 4.1: Overview of *DPH* and *LiveFusion* systems.

```
{-# LANGUAGE OverloadedLists #-}

filterMax xs
  = let xs' = map (+1) xs
        pos = filter (>. 0) xs'
        mx  = fold max 0 xs'
    in  (pos :*: mx)

main = print
     $ filterMax [1,-2,5,0]

-- Output:  (fromList [2,6,1], 6)
```

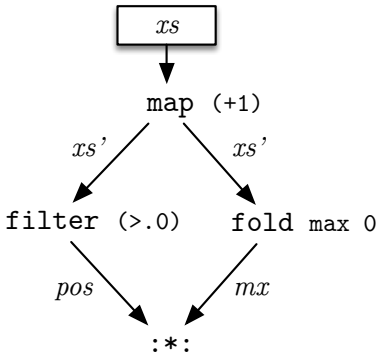


Figure 4.2: Example of a *LiveFusion* program (left) and its data flow diagram (right).

The example employs the explicit tuple constructor `(:*:)`, which brings together the branches computing `pos` and `mx` to be analysed for fusion. It addresses the problem of *tupling* that prohibits fusion of independent terms in a pure context.

A vectorised version of this function will be presented in the Section 7.4 when we look at the *QuickHull* algorithm implementation in *DPH*.

4.3 *LiveFusion* interface and AST

LiveFusion EDSL offers a library of flat and segmented array combinators some of which are presented in Listing 4.1.

```
type Array  a = ArrayAST  a
type Scalar a = ScalarAST a

------------------------- Flat array combinators -------------------------
map      :: (Term a → Term b) → Array a → Array b
filter   :: (Term a → Term Bool) → Array a → Array a
zipWith  :: (Term a → Term b → Term c) → Array a → Array b → Array c
zip      :: Array a → Array b → Array (a,b)
fold     :: (Term a → Term a → Term a) → Term a → Array a → Scalar a
scan     :: (Term a → Term a → Term a) → Term a → Array a → Array a
replicate:: Term Int → Term a → Array a
bpermute :: Array a      -- Source array
         → Array Int     -- Indices to pick elements from
         → Array a
packByTag:: Term Bool → Array Bool → Array a → Array a

------------------------ Segmented array combinators ------------------------
scan_s   :: (Term a → Term a → Term a) → Term a
         → Array Int → Array a → Array a
fold_s   :: (Term a → Term a → Term a) → Term a
         → Array Int → Array a → Array a
replicate_s :: Term Int → Array Int → Array a → Array a
```

Listing 4.1: *LiveFusion* interface functions. Typeclass constraints on array elements have been omitted for brevity.

The fundamental concept by which *LiveFusion* makes fusion possible is constructing an abstract syntax tree (AST) of pending array operations at runtime and compiling that AST to efficient code when the result is required in the host program. Thus the compile

```
type ArrayAST  a = AST (Vector a)
type ScalarAST a = AST a

data AST t where
  ---------------- Flat combinators ----------------
  Map           :: (Elt a, Elt b)
                ⇒ (Term a → Term b)
                → ArrayAST a
                → ArrayAST b

  Filter        :: Elt a
                ⇒ (Term a → Term Bool)
                → ArrayAST a
                → ArrayAST a

  Fold          :: Elt a
                ⇒ (Term a → Term a → Term a)
                → Term a
                → ArrayAST a
                → ScalarAST a

  Replicate     :: Elt a
                ⇒ Term Int
                → Term a
                → ArrayAST a
  ------------- Segmented combinators -------------
  Fold_s        :: Elt a
                ⇒ (Term a → Term a → Term a)
                → Term a
                → ArrayAST Int
                → ArrayAST a
                → ArrayAST a

  Replicate_s :: Elt a
                ⇒ Term Int
                → ArrayAST Int
                → ArrayAST a
                → ArrayAST a
  ---------------- Manifest arrays ----------------
  Manifest      :: Elt a
                ⇒ Vector a
                → ArrayAST a
  ---------------- Explicit tupling ----------------
  (:*:)         :: (Typeable t1, Typeable t2)
                ⇒ AST t1
                → AST t2
                → AST (t1,t2)
```

Listing 4.2: *LiveFusion* AST (partial).

time of *LiveFusion* language maps to the runtime of its host language – *Haskell*.

The AST is the topmost layer of *LiveFusion* system. Most of the library's user-facing functions are just constructing the nodes of the AST. Parts of *LiveFusion* AST are presented in Listing 4.2. Is it easy to spot the correspondence between the user-facing functions from Listing 4.1 and the constructors of the AST.

LiveFusion AST is implemented as a *Generalised Algebraic Data Type* (GADT) [23] in order to have more control over the types of individual constructors.

In principle, given an interpreting function such as `eval :: AST t → t`, a value of type `t` encoded in the `AST` language can be computed. Notably, `t` in `AST t` is not necessarily an array. In fact not all of the array combinators return an array: in particular `fold` and several others return a scalar value.

4.3.1 Sharing recovery

In pure functional languages like *Haskell* sharing of terms is *not observable*. That is, given two terms it is not possible to tell whether or not they reside at the same memory address (as it is possible with pointers in a procedural language). Thus, if *LiveFusion* AST constructed for the earlier `filterMax` example (Figure 4.2) was traversed, `Map` and `Manifest` nodes would be encountered twice as in Figure 4.3 (left). Naively compiling such code would result in two computations of the result of `Map`.[2]

Nonetheless, implementing support for sharing in an EDSL is possible. An overview of several approaches is given in [27]. There is a distinction between *explicit* and *implicit* sharing. When using explicit sharing, the sharing of terms is specified by the programmer using some explicit construct.

Moving on to *implicit* sharing, there are techniques based of *structural equality* of terms where two subtree of terms are considered shared if they are equal in structure and values. This approach is possible to implement without leaving the pure context (provided terms can be compared for equality). However, the experience of *Accelerate* language authors [11] shows that this approach suffers from severe complexity blowups for sufficiently large problems.

The second group of *implicit* sharing approaches involves using techniques to escape

[2]Notably, this is an instance of fusion into multiple consumers identified as a major challenge for a fusion system to overcome in Section 2.1.2.

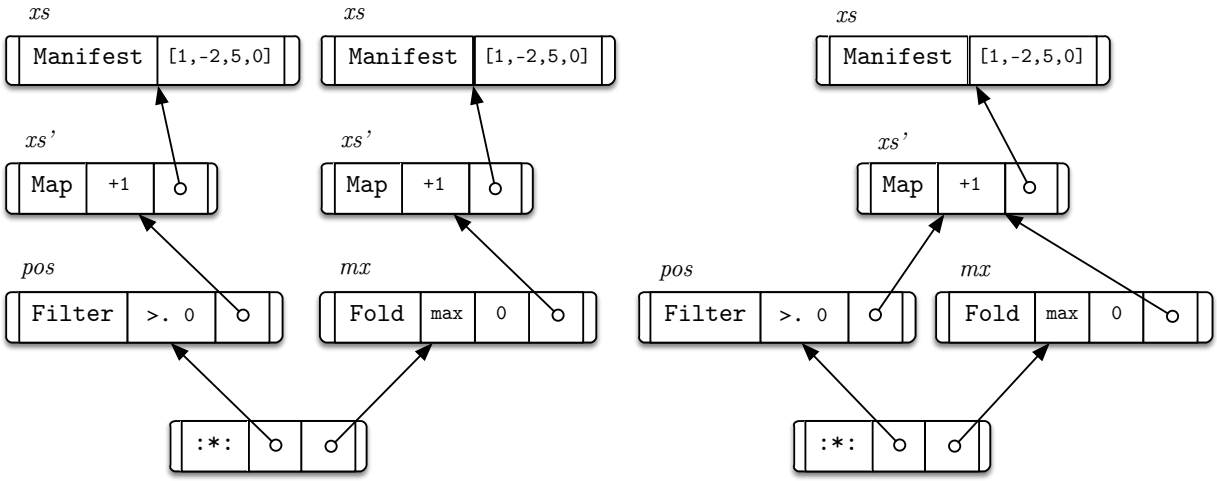


Figure 4.3: *Left:* Conceptual representation of `filterMax` AST in a pure context. *Right:* Likely runtime representation of `filterMax` node graph. Desired result of sharing recovery.

the pure context to acquire pointers or otherwise stable references to a term. Sharing recovery using this approach grows linearly in the size of the tree. *LiveFusion* incorporates sharing recovery of this type. The approach is called *Observable Sharing* and is described in [18].

More specifically, the approach is adapted to support a GADT AST [53]. An *abstract semantic graph (ASG)* data type is specified (Listing 4.3) which has the same constructors as the original AST in addition to `VarG` – a representation of a shared variable in the graph. Then a `mapDeRef` function is implemented that converts the AST to ASG, *replacing every point of recursion with a* `VarG` node. Sharing recovery of `filterMax` results in the following graph, where recursive variable references have been replaced with unique integers:

```
-- Nodes:
[
  (1,Wrap (BothG (VarG 2) (VarG 5))),
  (2,Wrap (FilterG <function> (VarG 3))),
  (3,Wrap (MapG <function> (VarG 4))),
  (4,Wrap (ManifestG (fromList [1,-2,5,0]))),
  (5,Wrap (FoldG <function> 0 (VarG 3))),
]
-- Entry node:
1
```

While the approaches to sharing recovery based on obtaining unique references to the term are generally non-portable and unsafe, it is guaranteed to be free of false positives. That is no erroneous sharing of terms that are not shared will occur. While false negatives

are a possibility (i.e. unrecovered sharing), they are rare in practice.

```
data ASG t s where
  MapG      :: (Elt a, Elt b)
            ⇒ (Term a → Term b)
            → ArrayASG a s
            → ArrayASG b s
  ...

  VarG      :: Typeable t
            ⇒ s
            → ASG t s


instance Typeable t ⇒ MuRef (AST t) where
  type DeRef (AST t) = WrappedASG
  mapDeRef ap t = Wrap <$> mapDeRef' ap t
    where
      mapDeRef' :: Applicative ap
                ⇒ (∀ b. (MuRef b, WrappedASG ~ DeRef b) ⇒ b → ap u)
                → AST t
                → ap (ASG t u)

      mapDeRef' ap (Map f arr)
        = MapG f
          <$> (VarG <$> ap arr)

      mapDeRef' ...
```

Listing 4.3: Abstract semantic graph type for *LiveFusion* AST and AST conversion function.

4.4 Parametrising higher-order combinators

Another notable feature of the interface to note here is the type of functions that parametrise higher-order combinators:

```
map    :: Elt a ⇒ (Term a → Term b) → Array a → Array b
filter :: Elt a ⇒ (Term a → Term Bool) → Array a → Array Bool
fold   :: Elt a ⇒ (Term a → Term a → Term a) → Term a → Array a
                                                → Scalar a
```

The corresponding *Haskell* list functions or array libraries like `Vector` or `Repa` [25] accept as arguments vanilla *Haskell* functions like `a → b` or `a → a → a` as long as

the types match.

The question that should be asked, however, is why cannot *LiveFusion* do the same?

The answer lies in the approach to evaluating the AST. In order to be efficient the AST needs to be compiled to highly optimised code. During compilation (discussed in Chapter 6) the AST is compiled *at runtime* to one or more loops for which *Haskell* code is generated.

If the parametrising functions were simple *Haskell* functions, they would have been compiled to machine code *before* the program is run. When the AST is later constructed at runtime they would only be available in the form of closures which can be called but not inspected[3].

Suppose the user writes `map (+1) xs`. The function `(+1)` needs to be inlined into the compiled loop code to produce efficient code. If this does not happen, calling `(+1)` for every element in the `xs` array will result in the following:

1. Boxing the element value by placing it in the heap
2. Calling the closure with the pointer to the boxed value
3. Jumping to the address of the `(+1)` function referenced in the closure
4. The code will unbox the value, increment it and create a new value on the heap
5. The loop code will then need to unbox the result and write it into the result array

This is a very round-about way of incrementing a value in a tight loop and will greatly affect the performance of the loop.

Unfortunately, at the time of generating code for AST, all these Haskell functions will have been compiled to machine code and are only available as closures. There is currently no way to inline them or reify them into the original *Haskell* source code from which they were created.

I needed to find a way to generate efficient code for user specified functions at runtime. I needed a way to record the user-provided functions in a more high level way than machine code.

In summary the chosen representation for functions:

1. Provides backend independent interface offering many common functions
2. Allows backend-specific implementation for each function

[3]There is a non-portable way to explore closures in *Haskell*, but it will not allow one to easily make use of the code compiled to binary.

3. Hides implementation from the user by default
4. Allows user to compose functions in a way that looks native
5. Allows user to provide direct backend-specific implementation for new functions that cannon be composed from functions already provided

The following section explains how this is achieved.

4.4.1 Scalar Term language

An expression of the type `Term a → Term b` is a term in the scalar expression language called *Term*, representing a function from type `a` to type `b`. The AST for the *Term* language is given in Listing 4.4.

```
data Term t where
  -- Function or constant in backend-specific form
  Con :: Impl t → Term t

  -- Lambda abstraction
  Lam :: (Term s → Term t) → Term (s → t)

  -- Function application
  App :: Term (s → t) → Term s → Term t

  -- for conversion to de Bruijn representation
  Tag :: Level → Term t

type Level = Int
```

Listing 4.4: Term language for HOAS representation.

The language allows the user to write functions in the higher-order abstract syntax (HOAS) form which looks native and in many cases reuses the familiar operators and function names:

```
f :: Term Int → Term Int
f x = x * x + 1
```

The library provides backend-specific implementations for many standard functions including all of the functions from such type classes as `Num`, `Floating`, `Ord`-like and `Eq`-like, as well as the ability to compose them into more complex functions.

4.4.2 *HOAS* vs. *de Bruijn* representation

Higher-order abstract syntax presented above is very convenient for library users to write. However, this representation is inconvenient for the embedded language compilers and interpreters to work with. In particular, it makes is impossible to inspect bodies of lambda functions.

The conversion of *HOAS* to an alternative representation based on *de Bruijn indices* has been independently discovered in [6] and [2]. The conversion code employed in *LiveFusion* is due to my supervisor Manuel Charkravarty, the author of the former approach.

The new representation offers an easier way to analyse embedded terms and is the assumed representation of scalar functions in the rest of this paper.

4.4.3 Backend specific function implementations

One notable part of the `Term` language is its `Con` constructor. The only argument to the constructor is `Impl t` offers a backend-dependent way of to define a function. `Impl` data type gives freedom to the backend to choose the most suitable representation for a function.

For instance, in the *Haskell* backend for *LiveFusion* we are interested in generating *Haskell* source code, for which *Template Haskell* [46] is a natural choice. Thus we choose the following definition of `Impl`:

```
data Impl t = HsImpl {
                hs :: t,          -- Native Haskell function
                th :: Q TH.Exp    -- TemplateHaskell quasiquoted expression
              }
```

The *Template Haskell* expression is really the core part of the function representation, but this is what would later on allow for an easy production of *Haskell* source code at the code generation stage.

The representation also includes the vanilla *Haskell* function. At present this is just for completeness. However in the future it may be possible to avoid compiling certain parts of AST at runtime and run statically scheduled combinators. In this case this would be the function to inline into the loop.

We will now see the (rather trivial) implementations for a couple of functions before continuing with our explanation of Impl.

```
plusImpl :: Num a ⇒ Impl (a → a → a)
plusImpl = HsImpl { hs = (+); th = [| (+) |] }

absImpl :: Num a ⇒ Impl (a → a)
absImpl = HsImpl { hs = abs; th = [| abs |] }
```

It is easy to see that *Template Haskell* expressions are trivially created from simple *Haskell* functions using *quasi-quotation* extension [32].

When the backend is ready to generate *Haskell* source for these functions, it will be a simple matter of using a *Template Haskell's* pretty printer.

The following are the reasons for using *Template Haskell* expressions to represent the user specified functions:

- The backend does non-trivial code generation as discussed in Chapter 6, where the full power of *Template Haskell* is required. Functions defined this way become very easy to incorporate into the generated code
- New developments in *Template Haskell*[4] will allow its expressions to be typed. Having `Q (TExp t)` as opposed to `Q Exp` will provide an extra layer of safety to the function representation

[4]Starting with GHC 7.8.

4.5 Related work

4.5.1 *DESOLA*

DESOLA (*Delayed Evaluation Self Optimising Linear Algebra*) [45] is an experimental *C++* library designed to explore the benefits of runtime code generation and optimisation for scientific computing. In contrast to *DPH*, *DESOLA* does not support higher order combinators or segmented operations, although it does support regular multidimensional arrays.

4.5.2 *Accelerate*

Accelerate EDSL [11, 35, 13] is probably the closest, by design, to *LiveFusion*. It was initially designed to offer a simple but powerful programming model targetting General Purpose GPU computing. When I was starting my work on *LiveFusion*, *Accelerate* was able to generate efficient code for GPUs but lacked fusion. Additionally, it only supported regular arrays (although with a very powerful *shape-polymorphic* model similar to that of *Repa* [25]). *Accelerate* is currently moving towards having more sophisticated fusion, nested parallelism and ability to generate code for CPUs.

4.5.3 *Flow Fusion*

Embedding EDSLs in a way that they construct their intermediate data structures at the runtime of the program is by far the most common approach but not the only one. An EDSL can also be implemented in a host language compiler.

In particular *GHC* allows to specify transformations of its internal *Core* language in the form of plugins integrating into the compilation pipeline. *Flow Fusion* framework [31, 16] was designed to fuse combinator graphs at compile time. The data flow graph to be fused is translated to *DDC Core* language – the intermediate language of *Disciplined Disciple Compiler* [30] – where it is fused using techniques that inspired the work on *LiveFusion* and are discussed in the next chapter: rate inference (Section 5.3.3) and compiling to a common imperative loop structure (Section 5.1).

The approach of *Flow Fusion* solves sharing by analysing the entire graphs of combinators and avoids runtime overheads of building, analysis and compilation of ASTs. The

downsides of *Flow Fusion* are that control flow breaks fusion. However, in *DPH* control flow is not an issue since most of control flow is transformed into data flow by the vectoriser.

Flow Fusion is currently in its early development stages and is not yet able to fuse segmented combinators. In the future it may become a very comprehensive fusion platform. In fact there are plans for building the CPU backend for *Accelerate* on top of *Flow Fusion*.

Chapter 5

Generic loop representation and rates

LiveFusion at the top level is a library of high level array combinators. Most of these conceptually represent a loop. Indeed, the user of the library can reason about the individual combinators as loops and consider the result of each to be an array.

However, when running the user's program, we are aiming to perform the required operations in as few loops as possible[1]. Subject to certain restrictions, two combinators in *LiveFusion* are fusible when the output array *produced* by one combinator is *consumed* as input by another combinator one element at a time from beginning to end.

In the *LiveFusion* AST discussed in the previous chapter, this relationship is usually seen between a child node (the producer) and its parent node (the consumer). This is not true for *random access* combinators like `backpermute` which we discuss separately.

Intuitively, two combinators can be fused whenever one could write a loop by hand which would give the same final result for the same input as the two separate loops would.

The focus of this chapter is to establish a common loop representation to which fusible combinators can be mapped.

In the following sections we will look more closely at what a loop really is and then present the core of our fusion optimisation.

[1]This may increase register pressure. However, unless register spilling poses a problem in the future we favour the decreased memory traffic which is attained by array fusion.

5.1 Anatomy of a loop

Despite the purely functional, combinatorial interface of the library, looking at a loop in a procedural way is the approach I have opted for in the middle layer of the system. It is represented by the *Loop* language.

The loops *LiveFusion* generates can be viewed as similar to those one might write in an Assembly language. It uses labelled basic blocks and has explicit control flow using `goto` statements[2].

