Parallel Programming in Futhark

Release 0.80

Martin Elsman **Troels Henriksen** **Cosmin E. Oancea**

Nov 18, 2018

Department of Computer Science (DIKU)
University of Copenhagen
mael@di.ku.dk
athas@di.ku.dk
cosmin.oancea@di.ku.dk

Contents

Chapter 1

Preface

Welcome to "Parallel Programming in Futhark", an introductory book about the Futhark programming language. Futhark is a data-parallel array programming language that uses the vocabulary of functional programming to provide a parallel programming model that is easy to understand, yet can be compiled to very efficient code by an optimising compiler. Futhark is a *small* language - it is not designed to replace general-purpose languages for application programming. The intended use case is that Futhark is only used for the small but compute-intensive parts of an application, as the Futhark compiler generates code that can be easily called from non-Futhark code. The language was originally developed in Denmark, and is therefore named after the runic alphabet.

This book is written for a reader who already has some programming experience. Prior experience with functional programming is useful, but not required. We will be learning Futhark through small examples that each aim to demonstrate some feature or facet of the language. Furthermore, we will discuss some of the theoretical background of data-parallel programming, as well as elaborate on some of the optimisations that can be expected from the compiler.

1.1 Contributing to the Book

The book is Open Source, maintained on Github, and distributed under the Creative Commons Attribution (By) 4.0 license. All code snippets in the book, including code in the book's repository directory is distributed under the ISC license. We will appreciate pull-requests for fixing any kinds of typos and errors in the text and in the enclosed programs, or making any other improvement. The book's main repository is https://github.com/diku-dk/futhark-book.

1.2 Acknowledgments

This work has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center *HIPERFIT: Functional High*

Performance Computing for Financial Information Technology (hiperfit.dk) under contract number 10-092299. The work has also been supported by Independent Research Fund Denmark as part of the project *Functional Technology for High-performance Architectures (FUTHARK)*.

When citing this work, please use this BibTeX entry.

Chapter 2

Introduction

In 1965, Gordon E. Moore predicted a doubling every year in the number of components in an integrated circuit *[Moo65]*. He revised the prediction in 1975 to a doubling every two year *[Moo75]* and later revisions suggest a slight decrease in the growth rate, while the growth rate, here 50 years after Moore's first prediction, is not seriously predicted to fade out in the next decade. In the first many years, the increase in components per chip area, as predicted by "Moore's law", had a direct influence on processor speed. The personal computer was getting popular and software providers were happy beneficials of the so-called "free lunch", which made programs running on single Central Processing Units (CPUs) double in speed whenever new processors hit the market.

The days of the "free lunches" for sequentially written programs is over. The physical speed limit for sequential processing units has pretty much been reached. Increases in processor clock frequency introduces heat problems that are difficult to deal with and chip providers have instead turned their focus on providing multiple cores in the same chip. Thus, for programs to run faster on ever new architectures, programs will have to make use of algorithms and data structures that benefit from simultaneous, that is *parallel*, execution on multiple cores. Newer architectures, such as Graphical Processing Units (GPUs), host a high number of cores that are designed for parallel processing and over the coming decade, we will see a drastic increase in the number of cores hosted in each chip.

In this book we distinguish between the notions of parallelism and concurrency. By *concurrency*, we refer to programming language controls for coordinating work done by multiple virtual processes. Such processes may in principle run on the same physical processor (using for instance time slicing) or they may run on multiple processors. Controlling the communication and dependencies between multiple processes turns out to be immensely difficult and programmers need to deal with problems such as unforeseen non-determinism and dead-locks, collectively named *race conditions*, issues that emerge when two or more processes (and their interaction with an external environment) interleave. By *parallelism*, on the other hand, we simply refer to the notion of speeding up a program by making it run on multiple processors. Given a program, we can analyze the program to discover dependencies between units of computation and as such, the program contains all the information there is to know about to which degree the program can be executed in parallel.

We emphasize here the notion that a parallel program should result in the same output given an input no-matter how many processors are used for executing the program. On the other hand, we hope that running the program in parallel with multiple processors will execute faster than if only one processor is used. As we shall see, making predictable models for determining whether a given program will run efficiently on a parallel machine can be difficult, in particular in cases where the program is inhomogeneously parallel at several levels, simultaneously.

Parallelism can be divided into the notions of *task parallelism*, which emphasizes the concept of executing multiple independent tasks in parallel, and *data parallelism*, which focuses on executing the same program on a number of different data objects in parallel. At the hardware side, multiple instruction multiple data (MIMD) processor designs, coined after Flynn's taxonomy *[Fly72]*, directly allow for different tasks to be executed in parallel. For such designs, each processor is quite complex and in terms of fitting most processors on a single chip, so as to increase overall throughput, vendors have increasing success with simpler chip designs for which compute units execute single instructions on multiple data (SIMD). Such processor designs have turned out to be useful for a large number of application domains, including graphics processing, machine learning, image analysis, financial algorithms, and many more. In particular, for graphics processing, chip designers have since the 1970'es developed the concept of graphics processing units (GPUs), which, in the later years, have turned into "general purpose" graphics processing units (GPGPUs).

The notions of parallel processing and parallel programming are not new. Concepts in these areas have emerged over a period of more than three decades and today the notion of parallelism appears in many disguises. For example, the internet as we know it can be understood as a giant parallel processing unit and whenever some user is browsing and searching the internet, a large number of processing units are working in parallel to provide the user with the best information available on the topic. At all levels, software engineers need to know how to exploit the ever increasing amount of computational resources.

For many years, programmers and engineers have been accustomed to the simple performance reasoning principles of the von Neumann machine model *[vN45]*, which is also often referred to as the sequential Random Access Machine (RAM) model. With ever more complex chip circuits, introducing speculative instruction scheduling and advanced memory cache hierarchies for leveraging the far from constant-time access to random memory, reasoning about performance has become difficult even for programs running on sequential hardware. The consequence is that, even for sequential programs, programmers and engineers are requesting better models for predicting performance. For programs designed to run on parallel hardware, the situation is often worse. Understanding the performance aspects of executing a task-parallel program on a MIMD architecture can quickly become an immensely complex task in particular because the programmer can be forced to reason about concurrency aspects of the program running on the MIMD architecture. Machines are becoming more complex and the abstractions provided by the simpler machine models seem broken as the models no longer can be used to reason, in a predictable way, about performance. One particular instance of this problem is the assumption in the shared memory PRAM model, which assumes that all processors have constant-time access to random memory.

Low-level languages and frameworks that more or less directly mirror their parallel target architectures include OpenCL *[GHK+11]* and CUDA *[NBGS08]* for data-parallel GPU programming. More abstract approaches to target parallel hardware include library-based approaches, such as

CUBLAS for GPU-targeted linear algebra routines, and annotation-based approaches, such as OpenAcc for targeting GPUs and OpenMP for targeting multi-core platforms.

Instead of requiring programmers to reason about programs based on a particular machine model, an alternative is to base performance reasoning on more abstract *language based cost models*, which are models that emphasize higher-level programming language concepts and functionalities. By introducing such an abstraction layer, programmers will no longer need to "port" their performance reasoning whenever a new parallel machine is targeted. It will instead be up to the language implementor to port the language to new architectures.

The introduction of language based cost models is of course not a silver bullet, but they may help isolate the assumptions under which performance reasoning is made. Guy Blelloch's seminal work on NESL *[Ble90][BHS+94]* introduces a cost model based on the concept of *work*, which, in abstract terms, defines a notion of the total work done by a program, and the concept of *steps*, which defines a notion of the number of dependent parallel steps that the program will take, assuming an infinite number of processors.

In this book we shall make use of a performance cost model for a subset of a data-parallel language and discuss benefits and limitations of the approach. The cost model is based on the language-based cost model developed for NESL, but in contrary to the cost model for NESL, we shall not base our reasoning on an automatic flattening technique for dealing with nested parallelism. Instead, we shall require the programmer to perform certain kinds of flattening manually. The cost model developed for Futhark has been adapted from the cost model developed for the SPARC parallel functional programming language developed for the Carnegie Mellon University (CMU) Fall 2016 course "15-210: Parallel and Sequential Data Structures and Algorithms" *[Org16]*.

We shall primarily look at parallelism from a data-parallel functional programming perspective. The development in the book is made through the introduction of the Futhark data-parallel functional language *[HSE+17][LH17][Hen17][HEO14][HLO16][HO14][HO13]*, which readily will generate GPU-executable code for a Futhark program by compiling the program into a number of OpenCL kernels and coordinating host code for spawning the kernels. Besides the OpenCL backend, Futhark also features a C backend and Futhark has been demonstrated to compile quite complex data-parallel programs into well-performing GPU code *[ABB+16][HDU+16]*.

2.1 Structure of the Book

The book is organised in chapters. In *The Futhark Language*, we introduce the Futhark language, including its basic syntax, the semantics of the core language, and the built-in array second-order array combinators and their parallel semantics. We also describe how to compile and execute Futhark programs using both the sequential C backend and the parallel GPU backend. Finally, we describe Futhark's module system, which allows for programmers to organise code into reusable components that carry no overhead whatsoever, due to Futhark's aggressive strategy of eliminating all module system constructs at compile time *[Els99][EHAO18]*. We also describe Futhark's support for parametric polymorphism and restricted form of higher-order functions, which provide programmers with excellent tooling for writing abstract reusable code.

In *Practical Matters* we discuss various practical matters related to Futhark programming. We discuss techniques for checking the correctness of Futhark programs using unit tests, demonstrate how to debug Futhark programs using the Futhark debugger, show how to benchmark Futhark programs (on both CPU and GPU hardware), and give suggestions of how to resolve issues that may occur when writing programs in Futhark. We also show how to use the Futhark package manager to download libraries of Futhark code.

In *Interoperability*, we describe how Futhark can be used in concert with Python, to develop, for instance, interactive, real time games. We also outline the possibilities for using Futhark in the context of C and .NET programming.

In *A Parallel Cost Model for Futhark Programs*, we introduce an "ideal" cost model for the Futhark language based on the notions of work and span. We present both a type system for an idealized version of Futhark and present a dynamic semantics for the language. The dynamic semantics is used for deriving the notions of work and span.

In *Fusion and List Homomorphisms*, we present to the reader the underlying algebraic reasoning principles that lie behind the Futhark internal fusion technology. We also present to the reader a series of parallel utility functions and demonstrate the usefulness of applying the list-homomorphism theorem *[Bir87]*, which forms the basis of map-reduce reasoning and which, in many cases, turns out to play an important role for implementing efficient data-parallel algorithms in Futhark.

In *Regular Flattening*, we present Futhark's way of dealing with nested regular parallelism. In this chapter, we also introduce a number of segmented operations, including the essential segmented scan operation, which turns out to be central to both Futhark's moderate flattening technique and as a central tool for programmers to flatten irregular data-parallel problems.

Futhark allows for programmers to organise and distribute libraries and applications in Futhark packages, which may be organised, managed, and documented using Futhark's package manager and Futhark's documentation tool. These tools are described in the Futhark User's Guide available at https://futhark.readthedocs.io/en/latest/. In *Pseudo-Random Numbers and Monte Carlo Sampling Methods*, we show how to program with pseudo-random numbers in Futhark using the Futhark package `cpprandom`. This package allows for generating pseudo-random numbers in parallel and further allows the programmer to generate random samples for a number of distributions, including uniform and normal distributions. In the chapter, we also present the Futhark package `sobol`, which allows for generating Sobol numbers efficiently in parallel. This library is useful for stochastic modeling and for Monte Carlo Simulation in high-dimensional spaces.

In *Irregular Flattening*, we describe the necessary tooling and building blocks for implementing irregular data-parallel algorithms in Futhark.

In *Conclusion*, we conclude and give directions for further reading.

Chapter 3

The Futhark Language

Futhark is a pure functional data-parallel array language. It is both syntactically and conceptually similar to established functional languages, such as Haskell and Standard ML. In contrast to these languages, Futhark focuses less on expressivity and elaborate type systems, and more on compilation to high-performance parallel code. Futhark programs are written with bulk operations on arrays, called *Second-Order Array Combinators* (SOACs), that mirror the higher-order functions found in conventional functional languages: `map`, `reduce`, `filter`, and so forth. In Futhark, the parallel SOACs have sequential semantics but permit parallel execution, and will typically be compiled to parallel code.