However, as we will see later, the loops are more structured than those in *C* and many other procedural languages. As such I urge the reader to think of them as being high-level and treat the explicit control flow as implementation details.

Without further delaying the discussion of the matter we will now look at the structure of a typical `for` loop in a language like *C* in an attempt to shape our own loop structure.

5.1.1 Structure of a `for` loop

Consider the following fragment of a *C* program that creates a new array `ys`, by applying some function `f` to those elements of array `xs` that satisfy some predicate function `p` (in *Haskell* this could be expressed as `ys = map f $ filter p $ xs`):

```
1 double *ys = malloc(len * sizeof(double)); // result array
2 int j = 0;                                 // output index
3 for(int i = 0; i < len; i++) {
4     if(p (xs[i])) {
5         ys[j] = f (xs[i]);
6         j++;
7     }
8 }
9 ys = realloc(ys, j * sizeof(double));
```

A `for` loop in *C* has four sections:

1. *initialisation* section (`i = 0`),
2. *guard* section (`i < len`),
3. the main *body* (`if ...`), and finally
4. the *update* section (`i++`).

[2]The formal grammar of the *Loop* language is introduced later in the chapter. It is presented in Figure 5.3 on page 51

Compared to free-formed `while` loops which only have a `guard` and a `body`, the `for` loops are already much more structured.

However, as we see next, further structural elements could be introduced which will ultimately assist us when composing loops from array combinators.

5.1.1.1 Initialisation

The very first observation to make is that both the result array `ys` and the output index `j` were declared and initialised outside the `for` loop.

In pursuit of a more structured and composable approach to looping all statements that are executed *once* before the loop begins are placed in the `init` basic block of the loop.

The initialisation code corresponds to the following in the *Loop* language:

```
init:
  let len = arrayLength xs
  let ys = newArray len
  let i = 0
  let j = 0
  goto guard
```

5.1.1.2 Guard

The `guard` section of a `for` loop corresponds to the `guard` block in the *Loop* language. It can contain arbitrary statements but usually contains at least one `unless` statement:

```
guard:
  unless i < len | done
  goto body_xs
```

The `unless` *statement* transfers the control to a different block if the specified condition is *false* (to `done` in this case, the finaliser block discussed later). Otherwise the control stays in the current block, which in this case results in entering the loop's body.

5.1.1.3 Update

We defer presenting the *Loop* language's equivalent of the `update` section until after the structure of loop bodies is discussed shortly.

5.1.1.4 Finalisation

In the general case the `filter` results in a shorter array than its input. Hence, after the `for` loop finishes, the resulting array is `realloc`'ated to free the unused memory (in practice the array is unlikely to be copied).

This is one of the use cases for what is called the `done` block of the loop:.

```
done:
  let result = sliceArray ys j   -- resize to length j
  return result
```

As seen from the code, every loop has a result which it returns explicitly.

5.1.2 "Dissecting loop bodies"

We will now attempt to categorise the types of operations that all belong to the bodies of conventional `for` and `while` loops.

We continue using example presented on page 42 as reference. In particular, the `body` of the `for` loop contained the following statements:

```
4 if(p (xs[i])) {
5     ys[j] = f (xs[i]);
6     j++;
7 }
```

5.1.2.1 Isolating combinators

Line 5 of the `for` loop, the statement `ys[j] = f (xs[i])`, is actually performing three operations:

1. *reading* an element from array `xs`,
2. *producing* a new element by applying function `f`, and
3. *writing* the new element into array `ys`.

Recall, however, that we are trying to devise a loop representation for an application of fusion. When several combinators are fused into a single loop and produce no intermediate arrays, such reading and writing only happens at the beginning and at the end of a combinator pipeline .

Hence we want to separate the notion of *producing* a new element from the fact that it came from a physical array or just another computation. Likewise, we should not

be concerned with how the new element is going to be consumed. It may or may not be written into a new array. It may or may not be used by a consuming combinator. However, it should not be up to an individual combinator to decide.

Each combinator in the pipeline is responsible to only take care of its own processing. In particular each combinator fills in its own `body`, `yield` and `bottom` basic blocks which we discuss next.

5.1.2.2 Computing a new element in the `body`

As discussed previously, each combinator fills in a number of blocks with statements specific to it. For now we will only focus on the `body` blocks produced:

```
body_xs:
  let x = readArray xs i

body_filt:
  unless (p x) | bottom_filt

body_map:
  let y = f x
```

The reading of `xs` array is treated as a separate combinator and results in a `readArray` statement. The `body` of the `filter` is a conditional jump if the predicate `p` is not satisfied. Lastly, the `body` of `map` is the application of function `f` to element `x` and binding it to a fresh variable `y`.

We know however, that if `filter` produces and element, then so does `map`. We say that `filter` and `map` produces elements at the same **rate**. Thus we can *merge* the respective blocks of the two:

```
body_xs:
  let x = readArray xs i

body_filt/map:
  unless (p x) | bottom_filt
  let y = f x
```

It was mentioned that each combinator also introduces `yield` and `bottom` blocks.

The complete *Loop* generated for `ys = map f $ filter p $ xs` as well as its control flow graph (CFG) are shown on Figure 5.1. In addition to `init`, `guard`, `body` and `done` blocks it also contains `yield` and `bottom` block with the required control flow. A step-by-step explanation of these spans the following four sections.

5.1.2.3 Yielding produced elements

The *Loop* language attempts to be precise as to where the element is produced and where it is consumed. More concretely, the `yield` block of the loop is only ever entered if an element has been produced in the current iteration.

The `body` block contains the code that is concerned with producing an element, however an element is known to have been produced only if the control reaches the `yield` block.

Every `body` block on the listing is accompanied by the `yield` block. The combinator reading the `xs` array in $\texttt{body}_{xs}$ produces an element `x` at every iteration, hence the corresponding $\texttt{yield}_{xs}$ block is entered unconditionally.

On the other hand, the `filter` (and the subsequent `map`) may skip an element hence the possibility possibility of bypassing $\texttt{yield}_{filt/map}$ block.

This allows us to make assumptions about the behaviour of a loop without knowing what the loop is doing internally.

5.1.2.4 Writing an element into result array

The separation of `yield` and `body` has allowed for a finer grained loop structure.

In particular, if the result of a combinator has to be materialised into the physical array (as in the the case of `map` in our example), the `writeArray` statement needs to be inserted into $\texttt{yield}_{map}$:

```
writeArray ys j y  -- write element y into array ys at index j
```

5.1.2.5 Updating index

The `yield` block also provides a convenient place to update the index variable.

In our example the loop contains two *rates* :

- one at which the elements are read from the source array,
- and one at which the elements of `filter` and `map` are produced

With the current loop structure, updating both `i` and `j` index is a matter of placing an increment statement in the right `yield` block:

```
yield_xs:
  i := i + 1
  goto body_filt

yield_filt/map:
  writeArray ys j y
  j := j + 1
  goto bottom_map
```

As seen in the code above *Loop* language supports destructive updates through using `assignment` (`:=`).

This introduction of the index update is uniform for all *rates* that may be present in the loop. This is in contrast to the `for` loop originally presented on page 42 where `i++` was part of `update` section, while `j++` was inside the `for` loop's body.

5.1.2.6 Ending the iteration

When the iteration has finished the last block that is entered *unconditionally* (for each rate) is `bottom`. In the given examples the two `bottom` blocks simply transfer the control to the beginning of the next iteration, the `guard`. However, in more sophisticated loops, e.g. `append` combinator, the `bottom` blocks may serve other purposes.

5.1.3 Summary of loop structure

In summary the common loop structure to be used throughout this chapter[3] is comprised of the following 6 loop sections, represented by basic blocks in the *Loop* language:

1. `init` block is the main entry into the loop and contains statements which need to execute only once for the whole loop. New array allocation, index and length variable initialisation all belong here.

2. `guard` block performs looping condition tests before every iteration of the loop.

3. `body` block contains all statements concerned with reading arrays and computing elements. This block is *rate*-specific and there is a new `body` block for every distinct rate in the loop.

[3] At least until segmented combinators are introduced.

```
init:
  let len = arrayLength xs
  let ys = newArray len
  let i = 0
  let j = 0
  goto guard

guard:
  unless i < len | done
  goto body_xs

body_xs:
  let x = readArray xs i
  goto yield_xs:

yield_xs:
  i := i + 1
  goto body_filt



body_filt/map:
  unless (p x) | bottom_filt
  let y = f x
  goto yield_map:

yield_filt/map:
  writeArray ys j y
  j := j + 1
  goto bottom_map

bottom_filt/map:
  goto bottom_xs



bottom_xs:
  goto guard

done:
  let result = sliceArray ys j
  return result
```

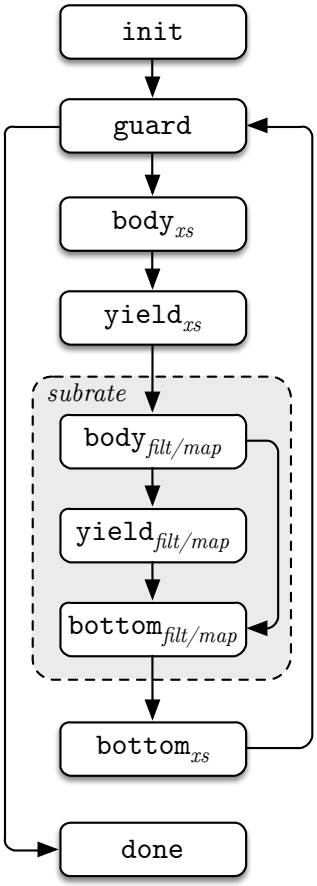


Figure 5.1: Internal *Loop* language representation for `ys = map f $ filter p $ xs` (left) and the corresponding CFG (right).

4. `yield` block for a particular rate is entered only if a *new element* has been produced by the `body` block of the same rate in the current iteration. It is not entered if the loop's logic has skipped to the next iteration without producing an element (e.g. in `filter` combinator). This is the place to update the loop index as well as *write* the produced element in the resulting array if required.

5. `bottom` block is entered unconditionally at the end of every iteration whether of not an element has been produced. Uses beyond going back up to the beginning of the next iteration will be seen later in this chapter.

6. `done` block performs any remaining operations after the loop has finished. In particular it returns the final result from the loop. Unlike the loops in the majority of procedural languages the `return` of values *from a loop* is explicit in the *Loop* language.

Figure 5.2 shows the default control flow between these sections.

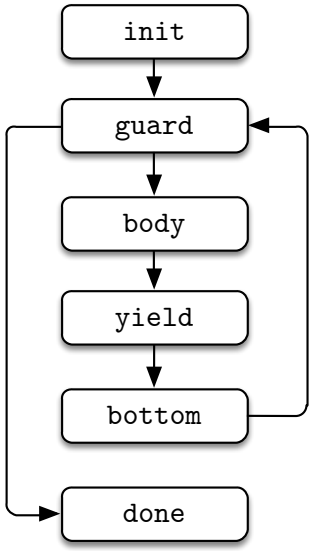


Figure 5.2: Default control flow between basic blocks in the *Loop* language.

5.2 The *Loop* language more formally

So far we have used the following features of the *Loop* language:

1. New variable binding (`i = 0`)
2. Variable assignment (`i := i + 1`)
3. Explicit control transfer (`goto body`)
4. Conditional control transfer (`unless i < len | done`)
5. Returning values from loop (`return ys'`)
6. A number of built-in array primitives: (`newArray`, `readArray`, `writeArray`, `arrayLength` and `sliceArray`).

All of the above are *statements*, which are grouped into labelled *basic blocks*.

The grammar of the *Loop* language is presented formally in Figure 5.3. The syntax seen in the examples throughout this thesis is the same as used by the pretty printer for the language and it faithfully[4] reproduces the internal representation of *Loop* programs.

If the grammar is studied more carefully one would note that one basic block may have multiple associated labels. We have seen a use case for this when we merged `body`, `yield` and `bottom` basic blocks of `filter` and `map` combinators (Figure 5.1). Such merged blocks can be referenced by either label. For example, $\texttt{bottom}_{filt/map}$ was being referenced both by $\texttt{bottom}_{filt}$ and $\texttt{bottom}_{map}$.

It should also be noted that each label and variable in the language is postfixed with a unique value which has so far been omitted from the listings. However, $\texttt{body}_{xs}$ is an example of such label identification. At the implementation level these values are unique integers. However, for improved readability they are replaced with combinator names and other meaningful identifiers.

Being a flexible assembly-style language, the *Loop* language is not limited by the basic block structure discussed in Section 5.1 and summarised in Section 5.1.3. However, as seen in the rest of this chapter this structure has proven to be a useful common base for many types of arrays combinators. Thus we assume this to be the common pattern of using the *Loop* language.

[4] I have taken the liberty of rewriting operators in infix notation and omitting explicit literal conversions such as `fromInteger 1`.

[5] For improved readability the unique integers are replaced with meaningful names in the code listings. *(Footnote for listing 5.3)*

$$
\begin{array}{lcl}
\mathit{name} & \rightarrow & (\texttt{arr}, \texttt{elt}, \texttt{f}, \texttt{acc}, \texttt{init}, \texttt{body}, \text{ etc.}) \\
\mathit{id} & \rightarrow & (\text{unique integer}^{5}) \\
\mathit{impl} & \rightarrow & (\text{backend specific code, e.g. } \textit{Template Haskell} \text{ expression}) \\
\mathit{lit} & \rightarrow & (\textit{Haskell} \text{ value of a supported type (Int, Float, Bool, tuple, etc..)} \\
\mathit{var} & ::= & \mathit{name}\ \mathit{id} \\
\mathit{label} & ::= & \mathit{name}\ \mathit{id}
\end{array}
$$

$$
\mathit{loop} \quad ::= \quad \overline{\mathit{block}}\ \overline{\mathit{var}_{arg}}\ \mathit{label}_{entry}
$$

$$
\mathit{block} \quad ::= \quad \overline{\mathit{label}}\ \overline{\mathit{stmt}}\ \mathit{stmt}_{final}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \texttt{let}\ \mathit{var}\ \texttt{=}\ \mathit{expr} \\
& | & \mathit{var}\ \texttt{:=}\ \mathit{expr} \\
& | & \texttt{if}\ \mathit{expr}_{bool}\ \texttt{|}\ \mathit{label}_{true}\ \mathit{label}_{false} \\
& | & \texttt{unless}\ \mathit{expr}_{bool}\ \texttt{|}\ \mathit{label} \\
& | & \texttt{goto}\ \mathit{label} \\
& | & \texttt{return}\ \overline{\mathit{var}} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{newArray}\ \mathit{expr}_{length} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{readArray}\ \mathit{var}_{array}\ \mathit{expr}_{index} \\
& | & \texttt{writeArray}\ \mathit{var}_{arr}\ \mathit{expr}_{index}\ \mathit{expr}_{element} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{sliceArray}\ \mathit{var}_{arr}\ \mathit{expr}_{new\_length} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{arrayLength}\ \mathit{var}_{arr}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{var} \\
& | & \mathit{expr}_{f}\ \mathit{exp}_{arg} \\
& | & \mathit{term} \\
& | & \mathit{lit}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{term} & ::= & \mathit{impl}_{t} \\
& | & \mathit{term}_{s} \rightarrow \mathit{term}_{t} \\
& | & \mathit{term}_{s \rightarrow t}\ \mathit{term}_{s}
\end{array}
$$

Figure 5.3: Grammar of *Loop* language.

Semantically, the *Loop* language does not presume a fixed order of evaluating `let` bound variables. However, all statements affecting control flow (`goto`, `unless`, `if`) are executed in the order they appear in the block.

```
toPercentages :: Array Double → Array Double
toPercentages fractions
  = let significant = filter (≥. 0.01) fractions
        percents    = map (* 100) significant
    in  percents
```

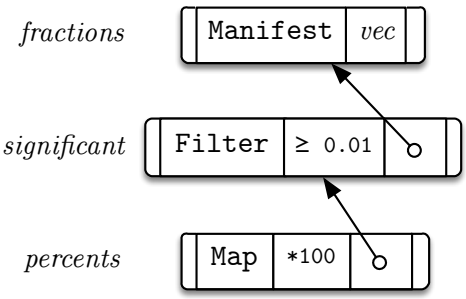


Figure 5.4: `toPercentages` function (left) and its *LiveFusion* AST (right).

5.3 Fused loops generation

In the previous section we have seen the first example of the *Loop* EDSL which computed a `filter` of an array followed by a `map`. We have also identified five main *sections* of a loop, represented by *basic blocks* in the *Loop* language: `init`, `guard`, `body`, `yield`, `bottom` and `done`.

We shall revisit the example from the previous section and introduce the process by which a *Loop* is generated.

This time we extend the example with specific functions passed to `map` and `filter` combinators. The function `toPercentages` shown on Figure 5.4 (left) converts an array of fractions to their percentage equivalents, filtering out those below 0.01 (or 1%).

5.3.1 Start with an AST

The `toPercentages` function internally uses a pipeline of two combinators: the output array of `filter` (called `significant`) becomes the input array of `map`. Recalling that `Array Double` is the type synonym for `AST (Vector Double)`, each combinator constructs a node in an AST representing a *delayed array computation*.

It is not known statically if the input array, `fractions`, is computed by a pipeline of combinators or whether it is a `Manifest` array stored in memory. Likewise, it is not known before runtime if the resulting `percents` array is consumed by another combinator and the pipeline of delayed operation will continue to grow after `toPercentages` function returns.

For the purposes of illustrating a complete running example we will assume that the `toPercentages` function is called with a `Manifest` array as argument and the result is immediately *forced* to be computed (e.g. to be written out to a file or to be consumed in

a random access fashion).

In this case the AST shown on Figure 5.4 (right) is constructed and *forced* to a `Manifest` array at the runtime of the program:

```
-- AST that may be a result of a call to toPercentages
ast :: AST (Vector Double)
ast = force
    $ Map (* 100)
    $ Filter (≥. 0.01)
    $ Manifest vec

-- LiveFusion library functions
force :: Elt a ⇒ AST (Vector a) → AST (Vector a)
force = Manifest . evalAST

evalAST :: AST a → a
evalAST = ...  -- compile AST and compute result
```

The three fusible combinators represented by `Manifest`, `Filter` and `Map` AST nodes are followed by the call to an evaluator, which processes them individually as shown next.

5.3.2 Generate loops for individual combinators

In order to generate the *Loop* language code for an AST of combinators, the evaluator first processes combinators individually. For each combinator it populates the sections of the loop that were identified in Section 5.1. It does so by inserting new statements into the appropriate basic blocks of a loop template.

The left of Figure 5.5 shows the loops generated for the individual combinators. The figure is implicitly referenced in the following discussion.

All variables and labels introduced by a particular combinator are given a unique identifier. In listings these are: *mfst* (short for *Manifest*), *map* or *filt*.

Internally however, the library uses unique integers to distinguish between similarly named variables and labels belonging to different combinators (e.g. `body_2` and `elt_3`). This not only avoids variable name clashes, but as we discuss next, enables communication among combinators through naming conventions.

5.3.2.1 Iterate over manifest array

Recalling that the `Manifest` combinator is the only combinator in the pipeline that holds a reference to the physical array, its loop does nothing more than to read the array $\texttt{arr}_{mfst}$

```
──────── Manifest ────────
init_mfst:
  let len_mfst = arrayLength arr_mfst

body_mfst:
  let elt_mfst = readArray arr_mfst ix_mfst

──────── Filter ────────
init_filt:
  let len_filt = len_mfst

body_filt:
  let elt_filt = elt_mfst
  unless elt_mfst ≥ 0.01 | bottom_map

──────── Map ────────
init_map:
  let len_map = len_filt

body_map:
  let elt_map = elt_filt * 100

──────── Writing result array ────────
init_map:
  let arr_map = newArray len_map

yield_map:
  writeArray arr_map ix_filt elt_map

done_map:
  let result_map = sliceArray arr_map ix_filt
  return result_map

──────── Looping ────────
init_map/filt/mfst:
  let ix_filt = 0
  let ix_mfst = 0

guard_map/filt/mfst:
  unless ix_mfst < len_mfst | done_map

yield_mfst:
  ix_mfst := ix_mfst + 1

yield_map/filt:
  ix_filt := ix_filt + 1
```

```
init_map/filt/mfst:
  let len_mfst = arrayLength arr_mfst
  let len_filt = len_mfst
  let len_map = len_filt
  let ix_filt = 0
  let ix_mfst = 0
  let arr_map = newArray len_map
  goto guard_map

guard_map/filt/mfst:
  unless ix_mfst < len_mfst | done_map
  goto body_mfst

body_mfst:
  let elt_mfst = readArray arr_mfst ix_mfst
  goto yield_mfst

yield_mfst:
  ix_mfst := ix_mfst + 1
  goto body_map

body_map/filt:
  let elt_filt = elt_mfst
  let elt_map = elt_filt * 100
  unless elt_mfst ≥ 0.01 | bottom_map
  goto yield_map

yield_map/filt:
  ix_filt := ix_filt + 1
  writeArray arr_map ix_filt elt_map
  goto bottom_map

bottom_map/filt:
  goto bottom_mfst

bottom_mfst:
  goto guard_map

done_map/filt/mfst:
  let result = sliceArray arr_map ix_filt
  return result
```

Figure 5.5: Loops generated for `toPercentages` function. Loops for individual combinators (left) and their merged equivalent (right).

element by element.

It is worth noting that the array is not `let`-bound in any of the blocks. This is due to the fact that the array is an *argument* to the loop and is passed from the running program when the computation is finally ready to be performed (code generation and loading are discussed in Chapter 6).

The combinator introduces a length variable $\texttt{len}_{mfst}$ which it binds in loop initialisation block $\texttt{init}_{mfst}$.

The result of array read is placed in $\texttt{elt}_{mfst}$ variable.

Notably, the combinator does not bind the index variable $\texttt{ix}_{mfst}$ itself but assumes that it's present in scope. The insertion of appropriate index variables is done after the complete loop is analysed and all *rates* are established.

5.3.2.2 Filter

In the partial loop representing the `filter` combinator, all bound variables and label names are identified by *filt*.

As discussed previously in Section 4.4 it is essential for performance that the user functions parametrising combinators be inlined in the generated code. In case of the `filter` the predicate function (≥ 0.01) has been inlined as the predicate expression to the `unless` statement.

While the length of output array of `filter` may be different from the length of input, the *upper bound* on the length is the same as that of the previous combinator. This is expressed by binding the length variable $\texttt{len}_{filt}$ to $\texttt{len}_{mfst}$ of the `Manifest` combinator.

Similarly, the resulting element $\texttt{elt}_{filt}$ is bound to be $\texttt{elt}_{mfst}$ – the output of `Manifest` in the same iteration of the loop.

5.3.2.3 Map

Populating the loop with `map`-specific statements is straight-forward.

Since the length of the output of `map` is always the same as the length of input, the $\texttt{len}_{map}$ variable is simply rebound. This also means that `Manifest`, `filter` and `map` share the same *upper bound* on the length of their output arrays.

To compute $\texttt{elt}_{map}$ for the current iteration, we only need to know $\texttt{elt}_{filt}$, that is the element produced by the previous combinator *in the current iteration.*

Again, the user specified function (∗100) is inlined to facilitate generation of fast code.

5.3.2.4 Physical array creation

For the `toPercentages` example we said the pipeline of combinators will be forced to a `Manifest` array immediately after the `map`.

The statements required to allocate, populate, slice and return the new array are self-explanatory.

It is worthy of note that the design decision to transfer control to `yield` basic block only when an element is produced has made it very easy to introduce the array write.

5.3.3 Rates and looping

Multiple times throughout this thesis, the concept of **rates** has been mentioned.

Rate is a property of a combinator within a combinator graph. If two combinators are statically known to produce arrays of the same length, they are said to have the same **rate**.

For example, all three `map`s in `map h . map g . map f` are known to share the same rate since their inputs and outputs will be of the same length.

For the same reason the `filter` and the subsequent `map` of the `toPercentages` example share the same rate since they both produce same-length arrays. As such, they are assigned a single index variable $\texttt{ix}_{filt}$ (Figure 5.5).

However, the `filter` combinator has earlier been identified as *rate-changing*. In particular we treat `filter` as a combinator producing elements at a **subrate** of the previous combinators. In the `toPercentages` example, the `filter` was preceded by `Manifest`, which receives it's own separate index variable $\texttt{ix}_{mfst}$.

If the rates of combinators are equal they share one index variable and their corresponding blocks can be merged together.
If a combinator runs at a *subrate* of another combinator then it has its own `body`, `yield` and `bottom` blocks since it may not produce an element at every iteration.

Rate equality and subrate relationship are not the only possible relationships between

two combinators. We will introduce other relationships when we discuss segmented combinators in Section 5.7.

5.3.4 Naming conventions

As seen from the loop code generated for the individual combinators in the `toPercentages` example, the *Loop* language uses a set of *naming conventions* to facilitate communication between consecutive combinators in a pipeline. In particular the following are always true for a combinator with identifier *id*:

- Variable $\texttt{elt}_{id}$ contains the value of produced element at every iteration
- Variable $\texttt{len}_{id}$ contains the upper bound on the length of the output array
- Variable $\texttt{ix}_{id}$ contains the index of the element being produced.

Binding $\texttt{elt}_{id}$ and $\texttt{len}_{id}$ in every combinator ensures that this information is *propagated* upwards through the pipeline of combinators. In particular, the `filter` only required to know the *id* of `Manifest` combinator below it to infer these variables.

By design the combinators know their own unique identifiers and the unique identifiers of the combinators they are referencing and nothing else.

5.3.5 Merging loops

We have given the *Loop* language representations for `Manifest`, `map` and `filter` combinators as well as the code that writes out physical arrays.

In order a create a succinct loop, these individual loops are merged into one loop.

For combinators as simple as `map` and `filter` merging loops means to simply merge the statements of the corresponding blocks blocks in no particular order.

As seen from the final loop on Figure 5.5 (right), not only the statements of same rate blocks have been merged together, but also their labels: each block can now be identified by one more of the three identifiers (e.g. $\texttt{body}_{\textit{filt/map}}$ can be identified by either *filt* or *map* identifier).

This is done in order to support fusing complex ASTs with many shared nodes. The benefit of this may appear much clearer when we introduce nested combinators in Section 5.7.

We will now briefly discuss the mechanism by which this is done.

```
-- * Map type
data AliasMap k a = AliasMap [(Set k, a)]
-- * Operators
(!)         :: AliasMap k a → k → a
-- * Query
lookup      :: k → AliasMap k a → Maybe a
lookup'     :: k → AliasMap k a → Maybe (Set k, a)
lookupAny'  :: Set k → AliasMap k a → Maybe (Set k, a)
synonyms    :: k → AliasMap k a → Set k
-- * Insertion
insert      :: k → a → AliasMap k a → AliasMap k a
addSynonym  :: k → k → AliasMap k a → AliasMap k a
-- * Combine
union          :: AliasMap k a → AliasMap k a → AliasMap k a
unionWithKeys  :: ...
mergeWithKeys  :: ...
-- * Traversal
map         :: (a → b) → AliasMap k a → AliasMap k b
-- * Conversion
elems       :: AliasMap k a → [a]
keys        :: AliasMap k a → [Set k]
assocs      :: AliasMap k a → [(Set k, a)]
```

Listing 5.1: *AliasMap* interface (partial). `Ord k` constraints omitted for brevity.

5.3.5.1 Managing label sets with *AliasMap*

In order to achieve the flexibility of having multiple labels associated with the same block I have implemented a new type of associations map called *AliasMap*.

It offers a familiar interface (Listing 5.1) similar to that of *Haskell*'s `Data.Map`. The difference is that it allows for a *set* of keys to be associated with one value. In addition to the usual functions found in `Data.Map` one can query the map using *synonyms*. That is, a value can be retrieved from the map given any key associated with that value.

The notion of *key synonyms* is used throughout *AliasMap* interface which greatly facilitates managing and merging loops where blocks are associated with multiple labels.

5.4 More on `gotos`

While the use of `gotos` is considered bad practice in modern software engineering I note several reasons for using them in the *Loop* intermediate language:

- It is completely hidden from the library user.

  Given a well behaving *Loop* code generator the produced code will always be valid if the user program is valid.

  This is akin to using `unsafePerformIO` and the like within the library internals for performance reasons. They can lead to bad code but with careful use result in very noticeable performance gains in purely functional programs.

- The *Loop* language was designed with a pluggable backend in mind.

  It was assumed that assembly-like *Loop* language would be easier to connect with any backend.

  Specifically in the *Haskell* backend, `goto` statements are translated to tail-recursive function calls.

  An *LLVM* backend is also planned. *LLVM* uses a very similar notion of *basic blocks*.

- The `goto` based design was the most flexible and the easiest to implement.

  Nonetheless, this does not prevent the *Loop* language from being extended to support a safer programming model.

5.5 Flat array combinators

5.5.1 Reductions

We have already looked at how the indices maintain the state of the loop between iterations. Although it is probably the most obvious state a loop has, it is not the only one. Perhaps the most prominent standard list combinators that pass partial results to recursive calls as accumulators are `fold`s and `scan`s. *LiveFusion* also offers these combinators.

In practice, binding and using accumulator variables like this in the *Loop* language is straight-forward and is no different from binding and using index variables. They are both treated as mutable variables in the *Loop* language.

Suppose, we wanted to translate `scan (*) 0 xs` to a loop. As discussed previously in Section 5.3.4 on naming conventions the generation of loop blocks for `scan` proceeds in assumption that variables $\texttt{len}_{xs}$ and $\texttt{elt}_{xs}$ are in scope.

The following is the portion of the loop generated for `scan` combinator:

```
init_scan/xs:
  let len_scan = len_xs
  let z_scan = 1
  let acc_scan = z_scan

body_scan/xs:
  let elt_scan = acc_scan

bottom_scan/xs:
  acc_scan := acc_scan * elt_xs
```

Generating loop code for `fold` is very similar except no element $\texttt{elt}_{fold}$ is produced in each iteration and the final result is stored in $\texttt{acc}_{fold}$.

5.5.2 Zipping

So far the combinator pipelines we have looked at had a list like structure. Each combinator would consume exactly one array and produce another. However, for many programs this is not sufficient. Many combinators take multiple arrays as input.

In general there is no constraint on how a combinator would consume each of those arrays. Some combinators (e.g. `backpermute`) require that one of the argument arrays can be accessed randomly. In other cases, combinators like `append` and `interleave` consume

arrays independently (not in a lock step).

However, there are combinators like `zipWith`N which consume N arrays in lock step.

A call to `zipWith (*) xs ys` element-wise multiplies `xs` and `ys`. Those two arrays may internally be pipelines of combinators. It results in the following *Loop* code:

```
initzip/xs/ys:
  let lenzip = lenxs

bodyzip/xs/ys:
  let eltzip = eltxs * eltys
```

The code is similar to that of `map` (Figure 5.5). In fact `zipWith`N combinators can be viewed as generalised `map`s.

There are two problems with `zipWith`N family of combinators:

1. The lengths of `xs` and `ys` may be different.

2. The loops for `xs` and `ys` may not produce elements in every iteration.

Appendix B considers both problems in turn, offering potential fusion strategies for each. While these may be legitimate cases to consider in a general purpose fusion framework, they do not appear in the vectorised *Data Parallel Haskell* code. Since *DPH* is currently the primary target for the application of *LiveFusion*, the fusion of these cases is left as future work.

```
1  init_{xs/r}:
2    let len_{xs} = arrayLength arr_{xs}
3    let ix_r = 0
4    goto guard_r
5
6  guard_{xs/r}:
7    unless ix_r < len_{xs} | done_r
8    goto body_r
9
10 body_{xs/r}:
11   let elt_{xs} = readArray arr_{xs} ix_r
12   goto yield_r
13
14 yield_{xs/r}:
15   ix_r := ix_r + 1
16   goto bottom_r
```

Figure 5.6: Sequential indexing in the *Loop* language.

5.6 Random access combinators

The combinators discussed so far consumed their inputs in a sequential manner from beginning to end. However, there are cases (in *DPH* inclusively) where elements need to be picked from the array at random. In particular, the `backpermute` combinator (`bpermute` for short) creates a new array by picking elements at given indices:

```
bpermute :: Array a → Array Int → Array a

bpermute [0,10,20,30,40,50] [3,4,5,1]
> [30,40,50,10]
```

To understand the translation of random access combinators to *Loop* language representation we must first recall how sequential looping is done. Whenever a sequential rate r is established, the looping statements are inserted into the loop (Figure 5.6). The corresponding index variable $\texttt{ix}_r$ is initialised (line 3) and updated (line 15) and the looping range is established (line 7).

However, the statement reading the `xs` array is oblivious to how the index variable was computed. It will use any value $\texttt{ix}_r$ currently has. In order to read variables at random, the `xs` array the rate r is no longer considered to be sequential and no index initialisation/update statements are inserted. Instead, the `backpermute` combinator will set the $\texttt{ix}_r$ variable to the appropriate value to read `xs` at the required index.

```
1 body_{bperm/is}:
2   let elt_{is} = ···                    -- produce index
3   let ix_r = elt_{is}                    -- set index to read xs at
4   let elt_{xs} = readArray arr_{xs} ix_r  -- read xs array
5   let elt_{bperm} = elt_{xs}             -- set result element
```

Figure 5.7: *Loop* code for reading `xs` array at index $\texttt{ix}_r$ set by `backpermute` combinator. where both data (`xs`) and indices (`is`) arrays are manifest and the result array is forced. Complete code is given in Appendix C.1.

The code presented in Figure 5.7 has been simplified to only include the statements relevant to reading `xs` array at the index read from `is` array. The complete code is presented in Appendix C.1.

Note that the array read on line 4 has been generated by the combinator responsible for `xs` array and is oblivious to the fact that `xs` would be read at random. Technically, `xs` does not need to be a manifest array! Any pipeline of combinators that can produce an element given an index will suffice. As such, the following expression will also fuse:

```
bpermute (map (+1) [0,10,20,30,40,50]) [3,4,5,1]
> [31,41,51,11]
```

The *rate system* of the *Loop* language makes it straightforward to switch between sequential and random accessing. Statements responsible for sequential looping (index initialisation and updates) are not inserted into the loop until the very end. This makes it possible for individual combinators to override that behaviour and take on the responsibility of setting the appropriate indices.

5.7 Segmented array combinators

The process of vectorisation in *Data Parallel Haskell* and in particular its *Flattening transform* (Section 3.2.1) mandates the need for combinators that operate on segmented arrays in the backend library. They have been discussed in an earlier chapter in Section 3.6.

Segmented arrays is a representation of nested arrays where all of the data is stored in a single flat array and a separate segment descriptor defines the partitioning of the data array into subarrays, called segments.

It was said that segmented combinators fall into two categories:

1. Combinators that *produce* elements at the *rate* of the segment descriptor, i.e. one element per segment (Figure 5.9).
2. Combinators that *produce* elements at the *rate* of data, i.e. producing entire segments (Figure 5.11).

The discussion of the combinators proceeds with reference to the two combinator categories.

5.7.1 Nested loops

Conceptually segmented combinators resemble nested loops in procedural languages where for each element taken from the segment descriptor array, that many elements of the data array get processed.

So far in this chapter we have been looking at flat loops. In the *Loop* language they have the standard structure and control flow presented in Figure 5.8 (left).

In order to support nested loops the *Loop* language offers a special basic block – `nest`. If a loop contains a nested loop, the `nest` basic block transfers control to it. After the nested loop is finished, the control is returned to the `body` block of the outer loop.

The default control flow for all nested combinators is shown on Figure 5.8 (right).

The following sections introduce the translations of the most prominent segmented combinators to the *Loop* language.

5.7.2 Segmented reductions

Segmented reductions represent the first type of segmented combinators outlined on page 65. They reduce each data segment to a single scalar value.

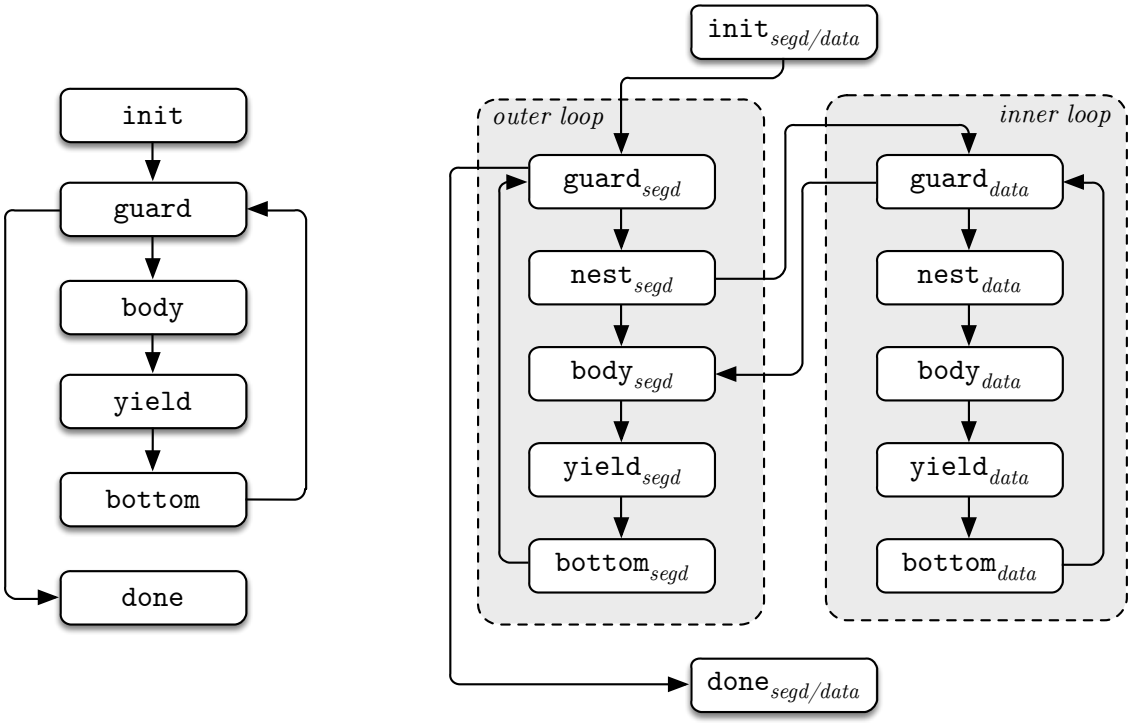


Figure 5.8: Default control flow between basic blocks in flat loops (left) and nested loops (right).

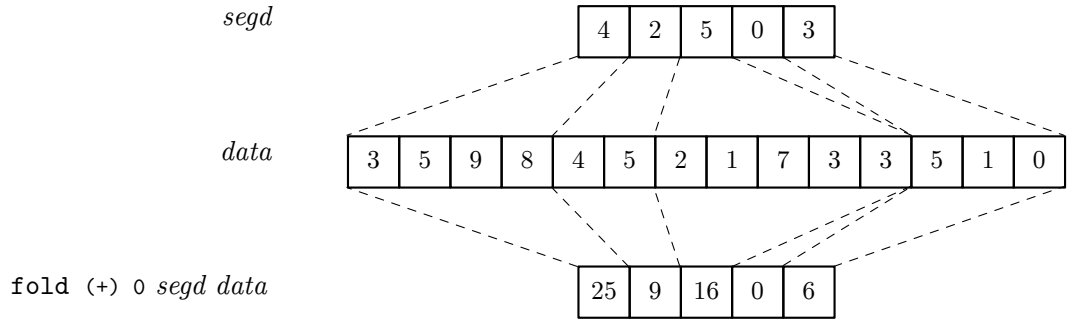


Figure 5.9: Segmented fold.