The primary idea behind Futhark is to design a language that has enough expressive power to conveniently express complex programs, yet is also amenable to aggressive optimisation and parallelisation. The tension is that as the expressive power of a language grows, the difficulty of efficient compilation rises likewise. For example, Futhark supports nested parallelism, despite the complexities of efficiently mapping it to the flat parallelism supported by hardware, as many algorithms are awkward to write with just flat parallelism. On the other hand, we do not support non-regular arrays, as they complicate size analysis a great deal. The fact that Futhark is purely functional is intended to give an optimising compiler more leeway in rearranging the code and performing high-level optimisations.

Programming in Futhark feels similar to programming in other functional languages. If you know Haskell or Standard ML, you will likely be able to read and modify most Futhark code. For example, this program computes the dot product $\Sigma_i x_i \cdot y_i$ of two vectors of integers:

```
let main (x: []i32) (y: []i32): i32 =
  reduce (+) 0 (map2 (*) x y)
```

In Futhark, the notation for an array of element type `t` is `[]t`. The program defines a function called `main` that takes two arguments, both integer arrays, and returns an integer. The `main` function first computes the element-wise product of its two arguments, resulting in an array of integers, then computes the product of the elements in this new array.

If we save the program in a file `dotprod.fut`, then we can compile it to a binary `dotprod` (or `dotprod.exe` on Windows) by running:

```
$ futhark-c dotprod.fut
```

A Futhark program compiled to an executable will read the arguments to its `main` function from standard input, and will print the result to standard output:

```
$ echo [2,2,3] [4,5,6] | ./dotprod
36i32
```

In Futhark, an array literal is written with square brackets surrounding a comma-separated sequence of elements. Integer literals can be suffixed with a specific type. This is why `dotprod` prints `36i32`, rather than just `36` - this makes it clear that the result is a 32-bit integer. Later we will see examples of when these suffixes are useful.

The `futhark-c` compiler we used above translates a Futhark program into sequential code running on the CPU. This can be useful for testing, and will work on most systems, even those without GPUs. However, it wastes the main potential of Futhark: fast parallel execution. We can instead use the `futhark-opencl` compiler to generate an executable that offloads execution via the OpenCL framework. In principle, this allows offloading to any kind of device, but the `futhark-opencl` compilation pipelines makes optimisation assumptions that are oriented towards contemporary GPUs. Use of `futhark-opencl` is simple, assuming your system has a working OpenCL setup:

```
$ futhark-opencl dotprod.fut
```

Execution is just as before:

```
$ echo [2,2,3] [4,5,6] | ./dotprod
36i32
```

In this case, the workload is small enough that there is little benefit in parallelising the execution. In fact, it is likely that for this tiny dataset, the OpenCL startup overhead results in several orders of magnitude slowdown over sequential execution. See *Benchmarking* for information on how to measure execution times.

The ability to compile Futhark programs to executables is useful for testing, but it should be noted that it is not how Futhark is intended to be used in practice. As a pure functional array language, Futhark is not capable of reading input or managing a user interface, and as such cannot be used as a general-purpose language. Futhark is intended to be used for small, performance-sensitive parts of larger applications, typically by compiling a Futhark program to a *library* that can be imported and used by applications written in conventional languages. See *Interoperability* for more information.

As compiled Futhark executables are intended for testing, they take a range of command line options to manipulate their behaviour and print debugging information. These will be introduced as needed.

For most of this book, we will be making use of the interactive Futhark *interpreter*: `futharki`. When launched with no options, it provides a Futhark REPL into which you can enter arbitrary expressions and declarations:

```
$ futharki
|// |\    |   |\  |\   /
|/  | \   |\  |\  |/  /
|   |  \  |/  |   |\  \
|   |   \ |   |   | \  \
Version 0.7.0.
Copyright (C) DIKU, University of Copenhagen, released under the ISC␣
↪license.

Run :help for a list of commands.

[0]> 1 + 2
3i32
[1]>
```

The prompts are numbered to permit error messages to refer to previous inputs. We will generally elide the numbers in this book, and just write the prompt as > (do not confuse this with the Unix prompt, which we write as `$`).

`futharki` supports a variety of commands for inspecting and debugging Futhark code. These will be introduced as necessary, in particular in *Testing and Debugging*.

3.1 Basic Language Features

As a functional or *value-oriented* language, the semantics of Futhark can be understood entirely by how values are constructed, and how expressions transform one value to another. As a statically typed language, all Futhark values are classified by their *type*. The primitive types in Futhark are the signed integer types `i8`, `i16`, `i32`, `i64`, the unsigned integer types `u8`, `u16`, `u32`, `u64`, the floating-point types `f32`, `f64`, and the boolean type `bool`. An `f32` is always a single-precision float and a `f64` is a double-precision float.

Numeric literals can be suffixed with their intended type. For example, `42i8` is of type `i8`, and `1337e2f64` is of type `f64`. If no suffix is given, the type is inferred by the context. In case of ambiguity, integral literals are given type `i32` and decimal literals are given `f64`. Boolean literals are written as `true` and `false`.

Note: converting between primitive values

Futhark provides a collection of functions for performing straightforward conversions between primitive types. These are all of the form `to.from`. For example, `i32.f64` converts a value of

type `f64` (double-precision float) to a value of type `i32` (32-bit signed integer), by truncating the fractional part:

```
> i32.f64 2.1
2

> f64.i32 2
2.0
```

Technically, `i32.f64` is not the name of the function. Rather, this is a reference to the function `f64` in the module `i32`. We will not discuss modules further until *Modules*, so for now it suffices to think of `i32.f64` as a function name. The only wrinkle is that if a variable with the name `i32` is in scope, the entire `i32` module becomes inaccessible by shadowing.

Futhark provides shorthand for the most common conversions:

```
r32 == f32.i32
t32 == i32.f32
r64 == f64.i32
t64 == i64.f32
```

All values can be combined in tuples and arrays. A tuple value or type is written as a sequence of comma-separated values or types enclosed in parentheses. For example, `(0, 1)` is a tuple value of type `(i32,i32)`. The elements of a tuple need not have the same type – the value `(false, 1, 2.0)` is of type `(bool, i32, f64)`. A tuple element can also be another tuple, as in `((1,2),(3,4))`, which is of type `((i32,i32),(i32,i32))`. A tuple cannot have just one element, but empty tuples are permitted, although they are not very useful — these are written `()` and are of type `()`. *Records* exist as syntactic sugar on top of tuples, and will be discussed in *Records*.

An array value is written as a sequence of comma-separated values enclosed in square brackets: `[1,2,3]`. An array type is written as `[d]t`, where `t` is the element type of the array, and `d` is an integer indicating the size. We often elide `d`, in which case the size will be inferred. As an example, an array of three integers could be written as `[1,2,3]`, and has type `[3]i32`. An empty array is written simply as `[]`, although the context must make the type of an empty array unambiguous.

Multi-dimensional arrays are supported in Futhark, but they must be *regular*, meaning that all inner arrays have the same shape. For example, `[[1,2], [3,4], [5,6]]` is a valid array of type `[3][2]i32`, but `[[1,2], [3,4,5], [6,7]]` is not, because there we cannot determine integers `m` and `n` such that `[m][n]i32` is the type of the array. The restriction to regular arrays is rooted in low-level concerns about efficient compilation, but we can understand it in language terms by the inability to write a type with consistent dimension sizes for an irregular array value. In a Futhark program, all array values, including intermediate (unnamed) arrays, must be typeable. We will return to the implications of this restriction in later chapters.

3.1.1 Simple Expressions

The Futhark expression syntax is mostly conventional ML-derived syntax, and supports the usual binary and unary operators, with few surprises. Futhark does not have syntactically significant indentation, so feel free to put white space whenever you like. This section will not try to cover the entire Futhark expression language in complete detail. See the reference manual for a comprehensive treatment.

Function application is via juxtaposition. For example, to apply a function `f` to a constant argument, we write:

```
f 1.0
```

See *Top-Level Definitions* for how to declare your own functions.

A let-expression can be used to give a name to the result of an expression:

```
let z = x + y
in body
```

Futhark is eagerly evaluated (unlike Haskell), so the expression for `z` will be fully evaluated before `body`. The keyword `in` is optional when it precedes another `let`. Thus, instead of writing:

```
let a = 0 in
let b = 1 in
let c = 2 in
a + b + c
```

we can write

```
let a = 0
let b = 1
let c = 2
in a + b + c
```

The final `in` is still necessary. In examples, we will often skip the body of a let-expression if it is not important. A limited amount of pattern matching is supported in let-bindings, which permits tuple components to be extracted:

```
let (x,y) = e       -- e must be of some type (t1,t2)
```

This feature also demonstrates the Futhark line comment syntax — two dashes followed by a space. Block comments are not supported.

A two-way if-then-else is the only branching construct in Futhark:

```
if x < 0 then -x else x
```

Arrays are indexed using the common row-major notation, as in the expression `a[i1, i2, i3, ...]`. All array accesses are checked at runtime, and the program will terminate abnormally if an invalid access is attempted.

White space is used to disambiguate indexing from application to array literals. For example, the expression `a b [i]` means "apply the function `a` to the arguments `b` and `[i]`", while `a b[i]` means "apply the function `a` to the argument `b[i]`".

Futhark also supports array *slices*. The expression `a[i:j:s]` returns a slice of the array `a` from index `i` (inclusive) to `j` (exclusive) with a stride of `s`. Slicing of multiple dimensions can be done by separating with commas, and may be intermixed freely with indexing.

If the stride is positive, then `i <= j` must hold, and if the stride is negative, then `j <= i` must hold.

Some syntactic sugar is provided for concisely specifying arrays of intervals of integers. The expression `x...y` produces an array of the integers from `x` to `y`, both inclusive. The upper bound can be made exclusive by writing `x..<y`. For example:

```
> 1...3
[1i32, 2i32, 3i32]
> 1..<3
[1i32, 2i32]
```

It is usually necessary to enclose a range expression in parentheses, because they bind very loosely. A stride can be provided by writing `x..y...z`, with the interpretation "first `x`, then `y`, up to `z`". For example:

```
> 1..3...7
[1i32, 3i32, 5i32, 7i32]
> (1..3..<7)
[1i32, 3i32, 5i32]
```

The element type of the produced array is the same as the type of the integers used to specify the bounds, which must all have the same type (but need not be constants). We will be making frequent use of this notation throughout this book.

Note: structural equality

The Futhark equality and inequality operators `==` and `!=` are overloaded operators, just like `+`. They work for types built from basic types (e.g., `i32`), array types, tuple types, and record types. The operators are not allowed on values containing sub-values of abstract types or function types.

Notice that Futhark does not support a notion of type classes *[PJ93]* or equality types *[Els98]*. Allowing the equality and inequality operators to work on values of abstract types could potentially violate abstraction properties, which is the reason for the special treatment of equality types and equality type variables in the Standard ML programming language.

3.1.2 Top-Level Definitions

A Futhark program consists of a sequence of top-level definitions, which are primarily *function definitions* and *value definitions*. A function definition has the following form:

```
let name params... : return_type = body
```

A function may optionally declare its return type and the types of its parameters. If type annotations are not provided, the types are inferred. As a concrete example, here is the definition of the Mandelbrot set iteration step $Z_{n+1} = Z_n^2 + C$, where Z_n is the actual iteration value, and C is the initial point. In this example, all operations on complex numbers are written as operations on pairs of numbers. In practice, we would use a library for complex numbers.

```
let mandelbrot_step ((Zn_r, Zn_i): (f64, f64))
                    ((C_r, C_i): (f64, f64))
                  : (f64, f64) =
  let real_part = Zn_r*Zn_r - Zn_i*Zn_i + C_r
  let imag_part = 2.0*Zn_r*Zn_i + C_i
  in (real_part, imag_part)
```

Or equivalently, without specifying the types:

```
let mandelbrot_step (Zn_r, Zn_i)
                    (C_r, C_i) =
  let real_part = Zn_r*Zn_r - Zn_i*Zn_i + C_r
  let imag_part = 2.0*Zn_r*Zn_i + C_i
  in (real_part, imag_part)
```

It is generally considered good style to specify the types of the parameters and the return value when defining top-level functions. Type inference is mostly used for local and anonymous functions, that we will get to later.

We can define a constant with very similar notation:

```
let name: value_type = definition
```

For example:

```
let physicists_pi: f64 = 4.0
```

Top-level definitions are declared in order, and a definition may refer *only* to those names that have been defined before it occurs. This means that circular and recursive definitions are not permitted. We will return to function definitions in *Size Annotations* and *Parametric Polymorphism*, where we will look at more advanced features, such as parametric polymorphism and implicit size parameters.