An example of `fold_s` application is shown in Figure 5.9.

Recalling the *Loop* language conventions, a single element $\texttt{elt}_{fold}$ should be produced for every iteration of the *outer* loop. Incorporating the new `nest` block into the loop, the call to a `fold_s` results in the *Loop* in Figure 5.10.

The code presented here has been simplified to only include statements generated by the call to `fold_s`. In particular the details of $\texttt{elt}_{segd}$ and $\texttt{elt}_{data}$ have been omitted. The complete code is presented in Appendix C.2.

Notably, `fold_s` can fuse with both the *segment descriptor* and the *data* loops. Additionally, it will fuse into its consumer. In fact, to any combinator consuming the output

of `fold_s` it will be indistinguishable from a flat combinator.

Other combinators that run at the rate of the segment descriptor include `count_s`, `sum_s` and several others. However, they are all implemented in terms of `fold_s`:

```
-- Count the number of occurences of a value in each segment
count_s :: Term a → Array Int → Array a → Array Int
count_s x segd = fold_s (+) 0 segd
               . map boolToInt
               . map (==. x)
  where boolToInt b = if b ==. true
                      then 1
                      else 0

sum_s :: Num a ⇒ Array Int → Array a → Array a
sum_s = fold_s (+) 0
```

5.7.3 Segmented data-rate combinators

In the previous section we have covered segmented reductions that produce arrays of the size of the segment descriptor. A different type of segmented operations are those that produce a data array containing segments of specified lengths.

An example of `scan_s` application is shown in Figure 5.11.

Following the *Loop* language conventions every iteration of the *inner* loop must produce an element $\texttt{elt}_{scan}$ of the result array. The relevant part of the *Loop* code are shown in Figure 5.12.

As before, the code presented has been simplified to only include statements generated by the call to `scan_s`. The complete code is available in Appendix C.3.

As in the case with `fold_s` and all other segmented combinators, `scanl_s` can fuse with both the *segment descriptor* and the *data* loops as well as its consumer.

5.7.4 Segmented generators

Segmented generators correspond to the category of combinators that *produce* elements at the rate of data. They differ from combinators like `scanl_s` discussed previously in that they do not consume a data array.

They are similar to their scalar counterparts in that they produce arrays from scalar values, only that they do it for an array of such scalar values, resulting in a segmented array:

```
init_fold/segd/data:
  ...
  let z_fold = 0
  goto guard_fold
```

Outer loop

```
guard_fold/segd:
  ...
  goto nest_fold

nest_fold/segd:
  let elt_segd = ···
  -- reset accumulator
  let acc_fold = z_fold
  -- find segment boundary
  let end_segd = ix_data + elt_segd
  -- enter inner loop
  goto guard_data

body_fold/segd:
  -- the fold value
  let elt_fold = acc_fold
  goto yield_fold

yield_fold/segd:
  ...
  goto bottom_fold

bottom_fold/segd:
  goto guard_fold

done_fold/segd/data:
  ...
```

Inner loop

```
guard_data:
  -- exit inner loop at the end of
segment
  unless ix_data < end_segd | body_fold
  goto nest_data

nest_data:
  goto body_data

body_data:
  let elt_data = ···
  goto yield_data

yield_data:
  ...
  goto bottom_data

bottom_data:
  -- update accumulator
  acc_fold := acc_fold + elt_data
  goto guard_data
```

Figure 5.10: *Loop* code for `folds (+) 0 segd data`. Complete code is given in Appendix C.2.

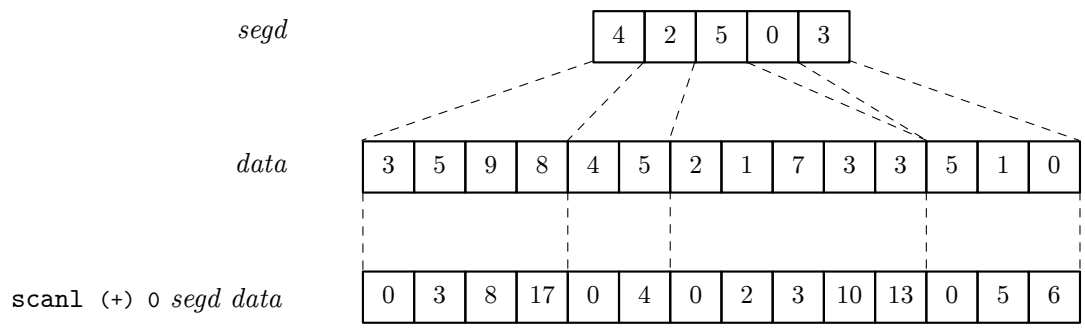


Figure 5.11: Segmented left scan.

```
init_scan/segd/data:
  ...
  let z_scan = 0
  goto guard_segd

──────── Outer loop ────────
guard_segd:
  ...
  goto nest_segd

nest_segd:
  let elt_segd = ···
  -- reset accumulator
  let acc_scan = z_scan
  -- find segment boundary
  let end_segd = ix_data + elt_segd
  -- enter inner loop
  goto guard_scan

body_segd:
  goto yield_segd

yield_segd:
  ...
  goto bottom_segd

bottom_segd:
  goto guard_segd

done_scan/segd/data:
  ...
```

```
──────── Inner loop ────────
guard_scan/data:
  -- exit inner loop at the end of
segment
  unless ix_data < end_segd | body_segd
  goto nest_scan

nest_scan/data:
  goto body_scan

body_scan/data:
  let elt_data = ···
  -- result element
  let elt_scan = acc_scan
  goto yield_scan

yield_scan/data:
  ...
  goto bottom_scan

bottom_scan/data:
  -- update accumulator
  acc_scan := acc_scan + elt_data
  goto guard_scan
```

Figure 5.12: *Loop* code for `scanls (+) 0 segd data`. Complete code is given in Appendix C.3.

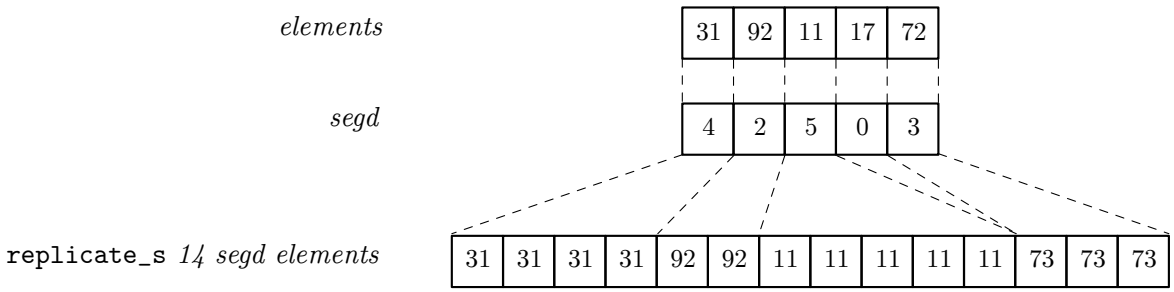


Figure 5.13: Segmented replicate.

```
replicate   :: Int        -- count
            → a           -- element
            → Array a

replicate_s :: Int        -- result length
            → Array Int   -- segment descriptor
            → Array a     -- elements
            → Array a

enumFromStepLen   :: Int        -- start
                  → Int        -- step
                  → Int        -- length
                  → Array a

enumFromStepLen_s :: Int        -- result length
                  → Array Int   -- starts
                  → Array Int   -- steps
                  → Array Int   -- lengths (segment descriptor)
                  → Array a
```

Most of these combinators are also *point-wise consumers* (Sections 5.5.2 and 3.6.1.1). They consume values from multiple arrays in lockstep in order to generate an array segment in the inner loop. An example of `replicate_s` application is shown in Figure 5.13.

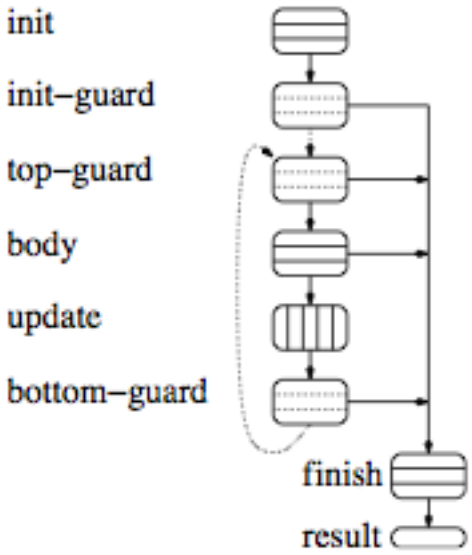


Figure 5.14: The loop template used in Shivers' work on *LISP* loops. Dotted lines indicate permutable sequences. Horizontal lines denote sequential statements, vertical lines – independent statements.

5.8 Related work

5.8.1 Shivers' Anatomy of the Loop

Shivers' work on loop anatomy in *LISP* [47] has served as the primary motivation for designing the structure for the loops described in this chapter. Besides striving for a more modular structure, the work was intended to ensure the correct scoping of variables introduced by loop generation macros. This problem is solved by *LiveFusion*'s *Liveness Analysis* introduced in the next chapter.

Shivers identifies eight sections shown in Figure 5.14. *LiveFusion*'s seven sections (`init`, `guard`, `nest`, `body`, `yield`, `bottom`, `done`) reflect its intended use as an intermediate language for fusing loops generated by *DPH* combinators. While Shivers' loops may be more flexible in how they can be constructed by the library user, *LiveFusion*'s rate system allows to compose loops that were generated independently of each other.

5.8.2 Waters' Series Expressions

While identifying loop anatomy provided a convenient way to fuse combinators to a common structure, Waters' work on *Series Expressions* [55] identifies criteria for such combinators to be fusible. Series expressions are akin to combinators and can be directly compared. He presents the four main restrictions on the series expressions:

1. The *online* criteria demands that every series expression in the graph must consume and produce elements in a lockstep.

   This restriction means that filtering and nested loops are not supported by Series Expressions. Because of the nature of *DPH* programs, this would result in severe performance penalties. The restriction is lifted in *LiveFusion* through the use of a rate system which introduces the notions of equal and nested rates as well as subrates.

2. Fusible series expressions must not be subjected to conditional control flow as in example in 5.2.

   This largely demands that the combinator graph contains no control flow at the time it is optimised. This restriction is important when the optimisations take place at compile time and the data flow is not statically known. However, in the case of *LiveFusion* all control flow takes place as AST is being constructed thus lifting the restriction.

3. The program is statically analysable.

   Again, this restriction is more concerned with fusion implemented by the compiler. In *LiveFusion* an AST of higher order combinators specialised by user functions expressed in a scalar EDSL (Section 4.4) allow for complete graph analysis.

4. Series are not consumed in a random access manner.

   This restriction is largely present in *LiveFusion*. Even though *LiveFusion* supports random accessing of arrays, these array are required to be manifest.

```
halfdouble flag xs ys = scan (+) 0
                      $ if flag then (map (* 2) xs)
                                else (map (/ 2) ys))
```

Listing 5.2: Control flow between combinators.

5.8.3 Flow Fusion

The work most closely related to *LiveFusion* is *Flow Fusion* framework [31] for *Haskell* originally also designed with *DPH* in mind. It extends Waters' work on Series Expressions with the notion of rates (which is where *LiveFusion* borrows the term from). It was

discussed in some detail in Section 4.5.3. One particular advantage of *Flow Fusion* is its ability to use rate information to cluster combinators into fusible groups optimising for memory traffic [16]. The need for this arises from the fact that for sufficiently advanced examples there is more that one way to cluster combinators in a graph. The system uses Integer Linear Programming to determine clustering which is a non-polynomial problem in the general case.

Its applicability to graphs with a large number of combinators *at runtime* remains an open question since running an ILP solver may outweigh performance gains achieved. None-the-less, clustering of graphs with heavy branching and many non-fusible edges remains an important research question *LiveFusion* is facing.

As mentioned previously, *Flow Fusion* cannot currently fuse segmented combinators although this is not a fundamental limitation and it is likely to be lifted in the future.

5.8.4 Correctness of resulting loops

The code for a single loop in the *Loop* language is generated by multiple combinators independently of each other. Because of the interleaving of variable binders, their use and the control flow, one must be careful to ensure that all the right variables are in scope when they are referenced.

The recent work on "Combinators for impure yet hygienic code generation" [24] may prove useful to avoid these problems statically.

Alternatively, *Flow Fusion* only allows binding variables before the loop begins ensuring that all variables will be available at all times.

Chapter 6

Imperative code generation

The previous chapter introduced the process of translating a graph of delayed array combinators to an assembly-like EDSL for expressing array computations.

We first identified a generic loop structure consisting of `init`, `guard`, `body`, `yield`, `bottom` and `done` section represented by *basic blocks* in the *Loop* language. A `nest` basic block was introduces later to support nested loops, needed by segmented combinators. These loop sections ultimately constituted a skeleton which every combinator would populate with relevant *statements* that would eventually be merged into a single loop.

We then looked at the translation of a number of flat and segmented array combinators into the *Loop* language and showed how the partial loops would be merged together into "runnable" loops.

While the basic blocks of the loops did contain imperative style statements with some assembly-style labels attached to them, it was never said just what makes a loop "runnable".

In this chapter we explore a method by which *Loop* language programs can be turned into runnable code. In practice the *Loop* language can be interpreted by multiple backends. I present *Loop*'s translation to one possible backend which is currently available. This backend is targeting (perhaps unimaginatively) *Haskell* source language, which means that the code generated is *Haskell*.

6.1 Approaches to looping

In the surface *LiveFusion* language loops are not explicit. However, most of the array combinators eventually form part of loops expressed in the internal *Loop* language. We have discussed this language at length in the previous chapter.

When it comes to expressing loops there is a fundamental difference between the vast majority of general purpose languages (even functional ones) and the purely functional languages like *Haskell*. In procedural languages the loop statements are usually built into the language. Further down, at the machine level, these loops become a collection of instructions with labels/addresses and `jump` instructions to transfer control between parts of the loop.

The situation is very different in *Haskell* however. The language does not offer an explicit construct for loops. All looping is expressed in the form of recursion.

If we are to generate *Haskell* code that loops over arrays we need a way to express the loops we outlined in the previous chapter as recursive functions in the generated code.

6.2 Fast mutable arrays in *Haskell*

The principle data structure that *LiveFusion* is ultimately working with is an array. To achieve high performance, the generated code must be able to efficiently loop over arrays in memory which includes both reading an writing.

We have established that looping in *Haskell* is done using recursive functions and this section discusses how this can be applied to array computations.

The internal *Loop* language has `writeArray` statement which hints the need for *mutable* arrays. The use of mutable values is not typical in *Haskell* and is discouraged. However, for performance reasons there are mechanisms through which this can be done.

In *Haskell* all variable are immutable by default. To achieve mutability one must leave the pure context and use one that supports mutable state. This usually amounts to using `IO` or `ST` monads [29].

The `IO` monad lets the programmer alter the global state of the program and perform any desired side-effecting computations. On the other hand, `ST` monad only allows one to work with thread-local mutable memory. This memory is completely local to the monadic computation in that it is allocated, used and destroyed before the monad returns to the pure context.

The return value from the `ST` monad is a pure *Haskell* value. Thus, stateful computations performed inside the monad are not visible outside of the monadic context. The compiler ensures that no mutable state escapes the `ST` computation. This means that for all intents and purposes an `ST` computation can be considered pure, despite the fact that it may use side-effects internally (usually for efficiency reasons).

In *Haskell* mutable arrays can be used from both `IO` and `ST` monads. There are several implementations available providing similar interfaces to efficient mutable arrays. Specifically *LiveFusion* internally uses the *vector*[1] package.[2] Its use in the examples in this chapter should be quite self-explanatory. Since the *Loop* language does not require any side-effecting computations beyond allocating and writing into memory, arrays in `ST` monad will suffice.

[1] `http://hackage.haskell.org/package/vector`

[2] The choice of *vector* library is mostly due to historical reasons. It has been first implemented by Roman Leshchinskiy in the research group I am part of. It provides array functionality to *Data Parallel Haskell* and *Repa* [25] both of which originate from this group.

6.3 Compiling *Loop* language to *Haskell* loops

In this section I present the translation of the *Loop* EDSL to *Haskell*. The grammar of *Loop* language is given once again in Figure 6.1.

A simple example to be used for illustration purposes is based on the `toPercetages` function thoroughly studied in the previous chapter (Section 5.3). The following expression multiplies every element of an array by 100:

```
map (* 100) xs
```

While the expression only consists of one combinator, `map`, the `xs` argument may be a fusible pipeline of combinators. Likewise the `map` itself may be fused into its consumer. However, for our purposes we assume that `xs` is a *manifest* array and that the result of `map` is immediately forced.

The generated *Loop* code for the example is shown in the right of Figure 6.2. Its translation by the *Haskell* backend is presented in the left of the same figure. For improved readability the code has been altered in the following ways:

1. Function applications β-reduced
2. Fully qualified names simplified
3. Operators moved to infix position
4. Unique integer suffixes replaced with more descriptive names
5. Explicit literal conversions omitted
6. Strictness annotations (`!`) omitted

```
let !elt_1 = (λa → λb → (GHC.Num.*) a b) elt_2 (GHC.Num.fromInteger 100)
           ⇓
let elt_map = elt_xs * 100
```

6.3.1 Blocks and `gotos`

A *Loop* is essentially a group of labelled basic blocks with a predefined entry block. Every basic block becomes its own function in the generated code. We see in the example that each block like `body`$_{map/xs}$ became `body`$_{map}$ function choosing only one label to represent the block. Each basic block has a unique identifier associated with it (in our case *map*),

[3]For improved readability the unique integers are replaced with meaningful names in the code listings. (Footnote for Figure 6.1)

$$
\begin{array}{lcl}
\mathit{name} & \rightarrow & (\texttt{arr}, \texttt{elt}, \texttt{f}, \texttt{acc}, \texttt{init}, \texttt{body}, \text{ etc.}) \\
\mathit{id} & \rightarrow & (\text{unique integer}^{3}) \\
\mathit{impl} & \rightarrow & (\text{backend specific code, e.g. } \textit{Template Haskell} \text{ expression}) \\
\mathit{lit} & \rightarrow & (\textit{Haskell} \text{ value of a supported type (Int, Float, Bool, tuple, etc..)} \\
\mathit{var} & ::= & \mathit{name}\ \mathit{id} \\
\mathit{label} & ::= & \mathit{name}\ \mathit{id}
\end{array}
$$

$$
\mathit{loop} \quad ::= \quad \overline{\mathit{block}}\ \overline{\mathit{var}_{arg}}\ \mathit{label}_{entry}
$$

$$
\mathit{block} \quad ::= \quad \overline{\mathit{label}}\ \overline{\mathit{stmt}}\ \mathit{stmt}_{final}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \texttt{let}\ \mathit{var}\ \texttt{=}\ \mathit{expr} \\
& | & \mathit{var}\ \texttt{:=}\ \mathit{expr} \\
& | & \texttt{if}\ \mathit{expr}_{bool}\ \texttt{|}\ \mathit{label}_{true}\ \mathit{label}_{false} \\
& | & \texttt{unless}\ \mathit{expr}_{bool}\ \texttt{|}\ \mathit{label} \\
& | & \texttt{goto}\ \mathit{label} \\
& | & \texttt{return}\ \overline{\mathit{var}} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{newArray}\ \mathit{expr}_{length} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{readArray}\ \mathit{var}_{array}\ \mathit{expr}_{index} \\
& | & \texttt{writeArray}\ \mathit{var}_{arr}\ \mathit{expr}_{index}\ \mathit{expr}_{element} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{sliceArray}\ \mathit{var}_{arr}\ \mathit{expr}_{new\_length} \\
& | & \texttt{let}\ \mathit{var}\ \texttt{=}\ \texttt{arrayLength}\ \mathit{var}_{arr}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{var} \\
& | & \mathit{expr}_{f}\ \mathit{exp}_{arg} \\
& | & \mathit{term} \\
& | & \mathit{lit}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{term} & ::= & \mathit{impl}_{t} \\
& | & \mathit{term}_{s} \rightarrow \mathit{term}_{t} \\
& | & \mathit{term}_{s \rightarrow t}\ \mathit{term}_{s}
\end{array}
$$

Figure 6.1: *Loop* language grammar.

```
entry :: [Dynamic] → Dynamic
entry [arr_xs] = toDyn (run (fromDyn arr_xs))

run :: Vector Double → Vector Double
run arr_xs = runST (init_map arr_xs)

init_map arr_xs = do
  let len_xs = arrayLength arr_xs
  let len_map = len_xs
  let ix_xs = 0
  arr_map ← newArray len_map
  guard_map arr_xs arr_map ix_xs len_xs

guard_map arr_xs arr_map ix_xs len_xs = do
  if ix_xs < len_xs
    then nest_map arr_xs arr_map ix_xs len_xs
    else done_map arr_xs arr_map ix_xs len_xs

nest_map arr_xs arr_map ix_xs len_xs = do
  body_map arr_xs arr_map ix_xs len_xs

body_map arr_xs arr_map ix_xs len_xs = do
  let elt_xs = readArray arr_xs ix_xs
  let elt_map = elt_xs * 100
  yield_map arr_xs arr_map ix_xs len_xs elt_map

yield_map arr_xs arr_map ix_xs len_xs elt_map = do
  let ix_xs' = ix_xs + 1
  writeArray arr_map ix_xs elt_map
  bottom_map arr_xs arr_map ix_xs' len_xs elt_map

bottom_map arr_xs arr_map ix_xs len_xs elt_map= do
  guard_map arr_xs arr_map ix_xs len_xs

done_map arr_xs arr_map ix_xs len_xs = do
  result_map ← sliceArray arr_map ix_xs
  return result_map
```

```
init_map/xs:
  let len_xs = arrayLength arr_xs
  let len_map = len_xs
  let ix_xs = 0
  let arr_map = newArray len_map
  goto guard_map

guard_map/xs:
  unless ix_xs < len_xs | done_map
  goto nest_map

nest_map/xs:
  goto body_map

body_map/xs:
  let elt_xs = readArray arr_xs ix_xs
  let elt_map = elt_xs * 100
  goto yield_map

yield_map/xs:
  ix_xs := ix_xs + 1
  writeArray arr_map ix_xs elt_map
  goto bottom_map

bottom_map/xs:
  goto guard_map

done_map/xs:
  let result_map = sliceArray arr_map ix_xs
  return result_map
```

Figure 6.2: *Haskell* code (left) and *Loop* code (right) generated for `map (* 100) xs`.

which distinguishes it from similarly named blocks of any nested loops which may be present.

A *block* becomes a *function* in the generated code. Consequently, the `goto` statements become *function calls*.

Blocks of the internal *Loop* language do not explicitly specify what loop state they reference or declare. A loop's state is implicit and global. In the generated code the state of the loop is passed as function arguments. This is the reason for a large number of arguments to functions generated from blocks. The process for generating the arguments from loop variables will be outlined in Section 6.4 on liveness analysis.

6.3.2 Conditionals

The *Loop* language offers two conditional statements: `if` and `unless`. Given a boolean expression the `if` statement transfers control to one of the two specified blocks. The `unless` is a simplified `if` statement which only transfers control to the specified block if the predicate is *false*. The `unless` is used to preempt execution of a block if a condition is not met.

Both `if` and `unless` are translated to *Haskell*'s `if` expression in the generated code as seen in the `guard` of the provided example.

6.3.3 Variable bindings and assignments

In the *Loop* language every local variable must be bound before use. The code generator checks to ensure that the control goes through the binding before the first use of the variable.

The bound variables can be mutable as well as immutable. In fact the *Loop* language does not currently make a distinction between the two in the way they are bound and used.

A variable *binding* appears as a *strict* `let` binding in the generated *Haskell* code. The exclamation marks, called *strictness annotations*, (as in `let !x = ...`; omitted from code examples) ensure that every bound variable is evaluated to normal form so no thunks are created during evaluation. In most cases this is not necessary since the *GHC*'s strictness analysis will find most variables to be strict anyway.

Whenever a new variable binding is encountered in a block of the loop it is assumed to be a fresh variable and it shadows any previous bindings of that variable in the environment. In practice this is useful for the cases where the block is entered multiple times throughout loop execution. In the studied example the `body` block binds variables $\texttt{elt}_{xs}$ and $\texttt{elt}_{map}$ in every iteration. They are subsequently read in `body` and `yield` but are not required afterwards. A simple liveness analysis described in section Section 6.4 allows us to only pass those variables from `body` to `yield` and disregard them everywhere else. As such those variables are bound anew in every iteration.

The situation with mutable variables is slightly more complex. For example, the loop index $\texttt{ix}_{xs}$ is bound like a regular variable in the `init` block. However, once it has been *assigned to* in `bottom`, it is given a fresh name $\texttt{ix}_{xs}\texttt{'}$ in the generated code. This is because variables in *Haskell* are immutable. While there are ways to have mutable variables in `ST` or `IO` monads it results in less efficient code.

During liveness analysis mutable variable are treated the same way as their immutable counterparts. Technically it is possible to have a mutable variable that lives during one loop iteration but is not carried over to the next iteration. However, the author is yet to come across a combinator that would require that.

6.3.4 Array manipulation

Array is the fundamental data structure the *Loop* language works with. Consequently the array primitives are built right into the language: `readArray`, `writeArray`, `arrayLength`, `newArray` and `sliceArray`. The implementation of the primitives is presented in Figure 6.3.

The implementation is straightforward and all of these array primitives can be seen used in the example generated code. One will notice, however, that there are two types of vectors in use: *immutable* `Vector`s and *mutable* `MVector`s.

Immutable vectors

The *immutable* vectors are the ones passed around as `Manifest` arrays in the surface *LiveFusion* language. They are pure Haskell values and can be read inside a pure function. They are the array values that are passed into the generated code from the host program and are returned as the result. The `arrayLength` and `readArray` statements of the *Loop*

```
import Data.Vector.Unboxed as V
import Data.Vector.Unboxed.Mutable as MV
import Control.Monad.ST

arrayLength :: V.Unbox a ⇒ V.Vector a → Int
arrayLength = V.length

readArray :: V.Unbox a ⇒ V.Vector a → Int → a
readArray = V.unsafeIndex

writeArray :: V.Unbox a ⇒ MV.MVector s a → Int → a → ST s ()
writeArray = MV.unsafeWrite

newArray :: V.Unbox a ⇒ Int → ST s (MV.MVector s a)
newArray = MV.new

sliceArray :: V.Unbox a ⇒ MV.MVector s a → Int → ST s (V.Vector a)
sliceArray vec len = V.unsafeFreeze $ MV.unsafeTake len vec
```

Figure 6.3: Array primitives implementation in *Haskell* backend.

,

language result in function calls of the same name in the generated code and operate on immutable `Vectors`.

Mutable vectors

The *mutable* vectors on the other hand are only used internally by the generated code to create and fill a new array which would represent the result of the loop. The mutability of the vector is indicated in its type and the type of functions that operate on it.

```
writeArray :: Unbox a
           ⇒ MVector s a
           → Int
           → a
           → ST s ()
```

In the above type of `writeArray`, both the type of `MVector` and the resulting `ST` computation are indexed by `s`. This ensures that stateful computations on the same MVector value are performed inside the same state thread and do not escape from the `ST` monad.

In order to return an array from the plugin, and thus outside the `ST` monad, it must first be converted to an immutable `Vector`. `sliceArray` is used for that as indicated by its type. The extra `Int` argument allows to take only a portion of the array and return that. This is useful in `filter`-like operations which may produce a shorter array than initially allocated.

6.3.5 Returning results

In all examples seen so far the the result of a loop computation was a single array variable ($\texttt{result}_{map}$ in the above example). However it doesn't need to be. Both the *Loop* language and the code generator can return multiple array and scalar values in nested tuples, e.g.:

```
let acc_fold = ...
result_scan ← sliceArray ix_r arr_scan
result_zip  ← sliceArray ix_r arr_zip
return ((result_scan, result_zip), acc_fold)
```

6.4 Liveness analysis and state passing

We have discussed that the individual basic blocks of the *Loop* language become `ST` functions in the generated code. Subsequently the statements that transfer control from one block to the other (`goto`, `unless`, `if`) become function calls.

Loop language is largely a procedural language with assignment statement (`:=`) that can mutate variables arbitrarily. However, since the target language *Haskell* has no mutable state by defaultall mutable variables of the *Loop* language must be translated to *Haskell* as immutable values.

How does one pass state as pure *Haskell* values? Each loop potentially has a lot of state that is written and read in different blocks of the loop. Since these blocks are functions it would be natural to assume that the state is passed in function arguments.

The problem, however, is that there is no fixed state alive in the loop at any given time. Referencing the original example in Figure 6.2:

- some intermediate variables like $\texttt{elt}_{xs}$ are only used inside the block they are defined in (`body`)
- some variables like $\texttt{elt}_{map}$ are bound in `body` and are later referenced in `yield` but are unused after that
- some variables are bound in `init` and are persistent for the whole duration of the loop (counters and accumulators).

In order to appropriately generate the argument lists to the functions representing loop blocks I have implemented a *liveness analysis* pass in the *LiveFusion* EDSL compiler.

Liveness analysis is not a backend feature

Liveness analysis operates on *Loop* EDSL specification of the program. It computes environment for a control flow graph (CFG) of *basic blocks* and operates on the notions of variable *bindings* (`let`) and *assignments* (`:=`). The outcome of liveness analysis is the information about the use of state in a CFG.

As such this pass is not specific to any specific backend and is a way to analyse a program given in *Loop* EDSL. Nonetheless, I believe that the necessity for liveness analysis has not been justified until now. With enough foreword, it is described below.

I will now offer an intuitive description for how the algorithm works.

Each loop has a single entry block. Starting with that block, and following the loop's control flow graph (CFG) we recursively determine an environment which holds three properties:

1. new variables *bound* in the block (using `let`)
2. variables that are *destructively updated* in the current block (using assignment `:=`)
3. variables otherwise *assumed* to be in *environment*:
   - either because they are referenced in the *current* block, or
   - they are *assumed* to be in the environment of any of the *successor* blocks, to which control may be transferred (i.e. variables referenced but not bound in those blocks)

Once the environment has been computed, it is used to generate function argument lists and the appropriate function calls for each `goto`, `unless` and `if` statement. Each of the block's *assumed* variables becomes the argument of the *Haskell* function corresponding to that block.

6.4.1 Discussion of liveness analysis

Several points to note about computing the environment:

1. Some blocks have multiple *successor* blocks to which control can be transferred (e.g. `guard` can go to `body` or `done`). The union of all *assumed* variables is taken to determine what need to be in the environment of the current block.

2. *Loop* CFGs are inherently cyclic (e.g. `bottom` block usually transfers control to `guard` to begin the new iteration). Cycle detection is required when exploring CFG.

3. *Loop* variables may not be required by the immediate *successor* basic blocks in the CFG. However, they will be in the set of *assumed* variables if they are required later on. The relationship is transitive.

4. The liveness analysis pass will catch any use of variables that have not been previously bound (resulting in a runtime error indicating a problem in the library, not in the client code).

There are also two caveats of the *Haskell* code generator in its current state:

- A variable can only be assigned once in any given block
- The new value won't be available until the control is transferred to a successor block

Both of the above are not fundamental limitations of either *Loop* EDSL or the *Haskell* backend. They arise from the fact that the liveness analysis works on *per-block* basis and not *per-statement*. This means that liveness analysis attempts to determine which variables are expected to be in scope in a particular block, which are updated and which are passed to successor blocks. However, it does not attempt to do the same for each statement.

So far neither of these limitations posed a problem since the loop structure itself is so modular. There is no apparent need in a more fine-grained per-statement liveness analysis. The generated *Haskell* code is run through *GHC* which itself has excellent liveness analysis and other compilation tactics resulting in efficient code. The planned *LLVM* backend is also likely to be able to optimise the code in its current form.

6.5 Interfacing host and plugin code

In the previous sections we have seen the process of generating *Haskell* code from *Loop* EDSL. The functions generated from basic blocks of the loop form the main part of the plugin that would later be compiled and loaded into the running host program.

This section describes an extra layer of functionality that enables the dynamic compilation and loading of the generated code.

6.5.1 Plugin side entry

Looking at the plugin code presented at the beginning of this chapter in Figure 6.2, the entry functions have the following type:

```
fromDyn :: Typeable a ⇒ Dynamic → a
fromDyn d = case fromDynamic d of
              Just v  → v
              Nothing → error "Argument␣type␣mismatch"

entry :: [Dynamic] → Dynamic
entry [arr_xs] = toDyn (run (fromDyn arr_xs))

run :: Vector Double → Vector Double
run arr_xs = runST (init_map arr_xs)
```

The `entry` function is the *dynamically* typed entry into the plugin. Its type is the same for all plugins generated by *LiveFusion*. The arguments such as manifest arrays are passed in as a list of `Dynamic` values[4] internally holding their type representation. This allows the plugin to coerce the argument to the appropriate type using `fromDyn` ensuring that the type is correct at runtime.

On the other hand, the `run` function is the typed entry into the plugin and is specific to every plugin generated.

In a more elaborate program with multiple arguments and return values the plugin may have entry functions such as the following:

[4] See `http://hackage.haskell.org/package/base/docs/Data-Dynamic.html`

```
entry :: [Dynamic] → Dynamic
entry [arr_1, arr_2, arr_3, len_4]
  = toDyn (run (fromDyn arr_1)
               (fromDyn arr_2)
               (fromDyn arr_3)
               (fromDyn len_4))

run :: Vector Double → Vector Double → Vector Int → Int
    → ((Vector Double, Vector Double), Vector Int)
run arr_1 arr_2 arr_3 len_4 = runST (init_5 arr_1 arr_2 arr_3 len_4)
```

Note that the return values are structured in a nested tuple. It resulted from the use of explicit tupling constructor `:*:` discussed in Section 4.3. For example:

```
three :: AST ((Vector Double, Vector Double), Vector Int)
three = let xs :: Array Double = ...
            ys :: Array Double = ...
            zs :: Array Int    = ...
        in  xs :*: ys :*: zs
```

6.5.2 Host side dynamic compilation and loading

The code for the host side dynamic compilation and loading is relatively straightforward and is presented in Listing 6.1. It uses *GHC API* for compilation, which means that the *GHC* is linked in as a library into every *LiveFusion* binary (provided the linking is static).

At the moment the compiled code is not cached for later reuse. This means that if the same combinator graph is executed multiple times, it is recompiled every time. The ability to amortise compilation costs over multiple recursive calls is important for *Data Parallel Haskell* programs as we will see in the next chapter and it is left as future work.

```
-- | Evaluates the AST to a final value.
evalAST :: Typeable t ⇒ AST t → t
evalAST ast = result
  where
    loop = getLoop ast
    dynResult = unsafePerformIO $ evalLoopIO loop (typeOf result)
    result = fromJust $ fromDynamic dynResult
{-# NOINLINE evalAST #-}

type Arg = Dynamic

evalLoopIO :: Loop → TypeRep → IO Arg
evalLoopIO loop resultTy = do
  (pluginPath, h, moduleName) ← openTempModuleFile
  let entryFnName  = defaultEntryFunctionName ++ moduleName
  let codeString   = pluginCode moduleName entryFnName loop resultTy
  dump codeString h
  pluginEntry ← compileAndLoad pluginPath moduleName entryFnName
  let args    = Map.elems $ loopArgs loop
  let result  = pluginEntry args
  return result

openTempModuleFile :: IO (FilePath, Handle, String)
openTempModuleFile = do
  (fp, h) ← openTempFile "/tmp/" "Plugin.hs"
  let moduleName = takeBaseName fp
  return (fp, h, moduleName)

dump :: String → Handle → IO ()
dump code h = hPutStrLn h code >> hClose h

compileAndLoad :: FilePath → String → String → IO ([Arg] → Arg)
compileAndLoad hsFilePath moduleName entryFnName =
    defaultErrorHandler defaultFatalMessager defaultFlushOut $ do
      runGhc (Just libdir) $ do
        dflags_def ← getSessionDynFlags
        let dflags_dyn = gopt_set dflags_def Opt_BuildDynamicToo
        let dflags_opt = dflags_dyn { optLevel = 2 }
        setSessionDynFlags dflags_opt
        target ← guessTarget hsFilePath Nothing
        setTargets [target]
        r ← load LoadAllTargets
        case r of
          Failed    → error "Compilation␣failed"
          Succeeded → do
            setContext [IIDecl $ simpleImportDecl (mkModuleName moduleName)]
            pluginEntry ← compileExpr (moduleName ++ "." ++ entryFnName)
            let pluginEntry' = unsafeCoerce pluginEntry :: [Arg] → Arg
            return pluginEntry'
```

Listing 6.1: Host side dynamic compilation and loading code

6.6 Related work

6.6.1 Alternative backends

This chapter has introduced the *Haskell* backend for *LiveFusion*. Dynamically compiling *Haskell* modules has become a popular way of extending *Haskell* applications with plugins [48, 36, 49]. These applications tend to have their plugins written by hand. This is contrary to the present work, where the plugin code is generated on the fly from the *Loop* language code.

However, *LiveFusion* is not limited to only one backend. In order to extend the framework to support other backends, a new interpreter for the *Loop* language can be built together with an appropriate implementation of the scalar language functions (Section 4.4).

One particularly attractive alternative is *LLVM* framework [28]. *GHC* already uses *LLVM* to produce highly efficient code. However, using a framework such as *LLVM* for compiling *LiveFusion* generated code may avoid the costs associated with running the generated code through the complete *GHC* compilation pipeline. As it will be seen in the next chapter, a compilation overhead of 600 ms is not uncommon when compiling the generated code with *GHC*. Reducing those costs may make *LiveFusion* more widely applicable to use cases where many pieces of code must be compiled independently and amortisation cannot be as easily exploited.

While *LiveFusion* is already producing optimised procedural style code, it still relies on *GHC* to perform some optimisations. In particular user defined scalar functions that parametrise higher-order combinators are inlined without further analysis in hopes that *GHC* will subsequently perform all the necessary optimisations on them. This may not be sufficient if *LLVM* code is generated directly and *LiveFusion* might need extra optimisation passes before the emitted code can target *LLVM*.

Direct *LLVM* code generation is becoming popular with EDSL for high performance computations. Both *Flow Fusion* [31] and *Accelerate* [34] are targeting it. The former first converts the AST to *DDC Core* [30], an intermediate compiler language similar to *GHC Core*. *LLVM* has also been successfully used for compiling a digital signal processing EDSL [52, 51].

Generating *Haskell* and *LLVM* are not the only options. [45] offers a comparative

study of *TaskGraph*, *Fabius* and *Tick C* system that are suitable for runtime generation of numeric code (referenced implicitly) before settling with *TaskGraph* for *DESOLA* (*Delayed Evaluation Self Optimising Linear Algebra*), a C++ library mentioned in Section 4.5.1. However, *Haskell* and *LLVM* may be the most suitable targets due to their maturity and availability as part of *Haskell* development toolchain.

6.6.2 Liveness Analysis and Mutability

LiveFusion takes a very high level specification of the program and converts in into an intermediate assembly-style *Loop* language. The *Loop* language has unconstrained control flow and destructive update which poses a problem when generating code targeting *single assignment* backends such as *Haskell* where all variables are immutable by default. *LLVM*, similarly, requires that all registers are in *SSA* (*Static Single Assignment*) form.