Note: Loading files into `futharki`

At this point you may want to start writing and applying functions. It is possible to do this directly in `futharki`, but it quickly becomes awkward for multi-line functions. You can use the `:load` command to read declarations from a file:

```
> :load test.fut
Loading test.fut
```

The `:load` command will remove any previously entered declarations and provide you with a clean slate. You can reload the file by running `:load` without further arguments:

```
> :load
Loading test.fut
```

Emacs users may want to consider futhark-mode, which is able to load the file being edited into `futharki` with `C-c C-l`, and provides other useful features as well.

Exercise: Simple Futhark programming

This is a good time to make sure you can actually write and run a Futhark program on your system. Write a program that contains a function `main` that accepts as input a parameter `x : i32`, and returns `x` if `x` is positive, and otherwise the negation of `x`. Compile your program with `futhark-c` and verify that it works, then try with `futhark-opencl`.

Type abbreviations

The previous definition of `mandelbrot_step` accepted arguments and produced results of type `(f64,f64)`, with the implied understanding that such pairs of floats represent complex numbers. To make this clearer, and thus improve the readability of the function, we can use a *type abbreviation* to define a type `complex`:

```
type complex = (f64, f64)
```

We can now define `mandelbrot_step` as follows:

```
let mandelbrot_step ((Zn_r, Zn_i): complex)
                    ((C_r, C_i): complex)
                  : complex =
    let real_part = Zn_r*Zn_r - Zn_i*Zn_i + C_r
    let imag_part = 2.0*Zn_r*Zn_i + C_i
    in (real_part, imag_part)
```

Type abbreviations are purely a syntactic convenience — the type `complex` is fully interchangeable with the type `(f64, f64)`:

```
> type complex = (f64, f64)
> let f (x: (f64, f64)): complex = x
> f (1,2)
(1.0f64, 2.0f64)
```

For abstract types, that hide their definition, we have to use the module system discussed in *Modules*.

3.2 Array Operations

Futhark provides various combinators for performing bulk transformations of arrays. Judicious use of these combinators is key to getting good performance. There are two overall categories: *first-order array combinators*, like `zip`, that always perform the same operation, and *second-order array combinators* (*SOAC*s), like `map`, that take a *functional argument* indicating the operation to perform. SOACs are the basic parallel building blocks of Futhark programming. While they are designed to resemble familiar higher-order functions from other functional languages, they have some restrictions to enable efficient parallel execution.

We can use `zip` to transform two arrays to a single array of pairs:

```
> zip [1,2,3] [true,false,true]
[(1i32, true), (2i32, false), (3i32, true)]
```

Notice that the input arrays may have different types. We can use `unzip` to perform the inverse transformation:

```
> unzip [(1,true),(2,false),(3,true)]
([1i32, 2i32, 3i32], [true, false, true])
```

Be aware that `zip` requires all input arrays to have the same length. This is checked at runtime. Transforming between arrays of tuples and tuples of arrays is common in Futhark programs, as many array operations accept only one array as input. Due to a clever implementation technique, `zip` and `unzip` usually have no runtime cost (they are fused into other operations), so you should not shy away from using them out of efficiency concerns. For operating on arrays of tuples with more than two elements, there are `zip`/`unzip` variants called `zip3`, `zip4`, etc, up to `zip8`/`unzip8`.

Now let's take a look at some SOACs.

3.2.1 Map

The simplest SOAC is probably `map`. It takes two arguments: a function and an array. The function argument can be a function name, or an anonymous function. The function is applied to every element of the input array, and an array of the result is returned. For example:

```
> map (\x -> x + 2) [1,2,3]
[3i32, 4i32, 5i32]
```

Anonymous functions need not define their parameter- or return types, but you are free to do so in cases where it aids readability:

```
> map (\(x:i32): i32 -> x + 2) [1,2,3]
[3i32, 4i32, 5i32]
```

The functional argument can also be an operator, which must be enclosed in parentheses:

```
> map (!) [true, false, true]
[false, true, false]
```

Partially applying operators is also supported using so-called *operator sections*, with a syntax taken from Haskell:

```
> map (+2) [1,2,3]
[3i32, 4i32, 5i32]

> map (2-) [1,2,3]
[1i32, 0i32, -1i32]
```

However, note that the following will *not* work:

```
[0]> map (-2) [1,2,3]
Error at [0]> :1:5-1:8:
Cannot unify `t2' with type `a0 -> x1' (must be one of i8, i16, i32,␣
↪i64, u8, u16, u32, u64, f32, f64 due to use at [0]> :1:7-1:7).
When matching type
  a0 -> x1
with
  t2
```

This is because the expression `(-2)` is taken as negative number `-2` encloses in parentheses. Instead, we have to write it with an explicit lambda:

```
> map (\x -> x-2) [1,2,3]
[-1i32, 0i32, 1i32]
```

There are variants of `map`, suffixed with an integer, that permit simultaneous mapping of multiple arrays, which must all have the same size. This is supported up to `map5`. For example, we can perform an element-wise sum of two arrays:

```
> map2 (+) [1,2,3] [4,5,6]
[5i32, 7i32, 9i32]
```

A combination of `map` and `zip` can be used to handle arbitrary numbers of simultaneous arrays.

Be careful when writing `map` expressions where the function returns an array. Futhark requires regular arrays, so this is unlikely to go well:

```
map (\n -> 1...n) ns
```

Unless the array `ns` consists of identical values, this expression will fail at runtime.

We can use `map` to duplicate many other language constructs. For example, if we have two arrays `xs:[n]i32` and `ys:[m]i32` — that is, two integer arrays of sizes `n` and `m` — we can concatenate them using:

```
map (\i -> if i < n then xs[i] else ys[i-n])
    (0..<n+m)
```

However, it is not a good idea to write code like this, as it hinders the compiler from using high-level properties to do optimisation. Using `map` with explicit indexing is usually only necessary when solving complicated irregular problems that cannot be represented directly.

3.2.2 Scan and Reduce

While `map` is an array transformer, the `reduce` SOAC is an array aggregator: it uses some function of type `t -> t -> t` to combine the elements of an array of type `[]t` to a value of type `t`. In order to perform this aggregation in parallel, the function must be *associative* and have a *neutral element* (in algebraic terms, constitute a monoid):

- A function f is associative if $f(x, f(y, z)) = f(f(x, y), z)$ for all x, y, z.
- A function f has a neutral element e if $f(x, e) = f(e, x) = x$ for all x.

Many common mathematical operators fulfill these laws, such as addition: $(x+y)+z = x+(y+z)$ and $x + 0 = 0 + x = x$. But others, like subtraction, do not. In Futhark, we can use the addition operator and its neutral element to compute the sum of an array of integers:

```
> reduce (+) 0 [1,2,3]
6i32
```

It turns out that combining `map` and `reduce` is both powerful and has remarkable optimisation properties, as we will discuss in *Fusion and List Homomorphisms*. Many Futhark programs are primarily `map-reduce` compositions. For example, we can define a function to compute the dot product of two vectors of integers:

```
let dotprod (xs: []i32) (ys: []i32): i32 =
  reduce (+) 0 (map2 (*) xs ys)
```

A close cousin of `reduce` is `scan`, often called *generalised prefix sum*. Where `reduce` produces just one result, `scan` produces one result for every prefix of the input array. This is perhaps best understood with an example:

```
scan (+) 0 [1,2,3] == [0+1, 0+1+2, 0+1+2+3] == [1, 3, 6]
```

Intuitively, the result of `scan` is an array of the results of calling `reduce` on increasing prefixes of the input array. The last element of the returned array is equivalent to the result of calling `reduce`. Like with `reduce`, the operator given to `scan` must be associative and have a neutral element.

There are two main ways to compute scans: *exclusive* and *inclusive*. The difference is that the empty prefix is considered in an exclusive scan, but not in an inclusive scan. Computing the exclusive +-scan of `[1,2,3]` thus gives `[0,1,3]`, while the inclusive +-scan is `[1,3,6]`. The `scan` in Futhark is inclusive, but it is easy to generate a corresponding exclusive scan simply by prepending the neutral element and removing the last element.

While the idea behind `reduce` is probably familiar, `scan` is a little more esoteric, and mostly has applications for handling problems that do not seem parallel at first glance. Several examples are discussed in the following chapters.

3.2.3 Filtering

We have seen `map`, which permits us to change all the elements of an array, and we have seen `reduce`, which lets us collapse all the elements of an array. But we still need something that lets us remove some, but not all, of the elements of an array. This SOAC is `filter`, which keeps only those elements of an array that satisfy some predicate.

```
> filter (<3) [1,5,2,3,4]
[1i32, 2i32]
```

The use of `filter` is mostly straightforward, but there are some patterns that may appear subtle at first glance. For example, how do we find the *indices* of all nonzero entries in an array of integers? Finding the values is simple enough:

```
> filter (!=0) [0,5,2,0,1]
[5i32, 2i32, 1i32]
```

But what are the corresponding indices? We can solve this using a combination of `zip`, `filter`, and `unzip`:

```
> let indices_of_nonzero (xs: []i32): []i32 =
    let n = length xs
    let xs_and_is = zip xs (0..<n)
    let xs_and_is' = filter (\(x,_) -> x != 0) xs_and_is
    let (_, is') = unzip xs_and_is'
    in is'
```

(continues on next page)

(continued from previous page)

```
> indices_of_nonzero [1, 0, -2, 4, 0, 0]
[0i32, 2i32, 3i32]
```

Be aware that `filter` is a somewhat expensive SOAC, corresponding roughly to a `scan` plus a `map`.

The idiom `0..<n` for constructing an array of the valid indices into an array of size `n` is so common that a predefined library function `iota` exists for this purpose:

```
> iota 5
[0i32, 1i32, 2i32, 3i32, 4i32]
```

The term `iota` is inherited from APL, where the corresponding operation is written with an actual ι (greek letter).

3.2.4 Sequential Loops

Futhark does not directly support recursive functions, but instead provides syntactical sugar for expressing the equivalent of certain tail-recursive functions. Consider the following hypothetical tail-recursive formulation of a function for computing the Fibonacci numbers

```
let fibhelper(x: i32, y: i32, n: i32): i32 =
  if n == 1 then x else fibhelper(y, x+y, n-1)

let fib(n: i32): i32 = fibhelper(1,1,n)
```

We cannot write this directly in Futhark, but we can express the same idea using the `loop` construct:

```
let fib(n: i32): i32 =
  let (x, _) = loop (x, y) = (1,1) for i < n do (y, x+y)
  in x
```

The semantics of this loop is precisely as in the tail-recursive function formulation. In general, a loop

```
loop pat = initial for i < bound do loopbody
```

has the following semantics:

1. Bind `pat` to the initial values given in `initial`.
2. While `i < bound`, evaluate `loopbody`, rebinding `pat` to be the value returned by the body. At the end of each iteration, increment `i` by one.
3. Return the final value of `pat`.

Semantically, a loop-expression is completely equivalent to a call to its corresponding tail-recursive function.

For example, denoting by `t` the type of `x`, the loop

```
loop x = a for i < n do
  g(x)
```

has the semantics of a call to the following tail-recursive function:

```
let f(i: i32, n: i32, x: t): t =
  if i >= n then x
  else f(i+1, n, g(x))

-- the call
let x = f(i, n, a)
in body
```

The syntax shown above is actually just syntactical sugar for a common special case of a *for-in* loop over an integer range, which is written as:

```
loop pat = initial for xpat in xs do loopbody
```

Here, `xpat` is an arbitrary pattern that matches an element of the array `xs`. For example:

```
loop acc = 0 for (x,y) in zip xs ys do
  acc + x * y
```

The purpose of the loop syntax is partly to render some sequential computations slightly more convenient, but primarily to express certain very specific forms of recursive functions, specifically those with a fixed iteration count. This property is used for analysis and optimisation by the Futhark compiler. In contrast to most functional languages, Futhark does not properly support recursion, and users are therefore required to use the loop syntax for sequential loops.

Apart from `for`-loops, Futhark also supports `while`-loops. These loops do not provide as much information to the compiler, but can be used for convergence loops, where the number of iterations cannot be predicted in advance. For example, the following program doubles a given number until it exceeds a given threshold value:

```
let main(x: i32, bound: i32): i32 =
  loop x while x < bound do x * 2
```

In all respects other than termination criteria, `while`-loops behave identically to `for`-loops.

For brevity, the initial value expression can be elided, in which case an expression equivalent to the pattern is implied. This feature is easier to understand with an example. The loop

```
let fib(n: i32): i32 =
  let x = 1
  let y = 1
  let (x, _) = loop (x, y) = (x, y) for i < n do (y, x+y)
  in x
```

can also be written:

```
let fib(n: i32): i32 =
  let x = 1
  let y = 1
  let (x, _) = loop (x, y) for i < n do (y, x+y)
  in x
```

This style of code can sometimes make imperative code look more natural.

Note: Type-checking with `futharki`

If you are uncertain about the type of some Futhark expression, the `:type` command (or `:t` for short) can help. For example:

```
> :t 2
2 : i32

> :t (+2)
(+ 2) : i32 -> i32
```

You will also be informed if the expression is ill-typed:

```
[1]> :t true : i32
Error at [1]> :1:1-1:10:
Couldn't match expected type `i32' with actual type `bool'.
When matching type
  i32
with
  bool
```

3.3 In-Place Updates

While Futhark is an uncompromisingly pure functional language, it may occasionally prove useful to express certain algorithms in an imperative style. Consider a function for computing the n first Fibonacci numbers:

```
let fib (n: i32): []i32 =
  -- Create "empty" array.
  let arr = replicate n 0
  -- Fill array with Fibonacci numbers.
  in loop (arr) for i < n-2 do
       arr with [i+2] = arr[i] + arr[i+1]
```

The notation `arr with [i+2] = arr[i] + arr[i+1]` produces an array equivalent to `arr`, but with a new value for the element at position `i+2`. A shorthand syntax is available for the (common) case where we immediately bind the array to a variable of the same name:

```
let arr = arr with [i+2] = arr[i] + arr[i+1]

-- Can be shortened to:

let arr[i+2] = arr[i] + arr[i+1]
```

If the array `arr` were to be copied for each iteration of the loop, we would spend a lot of time moving around data, even though it is clear in this case that the "old" value of `arr` will never be used again. Precisely, what should be an algorithm with complexity $O(n)$ would become $O(n^2)$, due to copying the size n array (an $O(n)$ operation) for each of the n iterations of the loop.

To prevent this copying, Futhark updates the array *in-place*, that is, with a static guarantee that the operation will not require any additional memory allocation, or copying the array. An *in-place update* can modify the array in time proportional to the elements being updated ($O(1)$ in the case of the Fibonacci function), rather than time proportional to the size of the final array, as would the case if we perform a copy. In order to perform the update without violating referential transparency, Futhark must know that no other references to the array exists, or at least that such references will not be used on any execution path following the in-place update.

In Futhark, this is done through a type system feature called *uniqueness types*, similar to, although simpler than, the uniqueness types of the programming language Clean. Alongside a (relatively) simple aliasing analysis in the type checker, this extension is sufficient to determine at compile time whether an in-place modification is safe, and signal a compile time error if in-place updates are used in a way where safety cannot be guaranteed.

The simplest way to introduce uniqueness types is through examples. To that end, let us consider the following function definition.

```
let modify (a: *[]i32) (i: i32) (x: i32): *[]i32 =
  a with [i] = a[i] + x
```

The function call `modify a i x` returns a, but where the element at index `i` has been increased by x. Notice the asterisks: in the parameter declaration `(a: *[i32])`, the asterisk means that the function `modify` has been given "ownership" of the array a, meaning that any caller of `modify` will never reference array a after the call again. In particular, `modify` can change the element at index `i` without first copying the array, i.e. `modify` is free to do an in-place

modification. Furthermore, the return value of `modify` is also unique - this means that the result of the call to `modify` does not share elements with any other visible variables.

Let us consider a call to `modify`, which might look as follows.

```
let b = modify a i x
```

Under which circumstances is this call valid? Two things must hold:

1. The type of `a` must be `*[]i32`, of course.
2. Neither `a` or any variable that *aliases* `a` may be used on any execution path following the call to `modify`.

When a value is passed as a unique-typed argument in a function call, we say that the value is *consumed*, and neither it nor any of its *aliases* (see below) can be used again. Otherwise, we would break the contract that gives the function liberty to manipulate the argument however it wants. Notice that it is the type in the argument declaration that must be unique - it is permissible to pass a unique-typed variable as a non-unique argument (that is, a unique type is a subtype of the corresponding nonunique type).

A variable v aliases a if they may share some elements, for instance by an overlap in memory. As the most trivial case, after evaluating the binding `b = a`, the variable `b` will alias `a`. As another example, if we extract a row from a two-dimensional array, the row will alias its source:

```
let b = a[0] -- b is aliased to a
             -- (assuming a is not one-dimensional)
```

Most array combinators produce fresh arrays that initially alias no other arrays in the program. In particular, the result of `map f a` does not alias `a`. One exception is array slicing, where the result is aliased to the original array.

Let us consider the definition of a function returning a unique array:

```
let f(a: []i32): *[]i32 = e
```

Notice that the argument, `a`, is non-unique, and hence we cannot modify it inside the function. There is another restriction as well: `a` must not be aliased to our return value, as the uniqueness contract requires us to ensure that there are no other references to the unique return value. This requirement would be violated if we permitted the return value in a unique-returning function to alias its (non-unique) parameters.

To summarise: *values are consumed by being the source in a in-place binding, or by being passed as a unique parameter in a function call*. We can crystallise valid usage in the form of three principal rules:

Uniqueness Rule 1 When a value is consumed — for example, by being passed in the place of a unique parameter in a function call, or used as the source in a in-place expression, neither that value, nor any value that aliases it, may be used on any execution path following the function call. A violation of this rule is as follows:

```
let b = a with [i] = 2 in -- Consumes 'a'
f(b,a) -- Error: a used after being consumed
```

Uniqueness Rule 2 If a function definition is declared to return a unique value, the return value (that is, the result of the body of the function) must not share memory with any non-unique arguments to the function. As a consequence, at the time of execution, the result of a call to the function is the only reference to that value. A violation of this rule is as follows:

```
let broken (a: [][]i32, i: i32): *[]i32 =
  a[i] -- Error: Return value aliased with 'a'.
```

Uniqueness Rule 3 If a function call yields a unique return value, the caller has exclusive access to that value. At *the point the call returns*, the return value may not share memory with any variable used in any execution path following the function call. This rule is particularly subtle, but can be considered a rephrasing of Uniqueness Rule 2 from the "calling side".

It is worth emphasising that everything related to uniqueness types is implemented as a static analysis. *All* violations of the uniqueness rules will be discovered at compile time (during type-checking), leaving the code generator and runtime system at liberty to exploit them for low-level optimisation.

3.3.1 When To Use In-Place Updates

If you are used to programming in impure languages, in-place updates may seem a natural and convenient tool that you may use frequently. However, Futhark is a functional array language, and should be used as such. In-place updates are restricted to simple cases that the compiler is able to analyze, and should only be used when absolutely necessary. Most Futhark programs are written without making use of in-place updates at all.

Typically, we use in-place updates to efficiently express sequential algorithms that are then mapped on some array. Somewhat counter-intuitively, however, in-place updates can also be used for expressing irregular nested parallel algorithms (which are otherwise not expressible in Futhark), albeit in a low-level way. The key here is the array combinator `scatter`, which writes to several positions in an array in parallel. Suppose we have an array `is` of type `[n]i32`, an array `vs` of type `[n]t` (for some `t`), and an array `as` of type `[m]t`. Then the expression `scatter as is vs` morally computes

```
for i in 0..n-1:
  j = is[i]
  v = vs[i]
  as[j] = v
```

and returns the modified `as` array. The old `as` array is marked as consumed and may not be used anymore. Parallel `scatter` can be used, for instance, to implement efficiently the radix sort algorithm, as demonstrated in *Radix-Sort in Futhark*.

3.4 Size Annotations

Functions on arrays typically impose constraints on the shape of their parameters, and often the shape of the result depends on the shape of the parameters. Futhark provides a language construct called *size annotations*, that give the programmer the option of encoding these properties directly into the type of a function. Consider first the trivial case of a function that packs a single `i32` value in an array:

```
let singleton (x: i32): [1]i32 = [x]
```

We explicitly annotate the return type to state that this function returns a single-element array.

For expressing constraints among the sizes of the parameters, Futhark provides *size parameters*. Consider the definition of dot product we have used so far:

```
let dotprod (xs: []i32) (ys: []i32): i32 =
  reduce (+) 0 (map2 (*) xs ys)
```

The `dotprod` function assumes that the two input arrays have the same size, or else the `map2` will fail. However, this constraint is not visible in the type of the function. Size parameters allow us to make this explicit:

```
let dotprod [n] (xs: [n]i32) (ys: [n]i32): i32 =
  reduce (+) 0 (map2 (*) xs ys)
```

The `[n]` preceding the *value parameters* (`xs` and `ys`) is called a *size parameter*, which lets us assign a name to the dimensions of the value parameters. A size parameter must be used at least once in the type of a value parameter, so that a concrete value for the size parameter can be determined at runtime. Size parameters are *implicit*, and need not an explicit argument when the function is called. For example, the `dotprod` function can be used as follows:

```
> dotprod [1,2] [3,4]
11i32
```

A size parameter is in scope in both the body of a function and its return type, which we can use, for instance, for defining a function for computing averages:

```
let average [n] (xs: [n]f64): f64 =
  reduce (+) 0 xs / f64 n
```

Size parameters are always of type `i32`, and in fact, *any* `i32`-typed variable in scope can be used as a size annotation. This feature lets us define a function that replicates an integer some number of times:

```
let replicate_i32 (n: i32) (x: i32): [n]i32 =
  map (\_ -> x) (0..<n)
```

In *Parametric Polymorphism* we will see how to write a polymorphic `replicate` function that works for any type.

As a more complicated example of using size parameters, consider multiplying two matrices `x` and `y`. This is only defined if the number of columns in `x` equals the number of rows in `y`. In Futhark, we can encode this as follows:

```
let matmult [n][m][p] (x: [n][m]i32, y: [m][p]i32): [n][p]i32 =
  map (\xr -> map (dotprod xr) (transpose y)) x
```

Three sizes are involved, `n`, `m`, and `p`. We indicate that the number of columns in `x` must match the number of columns in `y`, and that the size of the returned matrix has the same number of rows as `x`, and the same number of columns as `y`.

Be aware that size annotations are checked dynamically, not statically. Whenever we call a function or return a value, an error is raised if its size does not match the annotations. However, nothing prevents th following expression from passing the type checker:

```
> :t dotprod [1,2] [1,2,3]
dotprod [1, 2] [1, 2, 3] : i32
```

Although it will fail if actually executed:

```
[1]> dotprod [1,2] [1,2,3]
Error at [1]> :1:1-1:21 -> [35]> :1:35-1:44: Size annotation 2 does␣
↪not match observed size 3.
```

Presently, only variables and constants are legal as size annotations. This restriction means that the following function definition is not valid:

```
let doubleup [n] (xs: [n]i32): [2*n]i32 =
  map (\i -> xs[i/2]) (0..<n*2)
```

While size annotations are a simple and limited mechanism, they can help make hidden invariants visible to users of your code. In some cases, size annotations also help the compiler generate better code, as it becomes clear which arrays are supposed to have the same size, and lets the compiler hoist out checking as far as possible.

Size parameters are also permitted in type abbreviations. As an example, consider a type abbreviation for a vector of integers:

```
type intvec [n] = [n]i32
```

We can now use `intvec [n]` to refer to integer vectors of size `n`:

```
let x: intvec [3] = [1,2,3]
```

A type parameter can be used multiple times on the right-hand side of the definition; perhaps to define an abbreviation for square matrices:

```
type sqmat [n] = [n][n]i32
```

The brackets surrounding `[n]` and `[3]` are part of the notation, not the parameter itself, and are used for disambiguating size parameters from the *type parameters* we shall discuss in *Parametric Polymorphism*.

Parametric types must always be fully applied. Using `intvec` by itself (without a type argument) is an error.