In order to determine appropriate scoping of bound variables in the generated code and ensure that all destructive updates are represented as fresh variables, a liveness analysis pass was implemented (Section 6.4). The analysis has been developed independently, however, in retrospect it is similar to the ubiquitous data flow analysis of control flow graphs employed by optimising compilers [5, 55] including *GHC*'s own *Cmm* language.

While *LiveFusion* sports its own implementation of data flow analysis is may be possible to reuse a more easily extensible library for this purpose. For example, *Hoopl* [44] which provides a way to encode data flow analysis and transformations.

Chapter 7

Results

Throughout this chapter we will use a *Data Parallel Haskell* implementation of *QuickHull* algorithm [3] that computes the *convex hull* of a finite set of points in the plane. It derives its name from the *QuickSort* algorithm and similarly uses a divide and conquer approach.

The convex hull for a set of points in the plane can be visualised as a polygon formed by a rubber band stretched around the points. More formally, convex hull is the smallest set of points forming the polygon that encloses all other points.

Figure 7.1 depicts three steps of a sample run of *QuickHull* algorithm with the final result shown on the right.

7.1 QuickHull in DPH

Data Parallel Haskell implementation of *QuickHull* in Listing 33 closely follows the recursive solution:

1. Find the points with smallest and largest x coordinates (lines 12 and 13 as well as left of Figure 7.1). These points are bound to be in the convex hull.

2. The line between the two points forms two subsets of points, which will be processed recursively (line 8) by `QuickHullR` function.

3. The recursive `QuickHullR` function, given a subset of points and a line, determines the point, on one side of the line, farthest from it (line 27). This point is also in the convex hull.

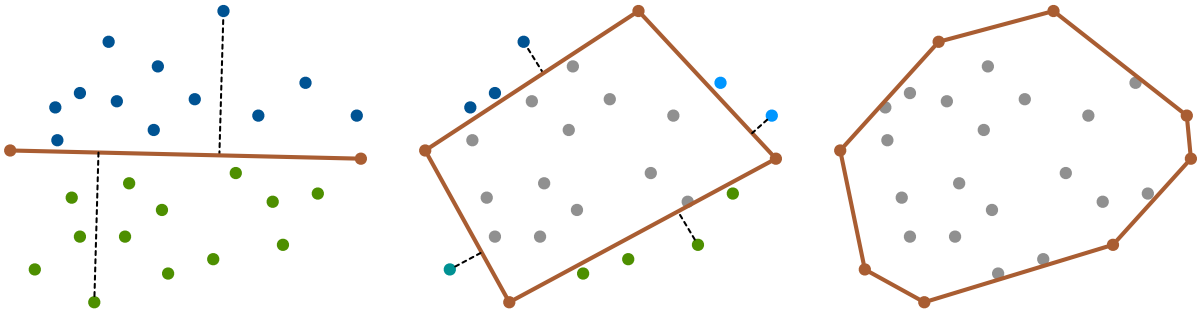


Figure 7.1: Three steps of a sample run of data parallel *QuickHull* algorithm.

4. `QuickHullR` also filters out all points on the other side of the line (line 25), keeping only the points `above` the line.

5. The two ends of the line each form a new line with the `far` point to be recursively processed (line 19).

6. The previous three steps are repeated, expanding the convex hull until no more points are left above any of the lines. When recursion is finished, points forming the lines constitute the convex hull.

7.2 QuickHull vectorisation

QuickHull program presented is a typical example of a divide and conquer algorithm. Recursive calls of `quickHullR` function in most languages would be processed sequentially one at a time. However, in *DPH* the function is vectorised as described in Section 3.2 to give $\texttt{quickHullR}^{\uparrow}$ of the following type:

```
-- Original
QuickHullR  :: [:Point:] → Line → [:Point:]
-- Vectorised
QuickHullR↑ :: [:[:Point:]:] → [:Line:] → [:[:Point:]:]
```

The vectorised function is able to process not one, but an array of lines at a time. This means that the "expansion" of convex hull in every direction happens in one call of $\texttt{quickHullR}^{\uparrow}$. In fact the three steps of vectorised *QuickHull* depicted in Figure 7.1 faithfully represent the only three calls to $\texttt{quickHullR}^{\uparrow}$ required to find the convex hull.

```
1  type Point = (Double, Double)
2  type Line  = (Point,  Point)
3
4  QuickHull :: [:Point:] → [:Point:]
5  QuickHull points
6    | lengthP points == 0 = points
7    | otherwise
8    = concatP [: QuickHullR points ends
9                  | ends ← [: (minx, maxx), (maxx, minx) :] :]
10   where
11     xs   = [: x | (x, y) ← points :]
12     minx = points !: minIndexP xs
13     maxx = points !: maxIndexP xs
14
15 QuickHullR :: [:Point:] → Line → [:Point:]
16 QuickHullR points line@(start, end)
17   | lengthP above == 0 = [:start:]
18   | otherwise
19   = concatP [: QuickHullR above ends
20                 | ends ← [:(start, far), (far, end):] :]
21   where
22     -- Find relative distance from each point to the line
23     distances = [: distance p line | p ← points :]
24     -- Only keep points above the line
25     above = [: p | (p,c) ← zipP points distances, c > 0.0 :]
26     -- Find the point farthest from the line
27     far = points !: maxIndexP distances
28
29 -- Cross product used as distance-like measure
30 distance :: Point → Line → Double
31 distance (xo, yo) ((x1, y1), (x2, y2))
32   = (x1 - xo) * (y2 - yo) - (y1 - yo) * (x2 - xo)
```

Listing 7.1: *Data Parallel Haskell* implementation of *QuickHull*.

7.3 QuickHull in the backend

The vectorised `quickHullR`$^{\uparrow}$ function takes a *nested parallel array* of points `[:[:Point:]:]` as its first argument. It was discussed in Section 3.2.1 that such nested arrays are represented using *segmented* arrays in the backend.

Recalling that an *array of pairs* is repesented as a *pair of arrays* (Section 3.3), the *Flattening transform* represents a *nested array* of `Point` as two data arrays and a segment descriptor:

```
type Point = (Double, Double)

Points:
  [:
     [:(3,3):],
     [:(4,7), (2,8),  (2,9), (4,9):],
     [:(8,8), (7,7),  (7,9), (9,10):],
     [:(9,5), (10,3), (8,1):],
     [:(4,0):]
  :]
```

⇓

```
segd: [ 1, 4, 4, 3, 1 ]
       _  ___________  ____________  ________  _
xs:   [ 3, 4, 2, 2, 4, 8, 7, 7,  9, 9, 10, 8, 4 ]
ys:   [ 3, 7, 8, 9, 9, 8, 7, 9, 10, 5,  3, 1, 0 ]
```

Since the the second argument to the function is a *flat* array of `Line`, no segment descriptor is required. However, it does need four separate arrays in accordance with the type of `Line`. For example:

```
type Line = (Point, Point)

x1s = [2, 1, 4, 8, 9]
y1s = [1, 5,10, 6, 0]
x2s = [1, 4, 8, 9, 2]
y2s = [5,10, 6, 0, 1]
```

Overall, `quckHullR`$^{\uparrow}$ function goes through the following transformations before it is compiled using the backend library of collective array operations (such as *LiveFusion*)[1]:

[1] I omit certain internal wrapper types to simplify the example.

```
QuickHullR  :: [:Point:] → Line → [:Point:]

                        ⇓   Lifting transform

QuickHullR↑ :: [:[:Point:]:] → [:Line:] → [:[:Point:]:]

                        ⇓   Flattening transform

QuickHullR_ :: Array Int                        -- segment descriptor
            → Array Double → Array Double       -- points
            → Array Double → Array Double       -- line starts
            → Array Double → Array Double       -- line ends
            → Array Double → Array Double       -- convex hull points
```

7.4 The heart of QuickHull: segmented FilterMax

By the time a *DPH* program is compiled in the backend, it is composed of a large data flow graph of flat and segmented array combinators. The complete data flow graph for *QuickHull* is shown in Figure 7.2.

While the graph consists of over 40 array combinators, roughly $^2/_3$ of its runtime the *QuickHull* program spends in what we will refer to as *segmented FilterMax*. The data flow diagram logically groups the constituent parts of *FilterMax* in boxes labelled a through e. In each recursive step of *QuickHull* it uses relative distances computed by group of combinators a to find points above the lines (c, d and e) as well as points farthest from each line (b).

The complete implementation of *FilterMax* using *LiveFusion* interface is given in Appendix C. I have implemented it by studying the *GHC Core* code generated by *DPH* vectoriser[2].

7.5 Stream Fusion and FilterMax

In the previous section we have identified groups of combinators a through e that make up the functionality of segmented *FilterMax*. The grouping is not incidental. When vectorised *QuickHull* is compiled using *Stream Fusion* framework the combinators in each of these groups get fused together.

[2]The complete hand-vectorised implementation of *QuickHull* is available at `http://github.com/ghc/packages-dph/tree/master/dph-examples/examples/spectral/QuickHull/handvec`

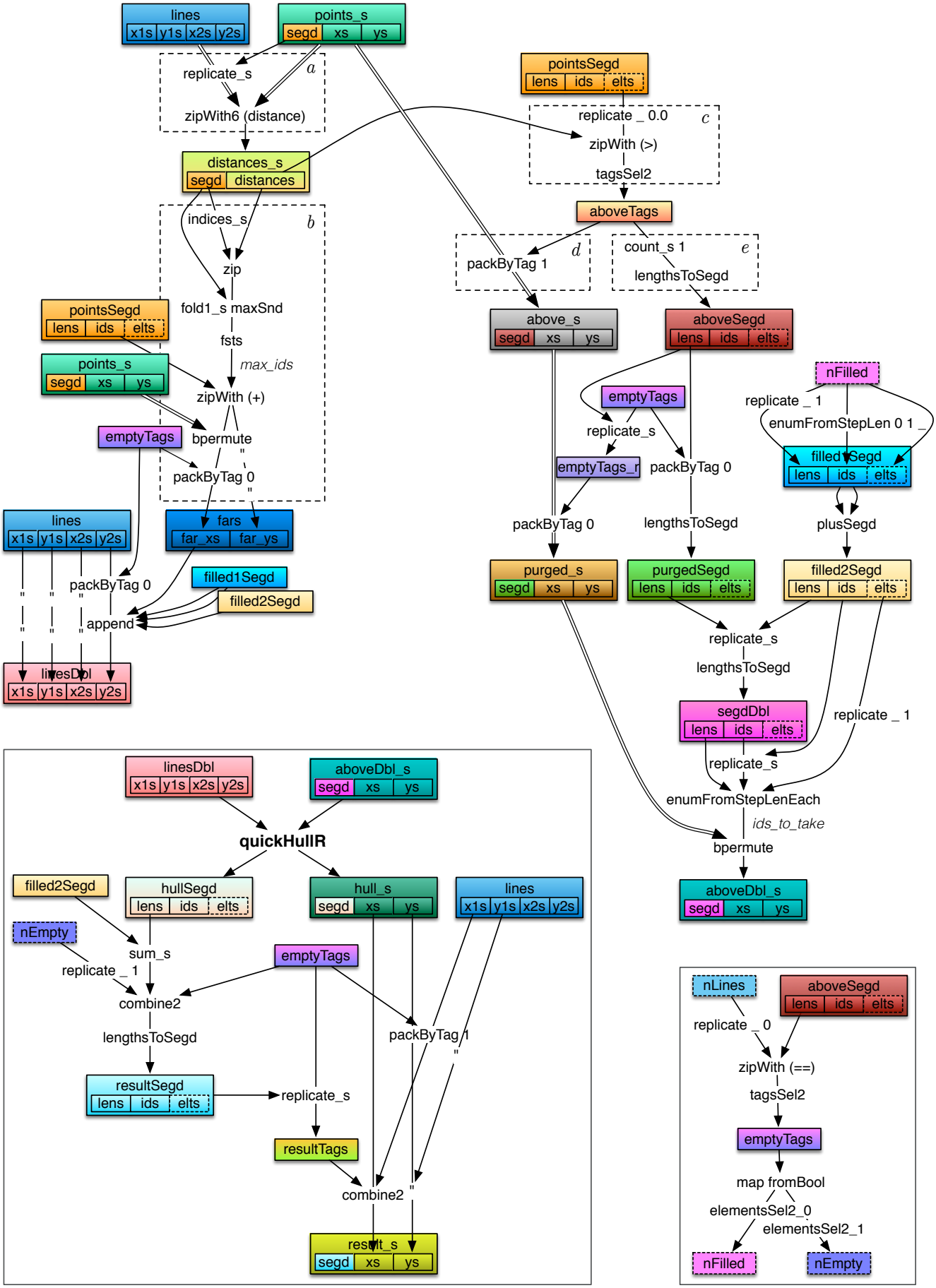


Figure 7.2: Data flow diagram of array combinators generated by the vectoriser for `QuickHullR` function.

However, *the fusion stops at the boundaries of these groups.*

The *FilterMax* example exhibits both problems with *Stream Fusion* identified in Section 2.1: the inability to fuse producers with multiple consumers and the duplicated counters.

7.5.1 Multiple consumers

Both the filtering part of *FilterMax* finding points above the lines (c, d, e), and the part finding the farthest points (b), rely on the distances to be computed first (a). Because of the inherent limitation of fusion systems that are based on rewrite rules (Section 2.1.2), *Stream Fusion* is unable to fuse the producer of the segmented `distances_s` array into the two consumers although they could easily be computed in the same loop.

For the same reason combinator groups d and e cannot be fused with group c that produces an array of boolean tags specifying which points should be kept and which filtered out.

In fact, the pattern formed by combinator groups c, d and e is very common to *DPH* programs. Based on the result of some computation in c, a new segmented array is produced in d *AND* a new accompanying segment descriptor in e.

LiveFusion on the other hand is able to fuse all combinator groups a through e into a single (nested) loop.

7.5.2 Duplicated loop counters

With arrays of points being represented by two arrays and arrays of lines – by four arrays, any computation involving their point-wise processing introduces at superfluous counter variable *and* bounds checks in *Stream Fusion*. This problem was discussed in Section 2.1.3 and solved in *LiveFusion* using rates (Section 5.3.3).

Referring to the *FilterMax* example, `zipWith6` combinator from group a uses 6 separate counters for each of the input arrays and one for the output. Similarly, `zip` and `zipWith` combinators of groups b and c use 3 counters each.

7.6 Evaluation

Benchmarking in this section has been carried out using an implementation of *QuickHull* that was vectorised by hand, closely following the intermediate *GHC Core* code produced by *DPH* vectoriser. It is expressed using array combinators.

A variant of *QuickHull* that uses *LiveFusion* to compute *FilterMax* has been compiled. As of this writing *LiveFusion* is able to dynamically generate and load fused code. However, it does not yet cache the compiled code. To avoid recompiling the same combinator graph in each recursive step of *QuickHull*, the pre-generated code of *FilterMax* has been compiled in. Additional compilation costs are factored into the results where appropriate.

The *Stream Fusion* variant was compiled against `dph-prim-seq` library found in the *DPH* distribution.

Both programs have been run on 10, 20, 25, 40 and 50 million points. The experiments were conducted on an Apple MacBook Pro 8.2 with a 2 GHz Intel Core i7 CPU with 256 KB L2 and 6 MB L3 cache and 16 GB of system memory. The benchmarks were compiled with a development version of GHC (7.9.20140514).[3]

7.6.1 Overall performance

Figure 7.3 shows the runtime of the complete *QuickHull* program. In one instance it uses *Stream Fusion* throughout. In another it substitutes *LiveFusion* generated code to compute *FilterMax*. The results are shown numerically in Figure 7.4 exhibiting 1.46 – 1.84 speedups even with amostised compilation costs factored in ($\approx 650\ ms$). It should be noted that *FilterMax* only accounts for $^2/_3$ of the runtime so a complete *LiveFusion* implementation is expected to yield further speedups.

7.6.2 FilterMax performance

To compare *LiveFusion* and *Stream Fusion* directly, Figures 7.2 and 7.4 show the performance of each running just *FilterMax*. The running times given are for 100 million points and less. Timing for fewer than 65 thousand points has been omitted even though it shows at least as good performance of *LiveFusion* as compared to *Stream Fusion*.

[3]Patched for *Template Haskell* pretty printing bug #9022 `http://ghc.haskell.org/trac/ghc/ticket/9022`.

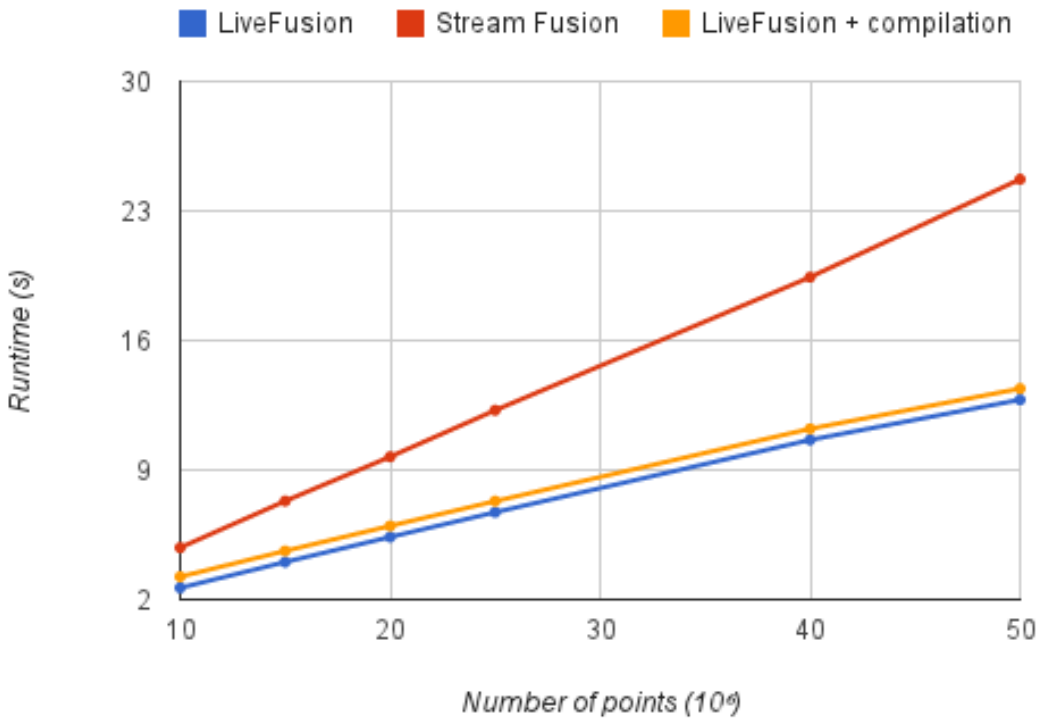


Figure 7.3: Runtime of vectorised *QuickHull* with *FilterMax* fused by *Stream Fusion* and *LiveFusion*.

Number of points	LiveFusion	Stream Fusion	LiveFusion + compilation	Speedup
10000000	2631	4803	3281	1.46
15000000	4026	7322	4676	1.57
20000000	5378	9715	6028	1.61
25000000	6714	12227	7364	1.66
40000000	10627	19423	11277	1.72
50000000	12798	24715	13448	1.84

Figure 7.4: Runtime of vectorised *QuickHull* with *FilterMax* fused by *Stream Fusion* and *LiveFusion* (measured in milliseconds).