3.5 Records

Semantically, a record is a finite map from labels to values. These are supported by Futhark as a convenient syntactic extension on top of tuples. A label-value pairing is often called a *field*. As an example, let us return to our previous definition of complex numbers:

```
type complex = (f64, f64)
```

We can make the role of the two floats clear by using a record instead.

```
type complex = {re: f64, im: f64}
```

We can construct values of a record type with a *record expression*, which consists of field assignments enclosed in curly braces:

```
let sqrt_minus_one = {re = 0.0, im = -1.0}
```

The order of the fields in a record type or value does not matter, so the following definition is equivalent to the one above:

```
let sqrt_minus_one = {im = -1.0, re = 0.0}
```

In contrast to most other programming languages, record types in Futhark are *structural*, not *nominal*. This means that the name (if any) of a record type does not matter. For example, we can define a type abbreviation that is equivalent to the previous definition of `complex`:

```
type another_complex = {re: f64, im: f64}
```

The types `complex` and `another_complex` are entirely interchangeable. In fact, we do not need to name record types at all; they can be used anonymously:

```
let sqrt_minus_one: {re: f64, im: f64} = {re = 0.0, im = -1.0}
```

However, for readability purposes it is usually a good idea to use type abbreviations when working with records.

There are two ways to access the fields of records. The first is by *field projection*, which is done by dot notation known from most other programming languages. To access the `re` field of the `sqrt_minus_one` value defined above, we write `sqrt_minus_one.re`.

The second way of accessing field values is by pattern matching, just like we do with tuples. A record pattern is similar to a record expression, and consists of field patterns enclosed in curly braces. For example, a function for adding complex numbers could be defined as:

```
let complex_add ({re = x_re, im = x_im}: complex)
                ({re = y_re, im = y_im}: complex)
              : complex =
  {re = x_re + y_re, im = x_im + y_im}
```

As with tuple patterns, we can use record patterns in both function parameters, `let`-bindings, and `loop` parameters.

As a special syntactic convenience, we can elide the `= pat` part of a record pattern, which will bind the value of the field to a variable of the same name as the field. For example:

```
let conj ({re, im}: complex): complex =
  {re = re, im = -im}
```

This convenience is also present in tuple expressions. If we elide the definition of a field, the value will be taken from the variable in scope with the same name:

```
let conj ({re, im}: complex): complex =
  {re, im = -im}
```

3.5.1 Tuples as a Special Case of Records

In Futhark, tuples are merely records with numeric labels starting from 1. For example, the types `(i32,f64)` and `{1:i32,2:f64}` are indistinguishable. The main utility of this equivalence is that we can use field projection to access the components of tuples, rather than using a pattern in a `let`-binding. For example, we can say `foo.1` to extract the first component of a tuple.

Notice that the fields of a record must constitute a prefix of the positive numbers for it to be considered a tuple. The record type `{1:i32,3:f64}` does not correspond to a tuple, and neither does `{2:i32,3:f64}` (but `{2:f64,1:i32}` is equivalent to the tuple `(i32,f64)`, because field order does not matter).

3.6 Parametric Polymorphism

Consider the replication function we wrote earlier:

```
let replicate_i32 (n: i32) (x: i32): [n]i32 =
  map (\_ -> x) (0..<n)
```

This function works only for replicating values of type `i32`. If we wanted to replicate, say, a boolean value, we would have to write another function:

```
let replicate_bool (n: i32) (x: bool): [n]bool =
  map (\_ -> x) (0..<n)
```

This duplication is not particularly nice. Since the only difference between the two functions is the type of the `x` parameter, and we don't actually use any `i32`-specific operations in `replicate_i32`, or `bool`-specific operations in `replicate_bool`, we ought to be able to write a single function that is *parameterised* over the element type. In some languages, this is done with *generics*, or *template functions*. In ML-derived languages, including Futhark, we use *parametric polymorphism*. Just like the size parameters we saw earlier, a Futhark function may have *type parameters*. These are written as a name preceded by an apostrophe. As an example, this is a polymorphic version of `replicate`:

```
let replicate 't (n: i32) (x: t): [n]t =
  map (\_ -> x) (0..<n)
```

Notice how the type parameter binding is written as `'t`; we use just `t` to refer to the parametric type in the `x` parameter and the function return type. Type parameters may be freely intermixed with size parameters, but must precede all ordinary parameters. Just as with size parameters, we do not need to explicitly pass the types when we call a polymorphic function; they are automatically deduced from the concrete parameters.

We can also use type parameters when defining type abbreviations:

```
type triple 't = [3]t
```

And of course, these can be intermixed with size parameters:

```
type vector 't [n] = [n]t
```

In contrast to function definitions, the order of parameters in a type *does* matter. Hence, `vector i32 [3]` is correct, and `vector [3] i32` would produce an error.

We might try to use parametric types to further refine our previous definition of complex numbers, by making it polymorphic in the representation of scalar numbers:

```
type complex 't = {re: t, im: t}
```

This type abbreviation is fine, but we will find it difficult to write useful functions with it. Consider an attempt to define complex addition:

```
let complex_add 't ({re = x_re, im = x_im}: complex t)
                   ({re = y_re, im = y_im}: complex t)
                : complex t =
  {re = ?, im = ?}
```

How do we perform an addition `x_re` and `y_re`? These are both of type `t`, of which we know nothing. For all we know, they might be instantiated to something that is not numeric at all. Hence, the Futhark compiler will prevent us from using the + operator. In some languages, such as Haskell, facilities such as *type classes* may be used to support a notion of restricted polymorphism, where we can require that an instantiation of a type variable supports certain operations (like +). Futhark does not have type classes, but it does support programming with certain kinds of higher-order functions and it does have a powerful module system. The support for higher-order functions in Futhark and the module system are the subjects of the following sections.

3.7 Higher-Order Functions

Futhark supports certain kinds of higher-order functions. For performance reasons, certain restrictions apply, which means that Futhark can eliminate higher-order functions at compile time through a technique called *defunctionalisation [Hov18][HHE18]*. From a programmer's point-of-view, the main restrictions are the following:

1. Functions may not be stored inside arrays.
2. Functions may not be returned from branches in conditional expressions.
3. Functions are not allowed in loop parameters.

Whereas these restrictions seem daunting, functions may still be grouped in records and tuples and such structures may be passed to functions and even returned by functions. In effect, quite a few functional design patterns may be applied, ranging from defining polymorphic higher-order functions, for the purpose of obtaining a high degree of abstraction and code reuse (e.g., for defining program libraries), to specific uses of higher-order functions for representing various concepts as functions. Examples of such uses include a library for type-indexed compact serialisation (and deserialisation) of Futhark values *[Els05][Ken04]* and encoding of Conal Elliott's functional images *[Ell03]*.

We have seen earlier how anonymous functions may be constructed and passed as arguments to SOACs. Here is an example anonymous function that takes parameters `x`, `y`, and `z`, returns a value of type `t`, and has body *e*:

```
\x y z: t -> e
```

Futhark allows for the programmer to specify so-called *sections*, which provide a way to form implicit eta-expansions of partially applied operations. Sections are encapsulated in parentheses. Assuming `binop` is a binary operator, such as +, the section `(binop)` is equivalent to the

expression `\x y -> x binop y`. Similarly, the section `(x binop)` is equivalent to the expression `\y -> x binop y` and the section `(binop y)` is equivalent to the expression `\x -> x binop y`.

For making it easy to select fields from records (and tuples), a select-section may be used. An example is the section `(.a.b.c)`, which is equivalent to the expression `\y -> y.a.b.c`. Similarly, the example section `(.[i])`, for indexing into an array, is equivalent to the expression `\y -> y[i]`.

At a high level, Futhark functions are values, which can be used as any other values. However, to ensure that the Futhark compiler is able to compile the higher-order functions efficiently via defunctionalisation, certain type-driven restrictions exist on how functions can be used, as described earlier. Moreover, for Futhark to support higher-order polymorphic functions, type variables, when bound, are divided into non-lifted (bound with an apostrophe, e.g. `'t`), and lifted (bound with an apostrophe and a hat, e.g. `'^t`). Only lifted type parameters may be instantiated with a functional type. Within a function, a lifted type parameter is treated as a functional type. All abstract types declared in modules (see *Modules*) are considered non-lifted, and may not be functional.

Uniqueness typing generally interacts poorly with higher-order functions. The issue is that there is no way to express, in the type of a function, how many times a function argument is applied, or to what, which means that it will not be safe to pass a function that consumes its argument. The following two conservative rules govern the interaction between uniqueness types and higher-order functions:

1. In the expression `let p = e in ...`, if any in-place update takes place in the expression `e`, the value bound by `p` must not be or contain a function.
2. A function that consumes one of its arguments may not be passed as a higher-order argument to another function.

A number of higher-order utility functions are available at top-level. Amongst these are the following quite useful functions:

```
val const '^a '^b  : a -> b -> a          -- constant function
val id    '^a      : a -> a               -- identity function
val |>    '^a '^b  : a -> (a -> b) -> b   -- pipe right
val <|    '^a '^b  : (a -> b) -> a -> b   -- pipe left

val >->     '^a '^b '^c : (a -> b) (b -> c) -> a -> c
val <-<     '^a '^b '^c : (b -> c) (a -> b) -> a -> c

val curry   '^a '^b '^c : ((a,b) -> c) -> a -> b -> c
val uncurry '^a '^b '^c : (a -> b -> c) -> (a,b) -> c
```

3.8 Modules

When most programmers think of module systems, they think of rather utilitarian systems for namespace control and splitting programs across multiple files. And in most languages, the module system is indeed little more than this. But in Futhark, we have adopted an ML-style higher-order module system that permits *abstraction* over modules *[EHAO18]*. The module system is not just a method for organising Futhark programs, it is also a powerful facility for writing generic code. Most importantly, all module language constructs are eliminated from the program at compile time, using a technique called static interpretation *[Els99][Ann18]*. As a consequence, from a programmer's perspective, there is no overhead involved with making use of module language features.

3.8.1 Simple Modules

At the most basic level, a *module* (called a *structure* in Standard ML) is merely a collection of declarations

```
module add_i32 = {
  type t = i32
  let add (x: t) (y: t): t = x + y
  let zero: t = 0
}
```

Now, `add_i32.t` is an alias for the type `i32`, and `Addi32.add` is a function that adds two values of type `i32`. The only peculiar thing about this notation is the equal sign before the opening brace. The declaration above is actually a combination of a *module binding*

```
module add_i32 = ...
```

and a *module expression*

```
{
  type t = i32
  let add (x: t) (y: t): t = x + y
  let zero: t = 0
}
```

In this case, the module expression encapsulates a number of declarations enclosed in curly braces. In general, as the name suggests, a module expression is an expression that returns a module. A module expression is syntactically and conceptually distinct from a regular value expression, but serves much the same purpose. The module language is designed such that evaluation of a module expression can always be done at compile time.

Apart from a sequence of declarations, a module expression can also be merely the name of another module

```
module foo = add_i32
```

Now every name defined in `add_i32` is also available in `foo`. At compile-time, only a single version of the `add` function is defined.

3.8.2 Module Types

What we have seen so far is nothing more than a simple namespace mechanism. The ML module system only becomes truly powerful once we introduce module types and parametric modules (in Standard ML, these are called *signatures* and *functors*).

A module type is the counterpart to a value type. It describes which names are defined, and as what. We can define a module type that describes `add_i32`:

```
module type i32_adder = {
  type t = i32
  val add : t -> t -> t
  val zero : t
}
```

As with modules, we have the notion of a *module type expression*. In this case, the module type expression is a sequence of *specifications* enclosed in curly braces. A specification specifies how a name must be defined: as a value (including functions) of some type, as a type abbreviation, or as an abstract type (which we will return to later).

We can assert that some module implements a specific module type via a *module type ascription*:

```
module foo = add_i32 : i32_adder
```

Syntactic sugar lets us move the module type to the left of the equal sign:

```
module add_i32: i32_adder = {
  ...
}
```

When we are ascribing a module with a module type, the module type functions as a filter, removing anything not explicitly mentioned in the module type:

```
module bar = add_i32 : { type t = int
                         val zero : t }
```

An attempt to access `bar.add` will result in a compilation error, as the ascription has hidden it. This is known as an *opaque* ascription, because it obscures anything not explicitly mentioned in the module type. The module systems in Standard ML and OCaml support both opaque and *transparent* ascription, but in Futhark we support only opaque ascription. This example also demonstrates

the use of an anonymous module type. Module types are structural (just like value types), and are named only for convenience.

We can use type ascription with abstract types to hide the definition of a type from the users of a module:

```
module speeds: { type thing
                 val car : thing
                 val plane : thing
                 val futhark : thing
                 val speed : thing -> i32 } = {
  type thing = i32

  let car: thing = 0
  let plane: thing = 1
  let futhark: thing = 2

  let speed (x: thing): i32 =
    if      x == car     then 120
    else if x == plane   then 800
    else if x == futhark then 10001
    else                      0 -- will never happen
}
```

The (anonymous) module type asserts that a distinct type `thing` must exist, but does not mention its definition. There is no way for a user of the `speeds` module to do anything with a value of type `speeds.thing` apart from passing it to `speeds.speed`. The definition is entirely abstract. Furthermore, no values of type `speeds.thing` exists except those that are created by the `speeds` module.