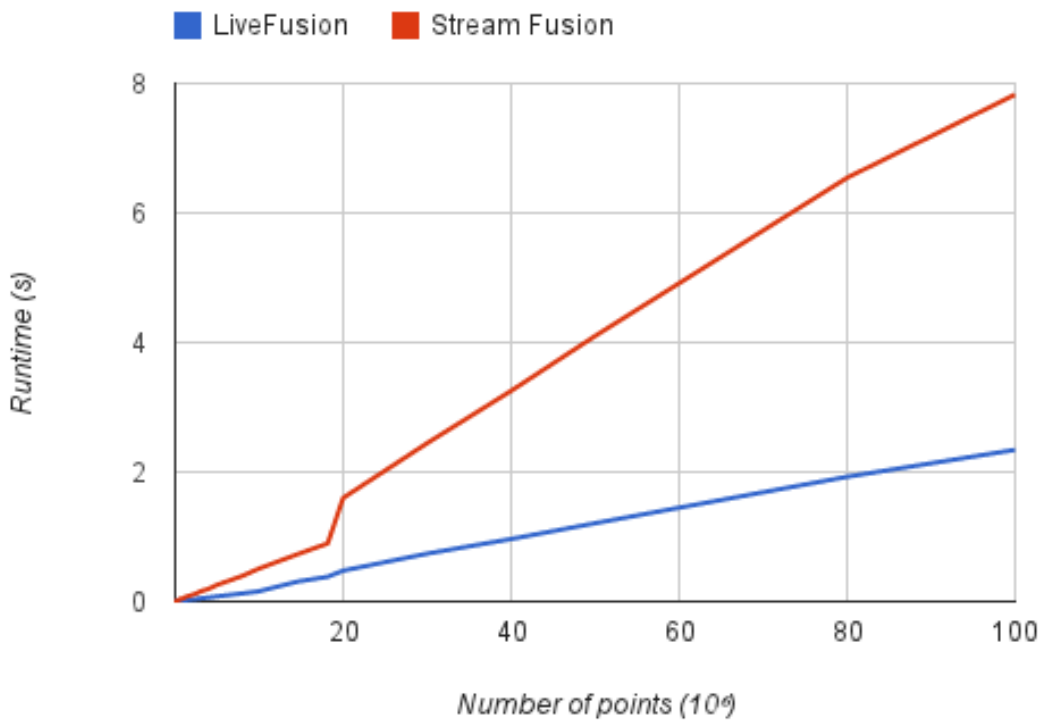


Figure 7.5: Runtime of *FilterMax* fused by *Stream Fusion* and *LiveFusion*.

Excluding the speedups for smaller number of points (161482 points and below, which tend to be higher), the average speedup observed is 3.2.

7.6.3 Discussion

It is important to note that the both *LiveFusion* and *Stream Fusion* are able to generate highly optimised code. Performance advantage of *LiveFusion* mainly comes from its ability to exploit more fusion opportunities than *Stream Fusion*. In the case of *FilterMax*, *LiveFusion* was able to fuse into one loop what *Stream Fusion* fused into five separate loops. Importantly each of those five loops were running at the rate of the number of points. The overhead of reading and writing large arrays to the memory by the *Stream Fusion* implementation had significant impact on the runtime of the program. *LiveFusion*'s improved performance is in a major way attributed to reduced memory traffic.

Number of lines	Number of points	LiveFusion	Stream Fusion	Speedup
2	100000000	2342	7828	3.3
2	80000000	1926	6543	3.4
2	50000000	1209	4100	3.4
2	40000000	965	3253	3.4
2	30000000	739	2445	3.3
2	20000000	477	1599	3.4
16	9971928	157	508	3.2
16	7979704	124	394	3.2
16	4988482	79	256	3.2
16	3989894	62	196	3.2
16	2992208	47	155	3.3
32	2550344	39	126	3.2
32	2040956	32	102	3.2
32	1275378	19	73	3.8
32	1020092	16	53	3.3
32	764138	12	40	3.3
64	643906	14	43	3.1
64	514694	8	28	3.5
32	508870	9	27	3
64	322464	5	18	3.6
64	257710	4	15	3.8
64	193086	3	11	3.7
128	161482	2	10	5
128	129112	2	8	4
64	128766	4	8	2
128	81172	1	5	5
128	65086	1	5	5

Figure 7.6: Runtime of *FilterMax* fused by *Stream Fusion* and *LiveFusion* (measured in milliseconds).

Chapter 8

Conclusion

The problem of array fusion is not in its infancy. Yet there is no definitive and widely accepted approach to it. Every technique is often tailored to the surrounding context to support the programming model offered by the framework. The work described in the present dissertation has been carried out in the context of nested data parallelism (Chapter 3) in purely functional languages. It was argued that *equational* approaches to fusion (Appendix A) fall short in attempting to fuse complex combinator graphs (Section 2.1.2) using only local program transformations (rewrite rules).

Chapter 4 introduced an EDSL based approach to fusion using a new language called *LiveFusion*. Following this approach, collective array operations are not performed immediately but are recorded in an AST. The chapter also covered sharing recovery of AST terms (Section 4.3.1) as well as a way of embedding inlineable scalar functions (Section 4.4).

Chapter 5 presented a common loop structure to which fusible combinators can be mapped using an assembly-like *Loop* language. It showed the translation of a large number of combinators to loops which included segmented combinators (Section 5.7) that are fundamental to fusing vectorised code.

Chapter 6 offered the requirements to the backend code generators such as fast mutable arrays and mutable variables. It applied these requirements to a backend that generated monadic *Haskell* code through the use of *Template Haskell*. The chapter also described *liveness analysis* used to analyse data flow in a control flow graph of *Loop*'s basic blocks (Section 6.4). The analysis ensured that the generated code is well-scoped and all variables are assigned exactly once (generating fresh names on destructive update).

Chapter 7 applied the *LiveFusion* framework to the *QuickHull* algorithm. The algorithm was chosen since it was illustrative of *DPH* vectoriser's inner workings and used a wide range of combinators offered by *LiveFusion*. It was shown that *LiveFusion*'s ability to fuse more complex combinator graphs can yield speedups of 3.2 times on average as compared to *Stream Fusion*.

8.1 Contributions

In this thesis I present two main research contributions. First, I show how to implement an embedded domain specific language that is able to:

- fuse flat and segmented array combinators without depending on compiler term rewriting
- fuse combinator graphs with a large amount of internal node sharing
- fuse graphs producing multiple results and/or demand materialisation of graph's intermediate nodes through explicit tupling.

LiveFusion language lifts many of the restrictions imposed by currently available fusion libraries such as the inability to fuse a producer into multiple consumers or to compute multiple results in one fusion pass (tupling). Compared to frameworks that do allow tupling and/or multiple consumers fusion, the new language has support for segmented combinators which is vital for its application in nested data parallelism.

Second, I devise a generic representation of fusible loops onto which many types of combinators can be mapped. The *Loop* language with a *rate* system and built-in control and data flow analysis offers a way to write composable loops. This is a core contribution which may be applied in a variety of languages and programming paradigms. It does not have to be a runtime feature, provided the combinator graph is known at compile time and is free of control flow.

8.2 Future work

Following on from the work carried out and the results achieved, there are several research and development paths that can be taken to advance the *LiveFusion* system.

8.2.1 Parallelism and vectorisation

The work described in this thesis has resulted in a system that can exploit more fusion opportunities than previous systems. However, the code generated by *LiveFusion* is still sequential. The reason for this is that I focused on developing a novel fusion system rather than a parallel execution engine.

Granted, such a fusion system targeting nested data parallel applications defeats the purpose. Granted, introducing parallel execution would have its own challenges since it typically results in more fusion-preventing edges in combinator graphs.

However, parallelising collective array operations is a well researched problem which includes the lessons learnt from implementing parallelism in the *Stream Fusion* based backend of *DPH*. In particular *Distributed Types* [9] allow for seamless parallelisation given sequential combinator implementations. Additionally, my previous work on parallelism granularity shows that for many operations, the overhead of distributing the computation across threads may be considerably higher than computing it sequentially.

Lastly, with vector registers growing in size in modern CPUs (512 bits in the near future [20]) and their instruction sets expanding, it may prove to be fruitful to target instruction-level vectorisation at the same time as full-featured parallelism.

8.2.2 Clustering and advanced rate inference

Not all combinator graphs can be fused into a single loop, especially in the parallel context. Randomly accessing the data, or reducing an array using `fold`, or scanning an array in the parallel context – all of these operations are fusion-preventing.

As discussed in Section 5.8.3 there may be more than one way to cluster combinators into fusible groups. At present the fusion performed by *LiveFusion* is opportunistic. Preventing fusion where it is not allowed and scheduling multiple loops would be the next priority for *LiveFusion*.

Finally, a more sophisticated rate inference system supporting flat and segmented appends and other interleaved access patterns would also be highly beneficial to the application of *LiveFusion* in *DPH*. It would not be an understatement to say that the rate system, being fundamental to composing loops, is at the absolute core of the proposed fusion technique.

Appendices

Appendix A

Equational fusion systems

This section will review two approaches to compile-time fusion in *Haskell*: *foldr/build* short cut fusion and *Stream Fusion*. They both fall into the category of *equational* fusion systems meaning that they rely on compiler's *rewrite rules* [41] for them to work.

A major problem with this approach is that it cannot fuse a producer into multiple consumers which is vital for *Data Parallel Haskell* as discussed in Section 2.1.

A.1 *foldr/build* fusion

foldr/build short cut fusion [17] is one of the most referenced techniques for fusion in *Haskell*. It has been developed for use in *GHC* to fuse pipelined list operations. It employs two combinators, `foldr` and `build`, and a single rewrite rule to eliminate adjacent occurrences of the combinators. It is suitable for use in the cases when the functions to be fused can be defined in terms of these combinators. The definition of `foldr` is reused from *Prelude – Haskell*'s standard library:

```
foldr :: (a → b → b) → b → [a] → b
foldr f z []     = z
foldr f z (x:xs) = f x z : (foldr f z xs)
```

To understand the motivation behind short cut fusion one may think of `foldr` as of replacing each `Cons` of a list with a binary operator and `Nil` with the neutral element. The other combinator, `build`, takes a second order function which in turn takes an operator to be used as `Cons`, and a value to be used as `Nil`. Since we are building a list, these immediately become `(:)` and `[ ]`.

```
build :: ((a → b → b) → b → b) → [a]
build g = g (:) []
```

To illustrate the approach a `map` could be defined in terms of `build/foldr` as follows:

```
map f xs = build (λc n → foldr (c . f) n xs)
-- c is (:), n is [ ]
```

In the above code the list is being folded into another list. With the help of a rewrite rule

$\langle foldr/build\ fusion \rangle \forall g\, k\, z. foldr\, k\, z\, (build\, g) \mapsto g\, k\, z$

the two consecutive `map`s could be reduced to as in the following:

```
map h (map f xs)
 -- inline
 = build(λc n → foldr (c.h) n
  (build (λc n → foldr (c.f) n xs)))
 -- apply rewrite rule
 = build(λc n → ((λc n → foldr (c.f) n xs) (c.h) n)
 -- β-reduce
 = build(λc n → foldr (c.h.f) n xs)
```

While this leads to the desired results in many cases, it requires the programmer to define the functions in a less readable form. This may be acceptable for some of the code

in the standard library but is not likely to be widely accepted among client programmers. Thus the fusion breaks for the parts of the code that do not use the explicit `foldr/build` definitions. To solve this, Chitil proposed a type inference algorithm to automatically infer the `foldr/build` definitions [12].

The above limiting factor is critical to the application of the fusion system to lists but is less relevant to *LiveFusion* system since the number of combinators is fixed.

However, the limitations that do make it unattractive are the inability to effectively fuse left `fold`s and `zip`s which are both crucial to *DPH*. The inability to fuse into multiple consumers is likewise very considerable for such application of the technique as *DPH*.

A dual of `foldr/build` is `destroy/unfoldr` [50] fusion framework, which while able to fuse reductions and pointwise consumers, is also unable to fuse a producer into multiple consumers.

A.2 Stream Fusion

Stream Fusion [14, 15] is originally employed in the *DPH* backend library at one of the two levels of fusion discussed in Section 3.4.1.

It introduces two data types: a **stream** and a **stepper**.

```
data Stream a = ∀ s. Stream (s → Step s a) s
```

A *Stream* is defined by its **stepper function** and **seed**. The **seed** is essentially the state of the stream at any given point. The stepper is used to produce a stream by taking the current *seed* and yielding the *next step*. That is, the stepper function may produce the next *element* and *new state* from *current state*. The step produced by calling stepper on stream may be one of the following:

```
data Step s a = Yield a s
              | Skip s
              | Done
```

A `Done` flags the end of the stream, while `Yield` contains an element and the new seed (i.e. new state). If a `Skip` is encountered it means that the current step did not contain an element but the stream has not yet finished[1].

The streaming of a list could be defined in the following way:

```
stream :: [a] → Stream a
stream step0 = Stream next step0
  where next []     = Done
        next (x:xs) = Yield x xs
```

The list itself is being used as the seed. Unstreaming back to a list takes the following form:

```
unstream :: Stream a → [a]
unstream (Stream next0 step0) = unfold step0
  where unfold s = case next0 s of
          Done       → []
          Skip s'    → unfold s'
          Yield x s' → x : unfold s'
```

Any fusible list function should now be defined in terms of Streams as opposed to lists. For example the `map` combinator is defined as follows:

[1]For example if the stream has skipped an element that did not satisfy the predicate of a `filter` combinator.

```
mapS :: (a → b) → Stream a → Stream b
mapS f (Stream next0 s0) = Stream next s0
  where next s = case next0 s of
    Done       → Done
    Skip    s' → Skip        s'
    Yield x s' → Yield (f x) s'

map :: (a → b) → [a] → [b]
map f xs = unstream . mapS f . stream
```

Note that the user-facing definition of `map` still takes a regular lists but converts the list to a stream internally.

To see how a `Skip` step might be used it may be worthwhile to look at the definition of the `filter` function.

```
filterS :: (a → Bool) → Stream a → Stream a
filterS p (Stream next0 s0) = Stream next s0
  where next s = case next0 s of
    Done                  → Done
    Skip    s'            → Skip    s'
    Yield x s' | p x      → Yield x s'
               | otherwise → Skip    s'

filter :: (a → Bool) → [a] → [a]
filter p = unstream . filterS p . stream
```

Again, the user-facing filter function `streams` the list before passing it to the stream-based counterpart.

Given the above definitions of `map` and `filter` the fusion opportunity such as `map f . filter p` may now be exploited with the following rewrite rule

$\langle stream/unstream\ fusion \rangle \forall stream\, (unstream\, s) \mapsto s$

```
(map f . filter p) xs
  -- inline definitions of map and filter
  = (unstream . mapS f . stream . unstream . filterS p) xs
  -- apply stream/unstream rewrite rule
  = (unstream . mapS f . filterS p . stream) xs
```

The fusion is not complete at this point but the stream processing functions `mapS` and `filterS` are not adjacent and can be fused by inlining the definitions and *GHC*'s built-in optimisations. In order to completely remove all traces of the helper data structures Stream Fusion relies on general purpose compiler optimisations and a *Constructor Specialisation* optimisation [37].

A more thorough description list combinator fusion using Stream Fusion can be found in [14].

Stream Fusion has proven to be very potent for array processing and is successfully employed in the *Vector* library. It can fuse more types of combinators than *forldr/build* and *Functional Array Fusion* and has recently been extended to support vectorised CPU instructions [33].

Appendix B

Considerations for fusing lockstep consumers

This appendix follows from the discussion of `zipWith`N combinators in Section 5.5.2 on page 61.

In general the fusion of combinators that consume multiple arrays in lockstep (`zipWith`N family of combinators, `indices_s`, `replicate_s`, etc..) faces the following two problems:

1. The lengths of the input arrays may be different.
2. The loops producing some of all of the input arrays may not produce elements in every iteration.

The following two sections discuss the two problems in turn and offer potential solutions. It should be noted that neither of the problems are affecting the fusion of *Data Parallel Haskell* programs and can be left as future work.

```
let xys = zipWith (*) xs ys
    zs  = map (+1) ys
in  (xys, zs)
```

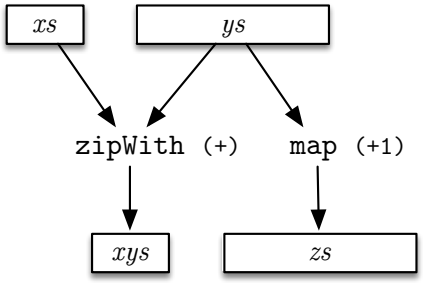


Figure B.1: Zipping arrays of different lengths. *Haskell* code (left) and the corresponding data flow diagram (right).

B.1 Inputs of different lengths

In the standard *Haskell* library the result list of `zipWith` combinator has the same length as the shortest of its inputs. In the *Loop* language this could easily be expressed by setting the output length to be the minimum of the input array lengths and adjusting the iteration range appropriately:

```
init_zip/xs/ys:
  let len_zip = min len_xs len_ys

guard_zip/xs/ys:
  unless ix_zip < len_zip | done_zip
```

However, there are still implications of this approach. Consider the program in Figure B.1. The `ys` array is consumed both by `zipWith` and `map`. Supposing that `xs` is shorter than `ys` it makes it harder (though not impossible) to compute both `xys` and `zs` in the same loop.

In order to successfully fuse the above example the *Loop* language needs to be able to express overlapping loop ranges. In this particular example a portion of the index range should be processed by both `zipWith` and `map` operations, and the rest with just the `map`.

Since *LiveFusion* was designed to target *DPH* , and *DPH* statically guarantees the inputs to be of the same lengths, the restriction on accepted input lengths is justified.

B.2 Skipping elements in the inputs

LiveFusion has support for fusing loops which do not produce an elements in every iteration. The most prominent of those is perhaps the `filter` combinator. If the combinator pipeline producing an input to `zipWith` contains a `filter`, fusing them becomes more challenging.

Consider the following example:

```
let xs = filter odd [0..9]
    ys = [11..15]
    zs = zipWith (*) xs ys
in  zs
```

While `zipWith` is supposed to consume elements from `xs` and `ys` in a lockstep, producing both those elements will not always happen in the same iteration. `ys` array does not pose a problem, it is guaranteed to produce one element in each iteration. However, the addition of `filter` into the equation for `xs` will mean that the loop for `xs` may require more than one iteration to produce an element. If the `filter` skips an element the loop for `ys` must wait.

B.2.1 A solution

Considering a potential implementation of the same program in a procedural language reveals that it is still possible to only use one overall loop for computing `zs`.

However, a separate nested loop is required for computing the elements of `xs`. This loop will exit when the `filter` predicate is satisfied.

Figure B.2 presents a potential solution to the problem expressed as a control flow graph in the *Loop* language. The bodies of `zipWith` and `ys` are merged together in the outer loop while the whole of `xs` including `filter` is an loop which is entered before the body of `zipWith` is executed.

B.2.2 Implications

A solution described above is plausible in *LiveFusion*. However, it requires more sophisticated analysis of the combinator graph.

Again, the *DPH* system which is the primary target of *LiveFusion* is guaranteed to feed `zipWith` combinator with uniformly produced arrays. Even if the input arrays are

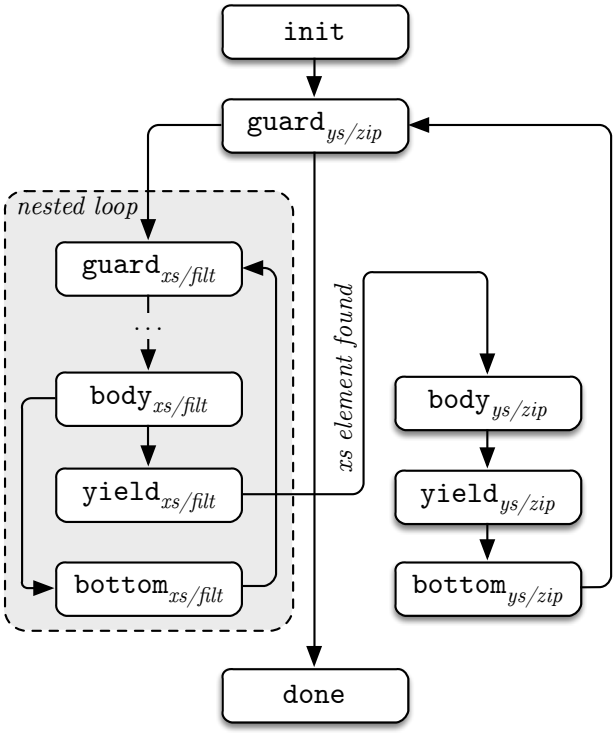


Figure B.2: *Loop* language CFG for a nested loop solution to lockstep consumption of producers that may skip.

filtered they all skip of produce an element at the same time.