3.8.3 Parametric Modules

While module types serve some purpose for namespace control and abstraction, their most interesting use is in the definition of parametric modules. A parametric module is conceptually equivalent to a function. Where a function takes a value as input and produces a value, a parametric module takes a module and produces a module. For example, given a module type

```
module type monoid = {
  type t
  val add : t -> t -> t
  val zero : t
}
```

We can define a parametric module that accepts a module satisfying the `monoid` module type, and produces a module containing a function for collapsing an array

```
module sum (M: monoid) = {
  let sum (a: []M.t): M.t =
    reduce M.add M.zero a
}
```

There is an implied assumption here, which is not captured by the type system: The function `add` must be associative and have `zero` as its neutral element. These constraints come from the parallel semantics of `reduce`, and the algebraic concept of a *monoid*. Notice that in `monoid`, no definition is given of the type `t`—we only assert that there must be some type `t`, and that certain operations are defined for it.

We can use the parametric module `sum` as follows:

```
module sum_i32 = sum add_i32
```

We can now refer to the function `sum_i32.sum`, which has type `[]i32 -> i32`. The type is only abstract inside the definition of the parametric module. We can instantiate `sum` again with another module, this time an anonymous module:

```
module prod_f64 = sum {
  type t = f64
  let add (x: f64) (y: f64): f64 = x * y
  let zero: f64 = 1.0
}
```

The function `prod_f64.sum` has type `[]f64 -> f64`, and computes the product of an array of numbers (we should probably have picked a more generic name than `sum` for this function).

Operationally, each application of a parametric module results in its definition being duplicated and references to the module parameter replace by references to the concrete module argument. This is quite similar to how C++ templates are implemented. Indeed, parametric modules can be seen as a simplified variant with no specialisation, and with module types to ensure rigid type checking. In C++, a template is type-checked when it is instantiated, whereas a parametric module is type-checked when it is defined.

Parametric modules, like other modules, can contain more than one declaration. This feature is useful for giving related functionality a common abstraction, for example to implement linear algebra operations that are polymorphic over the type of scalars. The following example uses an anonymous module type for the module parameter and the `open` declaration for bringing the names from a module into the current scope:

```
module linalg(M : {
  type scalar
  val zero : scalar
  val add : scalar -> scalar -> scalar
  val mul : scalar -> scalar -> scalar
```

(continues on next page)

(continued from previous page)

```
}) = {
  open M

  let dotprod [n] (xs: [n]scalar) (ys: [n]scalar)
    : scalar =
    reduce add zero (map2 mul xs ys)

  let matmul [n] [p] [m] (xss: [n][p]scalar)
                         (yss: [p][m]scalar)
    : [n][m]scalar =
    map (\xs -> map (dotprod xs) (transpose yss)) xss
}
```

3.8.4 Importing other files

While Futhark's module system is not directly file-oriented, there is still a close interaction. You can access code in other files as follows:

```
import "module"
```

The above will include all non-`local` top-level definitions from `module.fut` and make them available in the current Futhark program. The `.fut` extension is implied.

You can also include files from subdirectories::

```
import "path/to/a/file"
```

The above will include the file `path/to/a/file.fut` relative to the including file.

If we are defining a top-level function (or any other top-level construct) that we do not want to be visible outside the current file, we can prefix it with `local`:

```
local let i_am_hidden x = x + 2
```

Qualified imports are possible, where a module is created for the file::

```
module M = import "module"
```

In fact, a plain `import "module"` is equivalent to:

```
local open import "module"
```

This declaration opens `"module"` in the current file, but does not propagate its contents to modules that in turn `import` the current file.

Chapter 4

Practical Matters

The previous chapter introduced the Futhark language, the notion of parallel programming, and the most fundamental builtin functions. However, more knowledge is needed to write real high-quality Futhark programs. This chapter discusses various practicalities around Futhark programming: how to test and debug your code (*Testing and Debugging*), how to benchmark it once it works (*Benchmarking*), how to use the Futhark package manager to access library code (*Package Management*), and finally how to work around compiler limitations.

4.1 Testing and Debugging

This section discusses techniques for checking the correctness of Futhark programs via unit tests, as well as the debugging facilities provided by `futharki`.

The testing experience for Futhark is still rather raw. There are no advanced unit testing frameworks, no test generators or doc-testing, and certainly no property-based testing. Instead, we have `futhark-test`, which tests entry point functions against input/output example pairs. However, it is better than nothing, and quite simple to use. `futhark-test` will test the program with both a compiler (`futhark-c` by default, but this can be changed with `--compiler`) and `futharki`.

4.1.1 Testing with `futhark-test`

A Futhark program may contain a *test block*, which is a sequence of line comments in which one of the lines contains the divider `-- ==`. The lines preceding the divider are ignored, while the lines after are taken as a description of a test to perform. When `futhark-test` is passed one or more `.fut` files, it will look for test blocks and perform the tests they describe.

As an example, let us consider how to test a function for matrix multiplication. The function itself is defined as thus:

```
entry matmul [n][m][p] (x: [n][m]i32) (y: [m][p]i32): [n][p]i32 =
  map (\xr -> map (\yc -> reduce (+) 0 (map2 (*) xr yc))
                  (transpose y))
      x
```

Note that we use `entry` instead of `let` in order for the function to be callable from the outside.

We then add a test block:

```
-- Matrix multiplication.
-- ==
-- entry: matmul
```

The first line is a human-readable description, the second is the divider, and the third specifies the entry point that we wish to test. If the entry point is `main`, this part can be elided.

We now come to the input/output sets, which are written as follows:

```
-- input { [[1, 2]] [[3], [4]] }
-- output { [[11]] }
-- input { [[1, 2], [3, 4]] [[5, 6], [7, 8]] }
-- output { [[19, 22], [43, 50]] }
-- input { [[1, 2]] [[3]] }
```

The values are enclosed in curly braces, and multiple whitespace-separated values can be given. Only a limited subset of the Futhark value syntax is supported: Primitive values and multidimensional arrays of primitive values. In particular, no records or tuples are permitted. This subset is exactly that which is supported by compiled Futhark executables. If you have a need for testing functions that take more sophisticated input types, you will need to encode them using primitive types, and then construct them in the test function itself.

It is also possible to write *negative* tests, where we assert that the program must fail for a given input. In our case, when the shape of the matrices don't match up:

```
-- input { [[1, 2]] [[3]] }
-- error: matmul.fut:15
```

We provide a regular expression matching the expected error. In this case, we just assert that the correct line number is provided.

Type inference on the input/output values is not performed, so the types must be unambiguous. This means that the usual `[]` notation for an empty array will not work. Instead, a special `empty(t)` notation is used to represent an array of *row type* `t`. For example, we can test for empty arrays as such:

```
-- input { empty([]i32) empty([]i32) }
-- output { empty([]i32) }
```

Note also that since plain integer literals are assumed to be of type `i32`, and plain decimal literals to be of type `f64`, you will need to use type suffixes (*Basic Language Features*) to write values of other types.

As a convenience, `futhark-test` considers functions returning n-tuples to really be functions returning n values. This means we can put multiple values in an `output` stanza, just as we do with `input`.

Finally, it is also possible to specify test data stored in a separate file. This is useful when testing with very large datasets, in particular when they use the binary data format. This is done with the notation `@ file`:

```
-- compiled input @ big_matrices.in
-- output @ big_matrices.out
```

This also shows another feature of `futhark-test`: if we precede `input` with the word `compiled`, that test is not run with `futharki`. This is useful for large tests that would take too long to run interpreted. There are more ways to filter which tests and programs should be skipped for a given invocation of `futhark-test`; see the manual for more information.

Testing a Futhark Library

A Futhark library typically comprises a number of `.fut` files means to be `include`-ed by Futhark programs. Libraries typically do not define entry points of the form required by `futhark-test`. Indeed, it is not unusual for Futhark libraries to consist entirely of parametric modules and higher-order functions! These are not directly accessible to `futhark-test`.

The recommended solution is that, for every library file `foo.fut`, we define a corresponding `foo_tests.fut` that imports `foo.fut` and defines a number of entry points.

For example, suppose we have `sum.fut` that contains the `sum` module from *Parametric Modules*:

```
module type monoid = {
  type t
  val add : t -> t -> t
  val zero : t
}

module sum (M: monoid) = {
  let sum (a: []M.t): M.t =
    reduce M.add M.zero a
}
```

This cannot be tested directly with `futhark-test`, but we can define a `sum_tests.fut` that can:

```
import "sum"

-- ==
-- entry: test_sum_add_i32
-- input { [1, 2, 3, 4] }
-- output { 10 }

module sum_add_i32 = sum { type t = i32
                           let add = (i32.+)
                           let zero = 0i32
                         }

entry test_sum_add_i32 = sum_add_i32.sum

-- ==
-- entry: test_sum_prod_f32
-- input { [1f32, 2f32, 3f32, 4f32] }
-- output { 24f32 }

module sum_prod_f32 = sum { type t = f32
                            let add = (f32.*)
                            let zero = 1f32
                          }

entry test_sum_prod_f32 = sum_prod_f32.sum
```

You will have to use your own judgment when deciding which specific instantiations of a generic library you feel are worth testing.

4.1.2 Traces and Breakpoints

Testing is useful for determining the correctness of code, but does not in itself pinpoint the source of bugs. While you can go far simply by structuring your code as small functions that can be tested in isolation, it is sometimes necessary to inspect internal state and behaviour.

Compiled Futhark code does not possess much in the way of debugging facilities, but `futharki` has a couple of useful tools. Since `futharki` is very slow when compared to compiled code, this does mean that we can only debug on cut-down smaller testing sets, not on realistic workloads.

Specifically, we use the two functions `trace` and `break`. The `trace` function has the following type:

```
trace 't : t -> t
```

Semantically, `trace` just returns its argument unchanged, and when compiling your Futhark code, this is indeed all that will happen. However, `futharki` treats `trace` specially, and will print

the argument to the screen. This is useful for seeing the value of internal variables. For example, suppose we have the program `trace.fut`:

```
let main (xs: []i32) = map (\x -> trace x + 2) xs
```

We can then run it with `futharki` to get the following output:

```
$ echo [1,2,3] | futharki trace.fut
Trace at trace.fut:1:24-1:49: 1i32
Trace at trace.fut:1:24-1:49: 2i32
Trace at trace.fut:1:24-1:49: 3i32
[3i32, 4i32, 5i32]
```

Similarly, the `break` function is semantically also the identity function:

```
break 't : t -> t
```

When `futharki` encounters `break`, it suspends execution and lets us inspect the variables in scope. At the moment, this works *only* when running an expression within the `futharki` REPL, *not* when running directly from the command line. Suppose `break.fut` is:

```
let main (xs: []i32) = map (\x -> break x + 2) xs
```

Then we can load and run it from `futharki`:

```
[1]> main [1,2,3]
Breaking at [1]> :1:1-1:12 -> break.fut:1:24-1:49 -> /futlib/soacs.
↪fut:35:3-35:24 -> break.fut:1:35-1:41.
<Enter> to continue.
> x
1i32
>
Continuing...
Breaking at [1]> :1:1-1:12 -> break.fut:1:24-1:49 -> /futlib/soacs.
↪fut:35:3-35:24 -> break.fut:1:35-1:41.
<Enter> to continue.
>
Continuing...
Breaking at [1]> :1:1-1:12 -> break.fut:1:24-1:49 -> /futlib/soacs.
↪fut:35:3-35:24 -> break.fut:1:35-1:41.
<Enter> to continue.
>
Continuing...
[3i32, 4i32, 5i32]
>
```

Whenever we are stopped at a break point, we can enter arbitrary Futhark expressions to inspect the state of the environment. This is useful when operating on complex values.

4.2 Benchmarking

Consider an implementation of the dot product of two integer vectors:

```
let main (x: []i32) (y: []i32): i32 =
  reduce (+) 0 (map2 (*) x y)
```

We previously mentioned that, for small data sets, sequential execution is likely to be much faster than parallel execution. But how much faster? To answer this question, we need to measure the run time of the program on some data sets. This task is called *benchmarking*. There are many properties one can benchmark: memory usage, size of compiled executable, robustness to errors, and so forth. In this section, we are only concerned with run time. Specifically, we wish to measure *wall time*, which is how much time elapses in the real world from the time the computation starts, to the time it ends.

There is still some wiggle room in how we benchmark. For example, should we measure the time it takes to load the input data from disk? Or time it takes to initialise various devices and drivers? Should we perform a clean shutdown? How many times should we run the program, and should we report maximum, minimum, or average run time? We will not try to answer all of these questions, but instead merely describe the benchmarking tools provided by Futhark.

4.2.1 Simple Measurements

First, let us compile `dotprod.fut` to two different executables, one for each compiler:

```
$ futhark-c dotprod.fut -o dotprod-c
$ futhark-opencl dotprod.fut -o dotprod-opencl
```

One way to time execution is to use the standard `time(1)` tool:

```
$ echo [2,2,3] [4,5,6] | time ./dotprod-c
36i32
0.00user 0.00system 0:00.00elapsed ...
$ echo [2,2,3] [4,5,6] | time ./dotprod-opencl
36i32
0.20user 0.07system 0:00.29elapsed ...
```

It seems that `dotprod-c` executes in less than 10 milliseconds, while `dotprod-opencl` takes about 290 milliseconds. However, this comparison is not useful, as it also measures time taken to read the input (for both executables), as well as time taken to initialise the OpenCL driver (for `dotprod-opencl`). Recall that in a real application, the Futhark program would be compiled as a *library*, and the startup cost paid just once, while the program may be invoked multiple times. A more precise run-time measurement, where parsing, initialisation, and printing of results is not included, can be performed using the `-t` command line option, which specifies a file where the run-time (measured in microseconds) should be put:

```
$ echo [2,2,3] [4,5,6] | ./dotprod-c -t /dev/stderr > /dev/null
0
```

In this case, we ask for the runtime to be printed to the screen, and for the normal evaluation result to be thrown away. Apparently it takes less than one microsecond to compute the dot product of two three-element vectors on a CPU (this is not very surprising). On an AMD Vega 64 GPU:

```
$ echo [2,2,3] [4,5,6] | ./dotprod-opencl -t /dev/stderr > /dev/null
103
```

Over 100 microseconds! Most GPUs have fairly high launch invocation latencies, and so are not suited for small problems. We can use the `futhark-dataset(1)` tool to generate random test data of a desired size:

```
$ futhark-dataset -g [10000000]i32 -g [10000000]i32 > input
```

Two ten million element vectors should be enough work to amortise the GPU startup cost:

```
$ cat input | ./dotprod-opencl -t /dev/stderr > /dev/null
347
$ cat input | ./dotprod-c -t /dev/stderr > /dev/null
3801
```

That's more like it! Parallel execution is now more than ten times faster than sequential execution. This program is entirely memory-bound; on a compute-bound program we can expect much larger speedups.

You may have notice that these programs take *significantly* longer to run than indicated by these performance measurements. While GPU initialisation does take some time, most of the actual run-time in the example above is spent reading the data file from disk. By default, `futhark-dataset` produces output in a data format that is human-readable, but very slow for programs to process. We can use the `-b` option to make `futhark-dataset` generate data in an efficient binary format (which takes up less space on disk as well):

```
$ futhark-dataset -b -g [10000000]i32 -g [10000000]i32 > input
```

Reading binary data files is often orders of magnitude faster than reading textual input files. Compiled Futhark programs also support binary output via a `-b` option. The `futhark-dataset` tool can perform conversion between the binary and human-readable formats; see the manual page for more information.

4.2.2 Multiple Measurements

The technique presented in the previous section still has some problems. In particular, it is impractical if you want several measurements on the same dataset, which is in general preferable to even

out noise. While you can just repeat execution the desired number of times, this method has two problems:

1. The input file will be read multiple times, which can be slow for large data sets.
2. It prevents the device from "warming up", as every run re-initialises the GPU and re-uploads code.

The second point is more important than it may seem. Certain OpenCL operations (such as memory allocation) are relatively costly, and Futhark uses various caches and buffers to minimise the number of expensive OpenCL operations. However, these caches will all be cold the first time the program runs. Hence we wish to perform more than one run per program instance, so that we can take advantage of the warm caches. This method is also a more plausible proxy for real-world usage of Futhark, as Futhark is typically compiled to a library, where the same functions are called repeatedly by some client code.

Compiled Futhark executables support an `-r N` option that asks the program to perform `N` runs internally, and report runtime for each. Additionally, a non-measured warm-up run is performed initially. We can use it like this:

```
$ cat input | ./dotprod-opencl -t /dev/stderr -r 10 > /dev/null
285
330
281
284
285
278
285
330
284
282
```

Our runtimes are now much better. And importantly, there are more of them, so we can perform analyses like, such as determining the variance, to figure out how predictable the performance is.

However, we can do better still. Futhark comes with a tool for performing automated benchmark runs of programs, called `futhark-bench`. This tool relies on a specially formatted header comment that contains input/output pairs, just like `futhark-test` (see *Testing and Debugging*). The Futhark User's Guide contains a full description, but here is a simple example. First, we introduce a new program, `sumsquares.fut`, with smaller data sets for convenience:

```
-- Given N, compute the sum of squares of the first N integers.
-- ==
-- compiled input {       1000 } output {   332833500 }
-- compiled input {    1000000 } output {   584144992 }
-- compiled input { 1000000000 } output { -2087553280 }

let main (n: i32): i32 =
```

(continues on next page)

(continued from previous page)

```
  reduce (+) 0 (map (**2) (iota n))
```

The line containing == is used to separate the human-readable benchmark description from input-output pairs. It is also possible to keep the data set is located in an external file (see the manual page for more information.).

We can use `futhark-bench` to measure the performance of `sumsquares.fut` as follows:

```
$ futhark-bench sumsquares.fut
Compiling src/sumsquares.fut...
Results for src/sumsquares.fut:
dataset #0 ("1000i32"):              0.20us (avg. of 10 runs; RSD: 2.00)
dataset #1 ("1000000i32"):         290.00us (avg. of 10 runs; RSD: 0.03)
dataset #2 ("1000000000i32"):  270154.20us (avg. of 10 runs; RSD: 0.01)
```

These are measurements using the default compiler, which is `futhark-c`. If we want to see how our program performs when compiled with `futhark-opencl`, we can invoke `futhark-bench`:

```
$ futhark-bench --compiler=futhark-opencl sumsquares.fut
Compiling src/sumsquares.fut...
Results for src/sumsquares.fut:
dataset #0 ("1000i32"):             49.70us (avg. of 10 runs; RSD: 0.18)
dataset #1 ("1000000i32"):          44.40us (avg. of 10 runs; RSD: 0.02)
dataset #2 ("1000000000i32"):     1693.80us (avg. of 10 runs; RSD: 0.04)
```

We can now compare the performance of CPU execution with GPU execution. The tool takes care of the mechanics of run-time measurements, and even computes the relative standard deviation ("RSD") of the measurements for us. The correctness of the output is also automatically checked. By default, `futhark-bench` performs ten runs for every data set, but this number can be changed with the `--runs` command line option. Unless you can articulate a good reason not to, always use `futhark-bench` for benchmarking.

4.3 Package Management

A Futhark package is a downloadable collection of `.fut` files and little more. There is a (not necessarily comprehensive) list of known packages. The following discusses only how to *use* packages. For authoring your own, please see the corresponding section in the User's Guide.

4.3.1 Basic Concepts

A package is uniquely identified with a *package path*, which is similar to a URL, except without a protocol. At the moment, package paths are always links to Git repositories hosted on GitHub or GitLab. As an example, a package path may be `github.com/athas/fut-foo`.

Packages are versioned with semantic version numbers of the form `X.Y.Z`. Whenever versions are indicated, all three digits must always be given (that is, `1.0` is not a valid shorthand for `1.0.0`).

Most `futhark-pkg` operations involve reading and writing a *package manifest*, which is always stored in a file called `futhark.pkg`. The `futhark.pkg` file is human-editable, but is in day-to-day use mainly modified by `futhark-pkg` automatically. You will normally have one `futhark.pkg` file for each of your Futhark projects. Packages are installed in a location relative to the location of `futhark.pkg`.

4.3.2 Installing Packages

Required packages can be added by using `futhark-pkg add`, for example:

```
$ futhark-pkg add github.com/athas/fut-foo 0.1.0
```

This will create a new file `futhark.pkg` with the following contents:

```
require {
  github.com/athas/fut-foo 0.1.0
↪#d285563c25c5152b1ae80fc64de64ff2775fa733
}
```

This lists one required package, with its package path, minimum version, and the expected commit hash. The latter is used for verification, to ensure that the contents of a package version cannot silently change.

`futhark-pkg` will perform network requests to determine whether a package of the given name and with the given version exists and fail otherwise (but it will not check whether the package is otherwise well-formed). The version number can be elided, in which case `futhark-pkg` will use the newest available version. If the package is already present in `futhark.pkg`, it will simply have its version requirement changed to the one specified in the command. Any dependencies of the package will *not* be added to `futhark.pkg`, but will still be downloaded by `futhark-pkg sync` (see below).

Adding a package with `futhark-pkg add` modifies `futhark.pkg`, but does not download the package files. This is done with `futhark-pkg sync` (without further options). The contents of each required dependency and any transitive dependencies will be stored in a subdirectory of `lib/` corresponding to their package path. Following the earlier example:

```
$ futhark-pkg sync
$ tree lib
lib
└── github.com
    └── athas
        └── fut-foo
            └── foo.fut

3 directories, 1 file
```

Warning: `futhark-sync` will remove any unrecognized files or local modifications to files in `lib/`. Unless you are creating your own package, you should not add anything to the `lib/` directory - it is fully controlled by `futhark-pkg`.

Packages can be removed from `futhark.pkg` with:

```
$ futhark-pkg remove pkgpath
```

You will need to run `futhark-sync` to actually remove the files in `lib/`.

The intended usage is that `futhark.pkg` is added to version control, but `lib/` is not, as the contents of `lib/` can always be reproduced from `futhark.pkg`. However, adding `lib/` works just fine as well.

4.3.3 Importing Files from Dependencies

`futhark-pkg sync` will populate the `lib/` directory, but does not interact with the compiler in any way. The downloaded files can be imported using the `import` mechanism (see *Importing other files*). For example, assuming the package contains a file `foo.fut`, the following top-level declaration brings all names declared in the file into scope:

```
import "lib/github.com/athas/fut-foo/foo"
```

Ultimately, everything boils down to ordinary file system semantics. This has the downside of relatively long and clumsy import paths, but the upside of predictability.

4.3.4 Upgrading Dependencies

The `futhark-pkg upgrade` command will update every version requirement in `futhark.pkg` to be the most recent available version. You still need to run `futhark-pkg sync` to actually retrieve the new versions. Be careful - while upgrades are safe if semantic versioning is followed correctly, this is not yet properly machine-checked, so human mistakes may occur.

As an example:

```
$ cat futhark.pkg
require {
  github.com/athas/fut-foo 0.1.0
 ↪#d285563c25c5152b1ae80fc64de64ff2775fa733
}
$ futhark-pkg upgrade
Upgraded github.com/athas/fut-foo 0.1.0 => 0.2.1.
$ cat futhark.pkg
require {
  github.com/athas/fut-foo 0.2.1
 ↪#3ddc9fc93c1d8ce560a3961e55547e5c78bd0f3e
}
$ futhark-pkg sync
$ tree lib
lib
└── github.com
    └── athas
        ├── fut-bar
        │   └── bar.fut
        └── fut-foo
            └── foo.fut

4 directories, 2 files
```

Note that `fut-foo 0.2.1` depends on `github.com/athas/fut-bar`, so it was fetched by `futhark-pkg sync`.

`futhark-pkg upgrade` will *never* upgrade across a major version number. Due to the principle of Semantic Import Versioning, a new major version is a completely different package from the point of view of the package manager. Thus, to upgrade to a new major version, you will need to use `futhark-pkg add` to add the new version and `futhark-pkg remove` to remove the old version. Or you can keep it around - it is perfectly acceptable to depend on multiple major versions of the same package, because they are really different packages.

4.4 When Things Go Wrong

Futhark is a young language and an *on-going research project*, and you should not expect the same predictability and quality of error messages that you may be used to from more mature languages. Further, not all Futhark compilers are guaranteed to be able to compile all Futhark programs. In general, the limitations you will encounter will tend to fall in two categories:

Essential limitations touch upon fundamental restrictions in the target platform(s) for the Futhark compiler. For example, GPUs do not permit dynamic memory allocation inside GPU code. All memory must be pre-allocated before GPU programs are launched. This means that the

Futhark compiler must be able to pre-compute the size of all intermediate arrays (symbolically), or compilation will fail.