Appendix C

Listings

```
init_{bperm/is}:
  let len_{is} = arrayLength arr_{is}
  let len_{xs} = arrayLength arr_{xs}
  let len_{bperm} = len_{is}
  let ix_{is} = 0
  let arr_{bperm} = newArray len_{bperm}
  goto guard_{bperm}

guard_{bperm/is}:
  unless ix_{is} < len_{is} | done_{bperm}
  goto body_{bperm}

body_{bperm/is}:
  let elt_{is} = readArray arr_{is} ix_{is}
  -- set index to read xs at
  let ix_{r} = elt_{is}
  let elt_{xs} = readArray arr_{xs} ix_{r}
  let elt_{bperm} = elt_{xs}
  goto yield_{bperm}

yield_{bperm/is}:
  ix_{is} := ix_{is} + 1
  writeArray arr_{bperm} ix_{is} elt_{bperm}
  goto bottom_{bperm}

bottom_{bperm/is}:
  goto guard_{bperm}

done_{bperm/is}:
  let result_{bperm} = sliceArray arr_{bperm} ix_{is}
  return (result_{bperm})
```

Figure C.1: *Loop* code generated for `bpermute xs is` where both data (`xs`) and indices (`is`) arrays are manifest and the result array is forced. Referenced in Section 5.6.

```
init_fold/segd/data:
  let len_segd = arrayLength arr_segd
  let len_data = arrayLength arr_data
  let len_fold = len_segd
  let z_fold = 0
  let ix_segd = 0
  let ix_data = 0
  let arr_fold = newArray len_fold
  goto guard_fold

guard_fold/segd:
  unless ix_segd < len_segd | done_fold
  goto nest_fold

nest_fold/segd:
  let acc_fold = z_fold
  let elt_segd = readArray arr_segd ix_segd
  let end_segd = ix_data + elt_segd
  goto guard_data

body_fold/segd:
  let elt_fold = acc_fold
  goto yield_fold

yield_fold/segd:
  ix_segd := ix_segd + 1
  writeArray arr_fold ix_segd elt_fold
  goto bottom_fold

bottom_fold/segd:
  goto guard_fold

done_fold/segd/data:
  let result_fold = sliceArray arr_fold ix_segd
  return (result_fold)
```

```
guard_data:
  unless ix_data < end_segd | body_fold
  goto nest_data

nest_data:
  goto body_data

body_data:
  let elt_data = readArray arr_data ix_data
  goto yield_data

yield_data:
  ix_data := ix_data + 1
  goto bottom_data

bottom_data:
  acc_fold := acc_fold + elt_data
  goto guard_data
```

Figure C.2: *Loop* code generated for `folds (+) 0 segd data` where both `segd` and `data` are manifest arrays and the result array is forced. Referenced in Section 5.7.2.

```
init_scan/segd/data:
  let len_segd = arrayLength arr_segd
  let len_data = arrayLength arr_data
  let len_scan = len_data
  let z_scan = 0
  let ix_data = 0
  let ix_segd = 0
  let arr_scan = newArray len_scan
  goto guard_segd

guard_segd:
  unless ix_segd < len_segd | done_scan
  goto nest_segd

nest_segd:
  let acc_scan = z_scan
  let elt_segd = readArray arr_segd ix_segd
  let end_segd = ix_data + elt_segd
  goto guard_scan

body_segd:
  goto yield_segd

yield_segd:
  ix_segd := ix_segd + 1
  goto bottom_segd

bottom_segd:
  goto guard_segd

guard_scan/data:
  unless ix_data < end_segd | body_segd
  goto nest_scan

nest_scan/data:
  goto body_scan

body_scan/data:
  let elt_data = readArray arr_data ix_data
  let elt_scan = acc_scan
  goto yield_scan

yield_scan/data:
  ix_data := ix_data + 1
  writeArray arr_scan ix_data elt_scan
  goto bottom_scan

bottom_scan/data:
  acc_scan := acc_scan + elt_data
  goto guard_scan

done_scan/segd/data:
  let result_scan = sliceArray arr_scan ix_data
  return (result_scan)
```

Figure C.3: *Loop* code generated for `scanls (+) 0 segd data` where both `segd` and `data` are manifest arrays and the result array is forced. Referenced in Section 5.7.2.

Segmented FilterMax in *LiveFusion*

The following is the complete source of the program representing the FilterMax component of vectorised QuickHull. QuickHull was discussed in Section 7 on page 93 with *Data Parallel Haskell* source given in Listing 33 on page 95.

```
-- The heart of vectorised QuickHull.
-- Based on hand-vectorised version of QuickHull found in:
-- ghc/libraries/dph/dph-examples/examples/spectral/QuickHull/handvec/Handvec.hs
{-# LANGUAGE RebindableSyntax #-}
module FilterMax where

import Data.LiveFusion

import Prelude as P hiding ( map, replicate, zip, zipWith,
                             filter, fst, snd, unzip )

filterMax :: (IsNum a, IsOrd a, Elt a)
          ⇒ Term Int            -- Number of points
          → Array Int           -- Segd
          → Array a → Array a   -- Points
          → Array a → Array a   -- Line starts
          → Array a → Array a   -- Line ends
          → (Array a, Array a,  -- Farthest points
             Array a, Array a,  -- Points above lines
             Array Int)         -- New segd
filterMax npts segd xs ys x1s y1s x2s y2s
  = let -- Find distance-like measures between each point and its
respective line.
        distances = calcDistances npts segd xs ys x1s y1s x2s y2s

        -- Throw out all the points which are below the line.
        (above_xs, above_ys, aboveSegd) = calcAbove npts segd xs ys distances

        -- Find points farthest from each line.
        (fars_xs, fars_ys) = calcFarthest npts segd xs ys distances
    in  (fars_xs, fars_ys, above_xs, above_ys, aboveSegd)
```

```
-- | Find (relative) distances of points from line.
--
-- Each point can be above (positive distance) or below (negative
distance)
-- a line as looking from the centre of the convex hull.
--
-- Corresponds to 'distances' in the original program:
-- > distances = [:  distance p line | p ← points :]
calcDistances :: (IsNum a, IsOrd a, Elt a)
              ⇒ Term Int            -- Number of points
              → Array Int           -- Segd
              → Array a → Array a   -- Points
              → Array a → Array a   -- Line starts
              → Array a → Array a   -- Line ends
              → Array a             -- Relative distances from lines
calcDistances npts segd xs ys x1s y1s x2s y2s
  = zipWith6 distance xs
                      ys
                      (replicate_s npts segd x1s)
                      (replicate_s npts segd y1s)
                      (replicate_s npts segd x2s)
                      (replicate_s npts segd y2s)



-- | Compute cross product between vectors formed between a point (x,y)
-- and each of the two line ends:  (x1,y1) and (x2,y2).
distance :: (IsNum a, IsOrd a, Elt a)
         ⇒ Term a → Term a  -- Point
         → Term a → Term a  -- Line start
         → Term a → Term a  -- Line end
         → Term a           -- Distance
distance x y x1 y1 x2 y2
  = (x1 - x) * (y2 - y) - (y1 - y) * (x2 - x)
```

```
-- | Find points above the lines given distance-like measures.
--
-- Corresponds to 'above' in the original program:
-- > above = [:  p | (p,c) ← zipP points distances, c > 0.0 :]
calcAbove :: (IsNum a, IsOrd a, Elt a)
          ⇒ Term Int            -- Number of points
          → Array Int           -- Segd
          → Array a → Array a   -- Points
          → Array a             -- Distances
          → (Array a, Array a,  -- Points with positive distances
             Array Int       )  -- New Segd
calcAbove npts segd xs ys distances
  = let -- Compute selector for positive elements
        aboveTags  = zipWith (>.) distances (replicate npts 0)

        -- Compute segd for just the positive elements
        aboveSegd = count_s true segd aboveTags

        -- Get the actual points corresponding to positive elements
        (above_xs, above_ys)
                  = unzip
                  $ packByBoolTag true aboveTags
                  $ zip xs ys

    in  (above_xs, above_ys, aboveSegd)
```

```
-- | For each line find a point farthest from that line.
--
-- Each segment is a collection of points above a certain line.
-- The array of Doubles gives (relative) distances of points from the
line.
--
-- Corresponds to 'far' in the original program:
-- > far = points !:  maxIndexP distances
calcFarthest :: (IsNum a, IsOrd a, Elt a)
             ⇒ Term Int            -- Number of points
             → Array Int           -- Segment descriptor
             → Array a → Array a   -- Points
             → Array a             -- Distances
             → (Array a, Array a)  -- Points with biggest distances
calcFarthest npts segd xs ys distances
  = let -- Index the distances array, and find the indices corresponding
to the
        -- largest distance in each segment
        indexed  = zip (indices_s npts segd)
                       distances
        max_ids  = fsts
                 $ fold_s maxSnd (0 .*. small) segd indexed

        small    = -999999

        -- Find indices to take from the points array by offsetting from
segment starts
        ids      = zipWith (+) (indicesSegd segd)
                               max_ids
        max_xs   = bpermute xs ids
        max_ys   = bpermute ys ids

        -- We are only interested in the ones which are above the line
        -- (thus from segments containing points above line).
    in (max_xs, max_ys)


-- | Compute array containing starting indices of each segment
-- of a segment descriptor.
indicesSegd :: Array Int → Array Int
indicesSegd = scan (+) 0


-- | Find pair with the biggest second element.
maxSnd :: (Elt a, Elt b, IsOrd b) ⇒ Term (a, b) → Term (a, b) →
Term (a, b)
maxSnd ab1 ab2 = let b1 = snd ab1
                     b2 = snd ab2
                 in  if b1 ≥. b2 then ab1
                                 else ab2
```

Bibliography

[1] Eric Anvar Ghuloum and Jesse Fang Sprangle. "Flexible Parallel Programming for Tera-scale Architectures with Ct". 2007. URL: http://download.intel.com/pressroom/kits/research/Flexible_Parallel_Programming_Ct.pdf.

[2] Robert Atkey, Sam Lindley and Jeremy Yallop. "Unembedding domain-specific languages". In: *Haskell '09: The 2nd ACM SIGPLAN Symposium on Haskell*. New York, NY, USA: ACM, 2009, pp. 37–48.

[3] C. B. Barber, D. P. Dobkin and H. T. Huhdanpaa. "The Quickhull algorithm for convex hulls". In: *ACM Trans. on Mathematical Software* (Dec. 1996).

[4] Guy E. Blelloch et al. "Implementation of a Portable Nested Data-Parallel Language". In: *Journal of Parallel and Distributed Computing* 21.1 (Apr. 1994), pp. 4–14.

[5] Matthias Braun et al. "Simple and Efficient Construction of Static Single Assignment Form". In: *Compiler Construction*. Ed. by Ranjit Jhala and Koen Bosschere. Vol. 7791. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6. URL: http://dx.doi.org/10.1007/978-3-642-37051-9_6.

[6] Manuel M T Chakravarty. *Converting a HOAS term GADT into a de Bruijn term GADT*. 2009. URL: http://www.cse.unsw.edu.au/~chak/haskell/term-conv/.

[7] Manuel M. T. Chakravarty, Gabriel C. Ditu and Roman Leshchinskiy. *Instant Generics: Fast and Easy*. http://www.cse.unsw.edu.au/ chak/papers/CDL09.html. 2009.

[8] Manuel M. T. Chakravarty et al. "Associated Types with Class". In: *ACM Symposium on Principles of Programming Languages (POPL '05)*. ACM Press, 2005, pp. 1–13.

[9] Manuel M. T. Chakravarty et al. “Data Parallel Haskell: a status report”. In: *Symposium on Principles of Programming Languages*. 2007, pp. 10–18. DOI: 10.1145/1248648.1248652.

[10] Manuel M.T. Chakravarty and Gabriele Keller. “Functional Array Fusion”. In: *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP ’01)*. ACM Press, 2001.

[11] Manuel M.T. Chakravarty et al. “Accelerating Haskell array codes with multicore GPUs”. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. DAMP ’11. Austin, Texas, USA: ACM, 2011, pp. 3–14. DOI: http://doi.acm.org/10.1145/1926354.1926358.

[12] Olaf Chitil. “Type inference builds a short cut to deforestation”. In: *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP ’99)*. ACM Press, 1999.

[13] Robert Clifton-Everest et al. “Embedding Foreign Code”. In: *PADL ’14: The 16th International Symposium on Practical Aspects of Declarative Languages*. Jan. 2014.

[14] Duncan Coutts, Roman Leshchinskiy and Don Stewart. “Stream Fusion: From Lists to Streams to Nothing at All”. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*. Apr. 2007.

[15] Duncan Coutts, Don Stewart and Roman Leshchinskiy. “Rewriting Haskell Strings”. In: *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*. Springer-Verlag, Jan. 2007, pp. 50–64.

[16] *Fusing filters with integer linear programming*. New York, New York, USA: ACM Request Permissions, Sept. 2014.

[17] Andrew Gill, John Launchbury and Simon L. Peyton Jones. “A short cut to deforestation”. In: *Proceedings of the conference on Functional programming languages and computer architecture*. FPCA ’93. Copenhagen, Denmark: ACM, 1993, pp. 223–232. DOI: http://doi.acm.org/10.1145/165180.165214.

[18] Andy Gill. “Type-Safe Observable Sharing in Haskell”. In: *Haskell ’09: The 2nd ACM SIGPLAN Symposium on Haskell*. Sept. 2009, pp. 117–128.

[19] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std. 1364-2000* (2005).

[20] *Intel Instruction Set Architecture Extensions*. 2014. URL: `https://software.intel.com/en-us/intel-isa-extensions`.

[21] ISO. *Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)*. ISO 9075-1:2011. Geneva, Switzerland: International Organization for Standardization, 2011.

[22] ISO. *Information technology — Document description and processing languages — HyperText Markup Language (HTML)*. ISO 15445-2000. Geneva, Switzerland: International Organization for Standardization, 2000.

[23] Simon Peyton Jones et al. "Simple unification-based type inference for GADTs". In: *ICFP '06: The 11th ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2006, pp. 50–61.

[24] Yukiyoshi Kameyama, Oleg Kiselyov and Chung-chieh Shan. *Combinators for impure yet hygienic code generation*. San Diego, CA, USA: PEPM, Jan. 2014.

[25] Gabriele Keller et al. "Regular, shape-polymorphic, parallel arrays in Haskell". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272. DOI: `http://doi.acm.org/10.1145/1863543.1863582`.

[26] Gabriele Keller et al. "Vectorisation avoidance". In: *Haskell '12: Proceedings of the 2012 Haskell Symposium*. New York, New York, USA: ACM Request Permissions, Jan. 2013, pp. 37–48.

[27] Oleg Kiselyov. "Implementing Explicit and Finding Implicit Sharing in Embedded DSLs". In: *Electronic Proceedings in Theoretical Computer Science* 66 (Sept. 2011), pp. 210–225.

[28] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, 2004.

[29] John Launchbury and Simon L Peyton Jones. "State in Haskell". In: *Lisp and Symbolic Computation* 8.4 (Dec. 1995), pp. 293–341.

[30] *Witnessing Purity, Constancy and Mutability*. 2009.

[31] Ben Lippmeier et al. "Data flow fusion with series expressions in Haskell". In: *ICFP '14*.

[32] Geoffrey Mainland. "Why it's nice to be quoted". In: *Haskell '07: The 2007 ACM SIGPLAN workshop on Haskell*. New York, New York, USA: ACM Press, 2007, p. 73.

[33] Geoffrey Mainland, Roman Leshchinskiy and Simon Peyton Jones. "Exploiting vector instructions with generalized stream fusion". In: *International Conference on Functional Programming 2013*. ACM, Nov. 2013, pp. 37–48.

[34] Trevor L McDonell. *accelerate-llvm*. 2014. URL: `https://github.com/AccelerateHS/accelerate-llvm`.

[35] Trevor L McDonell et al. "Optimising Purely Functional GPU Programs". In: *ICFP '13: The 18th ACM SIGPLAN international conference on Functional programming*. Sept. 2013.

[36] *Plugging haskell in*. 2004.

[37] Simon Peyton Jones. *Call-pattern specialisation for Haskell programs*. Vol. 42. ACM, Oct. 2007.

[38] Simon Peyton Jones. "Haskell 98 language and libraries: the revised report". In: *Journal of Functional Programming* (2003).

[39] Simon Peyton Jones and Simon Marlow. "Secrets of the Glasgow Haskell Compiler inliner". In: *Journal of Functional Programming* (2002), pp. 393–434.

[40] Simon Peyton Jones and Satnam Singh. "A Tutorial on Parallel and Concurrent Programming in Haskell". In: *Lecture Notes in Computer Science*. Springer Verlag, 2008.

[41] Simon Peyton Jones, Andrew Tolmach and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *Haskell Workshop*. Ed. by Ralf Hinze. ACM SIGPLAN, Sept. 2001.

[42] Simon Peyton Jones et al. "Harnessing the Multicores: Nested Data Parallelism in Haskell". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008), IBFI, Schloss Dagstuhl*. 2008.

[43] Simon L Peyton Jones and AndréL M Santos. "A transformation-based optimiser for Haskell". In: *Science of computer programming* 32.1-3 (Sept. 1998), pp. 3–47.

[44] Norman Ramsey, Simon Peyton Jones and João Dias. "Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation". In: *Haskell Symposium*. 2010, pp. 1–14.

[45] Francis P. Russell et al. *DESOLA: an Active Linear Algebra Library Using Delayed Evaluation and Runtime Code Generation*. 2006.

[46] T Sheard and S P Jones. "Template meta-programming for Haskell". In: *... of the 2002 ACM SIGPLAN workshop on Haskell* (2002).

[47] Olin Shivers. "The Anatomy of a Loop". In: *International Conference on Functional Programming*. ACM, Sept. 2005, pp. 2–14.

[48] Don Stewart. "Dynamic Extension of Typed Functional Languages". PhD thesis. University of New South Wales, 2010.

[49] *Dynamic applications from the ground up*. 2005.

[50] Josef Svenningsson. "Shortcut fusion for accumulating parameters & zip-like functions". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. ICFP '02. Pittsburgh, PA, USA: ACM, 2002, pp. 124–132. DOI: `http://doi.acm.org/10.1145/581478.581491`.

[51] Henning Thielemann. "Audio Processing using Haskell". In: *International Conference on Digital Audio Effects (DAFx)*. Naples, Italy, Oct. 2004.

[52] Henning Thielemann. "Compiling Signal Processing Code embedded in Haskell via LLVM". In: *International Conference on Digital Audio Effects (DAFx)*. Graz, Austria, Sept. 2010.

[53] Sjoerd Visscher. *How can I recover sharing in a GADT?* 2012. URL: `http://stackoverflow.com/a/12232127/500700` (visited on 30/11/2014).

[54] Philip Wadler. “Deforestation: transforming programs to eliminate trees”. In: *Special issue of selected papers from 2'nd European Symposium of Programming*. 1990, pp. 231–248.

[55] Richard C Waters. “Automatic transformation of series expressions into loops”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (Jan. 1991), pp. 52–98.

Index