Implementation limitations are weaknesses in the Futhark compiler that could reasonably be solved. Many implementation limitations, such as the inability to pre-compute some array sizes, or eliminate bounds checks inside parallel sections, will manifest themselves as essential limitations that could be worked around by a smarter compiler.

For example, consider this program:

```
let main (n: i32): [][]i32 =
  map (\i ->
         let a = (0..<i)
         let b = (0..<n-i)
         in concat a b)
      (0..<n)
```

At the time of this writing, the `futhark-opencl` compiler will fail with the not particularly illuminating error message `Cannot allocate memory in kernel`. The reason is that the compiler is trying to compile the `map` to parallel code, which involves pre-allocating memory for the `a` and `b` array. It is unable to do this, as the sizes of these two arrays depend on values that are only known *inside* the map, which is too late. There are various techniques the Futhark compiler could use to estimate how much memory would be needed, but these have not yet been implemented.

It is usually possible, sometimes with some pain, to come up with a workaround. We could rewrite the program as:

```
let main(n: i32): [][]i32 =
  let scratch = (0..<n)
  in map (\i ->
            let res = (0..<n)
            let res[i:n] = scratch[0:n-i]
            in res)
         (0..<n)
```

This exploits the fact that the compiler does not generate allocations for array slices or in-place updates. The only allocation is of the initial `res`, the size of which can be computed before entering the `map`.

Chapter 5

Interoperability

Futhark is a purely functional high-performance language incapable of interacting with the outside world except through function parameters. This makes it impossible to write full applications in Futhark, except via the limited standard input-based interface that we used in the preceding chapters. In practice, this interface is too slow and too inflexible to be useful. Instead, the Futhark compiler is designed to generate *libraries*, which can then be invoked by general-purpose languages. In this chapter we will see how to call Futhark from Python and C, with particular attention paid to the former.

5.1 Calling Futhark from Python

Python is a language with many qualities, but few would claim that performance is among them. While libraries such as NumPy can be used, they are not as flexible as being able to write code directly in a high-performance language. Unfortunately, writing the performance-critical parts of a Python program in (say) C is not always a good experience, and the interfacing between the Python code and the C code can be awkward and inelegant (although to be fair, it is still nicer in Python than in many other languages). It would be more convenient if we could compile a high-performance language directly to a Python module that we could then `import` like any other piece of Python code. Of course, this entire exercise is only worthwhile if the code in the resulting Python module executes much faster than manually written Python. Fortunately, when most of the computation can be offloaded to the GPU via OpenCL, the Futhark compiler is capable of this feat.

OpenCL works by having an ordinary program running on the CPU that transmits code and data to the GPU (or any other *accelerator*, but we'll stick to GPUs). In the ideal case, the CPU-code is mostly glue that performs bookkeeping and making API calls - in other words, not resource-intensive, and exactly what Python is good at. No matter the language the CPU code is written in, the GPU code will be written in OpenCL C and translated at program initialisation to whatever machine code is needed by the concrete GPU.

This is what is exploited by the PyOpenCL backend in the Futhark compiler. Certainly, the CPU-

level code is written in pure Python and quite slow, but all it does is use the PyOpenCL library to offload work to the GPU. The fact that this offloading takes place is hidden from the user of the generated code, who is provided a module with functions that accept and produce ordinary NumPy arrays.

Consider our usual dot product program:

```
let main (x: []i32) (y: []i32): i32 =
  reduce (+) 0 (map2 (*) x y)
```

We can compile this to a Python module:

```
$ futhark-pyopencl --library dotprod.fut
```

The result is a file `dotprod.py` that we can import from within Python:

```
$ python
>>> import dotprod
```

The `dotprod.py` module defines a class `dotprod` that we must instantiate. The class maintains various bits of bookkeeping information, and exposes a method for every entry point in our program (here just `main`):

```
>>> o = dotprod.dotprod()
```

We will get an error if we try to pass Python lists to the entry point, as lists are not arrays:

```
>>> o.main([1,2,3], [4,5,6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dotprod.py", line 2416, in main
    x_mem_3884_ext))
TypeError: Argument #0 has invalid value
Futhark type: []i32
Argument has Python type <type 'list'> and value: [1, 2, 3]
```

Instead, we have to construct a properly typed NumPy array:

```
>>> import numpy as np
>>> o.main(np.array([1,2,3], dtype=np.int32),
           np.array([4,5,6], dtype=np.int32))
32
```

The integer that is returned is a normal Python object of an appropriate type (in this case it will have type `np.int32`). If an array is returned, it is in the form of a PyOpenCL array, which is mostly compatible with NumPy arrays, except that the backing memory still resides on the GPU, and is not copied over to the CPU unless necessary. This makes it efficient to take the output of

one entry point and pass it as the input to another. PyOpenCL arrays contain a `.get()` method that can be used to construct an equivalent NumPy array, if desired.

5.2 Calling Futhark from C

Let us once again consider `dotprod.fut`:

```
let main (x: []i32) (y: []i32): i32 =
  reduce (+) 0 (map2 (*) x y)
```

We can compile it with the `futhark-opencl` compiler:

```
$ futhark-opencl --library dotprod.fut
```

This produces two files in the current directory: `dotprod.c` and `dotprod.h`. We can compile `dotprod.c` to a shared library like this:

```
$ gcc dotprod.c -o libdotprod.so -fPIC -shared
```

We can now link to `libdotprod.so` the same way we link with any other shared library. But before we get that far, let's take a look at (parts of) the generated `dotprod.h` file. We have written the code generator to produce as simple header files as possible, with no superfluous crud, in order to make them human-readable. This is particularly useful at the moment, since few explanatory comments are inserted in the header file.

The first declarations are related to initialisation, which is based on first constructing a *configuration* object, which can then be used to obtain a *context*. The context is used in all subsequent calls, and contains GPU state and the like. We elide most of the functions for setting configuration properties, as they are not very interesting:

```
/*
 * Initialisation
*/

struct futhark_context_config ;

struct futhark_context_config *futhark_context_config_new();

void futhark_context_config_free(struct futhark_context_config *cfg);

void futhark_context_config_set_device(struct futhark_context_config␣
↪*cfg,
                                       const char *s);

...
```

(continues on next page)

(continued from previous page)

```
struct futhark_context ;

struct futhark_context *futhark_context_new(struct futhark_context_
↪config *cfg);

void futhark_context_free(struct futhark_context *ctx);

int futhark_context_sync(struct futhark_context *ctx);
```

The above demonstrates a pervasive design decision in the API: the use of pointers to *opaque structs*. The struct `futhark_context` is not given a definition, and the only way to construct it is via the function `futhark_context_new()`. This means that we cannot allocate it statically, which is contrary to how one would normally design a C library. The motivation behind this design is twofold:

1. It keeps the header file readable, as it elides implementation details like struct members.
2. It is easier to use from FFIs. Most FFIs make it very easy to work with functions that only accept and produce pointers (and primitive types), but accessing and allocating structs is a little more involved.

The disadvantage is a little more boilerplate, and a little more dynamic allocation. However, relatively few objects of this kind are used, so the performance impact should be nil.

The next part of the header file concerns itself with arrays - how they are created and accessed:

```
/*
 * Arrays
*/

struct futhark_i32_1d ;

struct futhark_i32_1d *futhark_new_i32_1d(struct futhark_context *ctx,
                                          int32_t *data,
                                          int dim0);

int futhark_free_i32_1d(struct futhark_context *ctx,
                        struct futhark_i32_1d *arr);

int futhark_values_i32_1d(struct futhark_context *ctx,
                          struct futhark_i32_1d *arr,
                          int32_t *data);

int64_t *futhark_shape_i32_1d(struct futhark_context *ctx,
                              struct futhark_i32_1d *arr);
```

Again we see the use of pointers to opaque structs. We can use `futhark_new_i32_1d` to construct a Futhark array from a C array, and we can use `futhark_values_i32_1d` to read all elements from a Futhark array. The representation used by the Futhark array is intentionally hidden from us - we do not even know (or care) whether it is resident in CPU or GPU memory. The code generator automatically generates a struct and accessor functions for every distinct array type used in the entry points of the Futhark program.

The single entry point is declared like this:

```
int futhark_entry_dotprod(struct futhark_context *ctx,
                          int32_t *out0,
                          const struct futhark_i32_1d *in0,
                          const struct futhark_i32_1d *in1);
```

As the original Futhark program accepted two parameters and returned one value, the corresponding C function takes one *out* parameter and two *in* parameters (as well as a context parameter).

We have now seen enough to write a small C program (with no error handling) that calls our generated library:

```
#include <stdio.h>

#include "dotprod.h"

int main() {
  int x[] = { 1, 2, 3, 4 };
  int y[] = { 2, 3, 4, 1 };

  struct futhark_context_config *cfg = futhark_context_config_new();
  struct futhark_context *ctx = futhark_context_new(cfg);

  struct futhark_i32_1d *x_arr = futhark_new_i32_1d(ctx, x, 4);
  struct futhark_i32_1d *y_arr = futhark_new_i32_1d(ctx, y, 4);

  int res;
  futhark_entry_dotprod(ctx, &res, x_arr, y_arr);
  futhark_context_sync(ctx);

  printf("Result: %d\n", res);

  futhark_free_i32_1d(ctx, x_arr);
  futhark_free_i32_1d(ctx, y_arr);

  futhark_context_free(ctx);
  futhark_context_config_free(cfg);
}
```

We hard-code the input data here, but we could just as well have read it from somewhere. The

call to `futhark_context_new()` is where the GPU is initialised (is applicable) and OpenCL kernel code is compiled and uploaded to the device. This call might be relatively slow. However, subsequent calls to entry point functions (`futhark_dotprod()`) will be efficient, as they re-use the already initialised context.

Note the use of `futhark_context_sync()` after calling the entry point: Futhark does not guarantee that the final results have been written until we synchronise explicitly. Note also that we free the two arrays `x_arr` and `y_arr` once we are done with them - memory management is entirely manual.

If we save this program as `luser.c`, we can compile and run it like this:

```
$ gcc luser.c -o luser -lOpenCL -lm -ldotprod
$ ./luser
Result: 24
```

You may need to set `LD_LIBRARY_PATH=.` before the dynamic linker can find `libdotprod.so`. Also, this program will only work if the default OpenCL device is usable on your system, since we did not request any specific device. For testing on a system that does not support OpenCL, simply use `futhark-c` instead of `futhark-opencl`. The generated API will be the same.

5.3 Handling Awkward Futhark Types

Our dot product function uses only types that map easily to NumPy and C: primitives and arrays of primitives. But what happens if we have an entry point that involves abstract types with hidden definitions, or types with no clear analogue in C, such as records or arrays of tuples? In this case, the generated API defines structs for *opaque types* that support very few operations.

Consider the following contrived program, `pack.fut`, which contains two entry points:

```
entry pack (xs: []i32) (ys: []i32): [](i32,i32) = zip xs ys

entry unpack (zs: [](i32,i32)): ([]i32,[]i32) = unzip zs
```

The `pack` function turns two arrays into one array of pairs, and the `unpack` function reverses the operation. If compiled to Python, the `pack` function will return a special "opaque" object whose contents cannot be inspected. If compiled to C, `pack.h` contains the following definitions:

```
struct futhark_opaque_z31U814583239044437263 ;

int futhark_free_opaque_z31U814583239044437263(struct futhark_context␣
↪*ctx,
                                               struct futhark_opaque_
↪z31U814583239044437263 *obj);
```

(continues on next page)

(continued from previous page)

```
int futhark_pack(struct futhark_context *ctx,
                 struct futhark_opaque_z31U814583239044437263 **out0,
                 struct futhark_i32_1d *in0,
                 struct futhark_i32_1d *in1);

int futhark_unpack(struct futhark_context *ctx,
                   struct futhark_i32_1d **out0,
                   struct futhark_i32_1d **out1,
                   struct futhark_opaque_z31U814583239044437263 *in0);
```

The unfortunately named struct, `futhark_opaque_z31U814583239044437263`, represents an array of tuples. There is nothing we can do with it except for freeing it, or passing it back to an entry point. In fact, the name is not even stable - it's a hash of the internal representation. If you try the above example, you may see a different name.

Opaque types typically occur when you are writing a Futhark program that keeps some kind of state that you don't want the user modifying or reading directly, but you need access to for each call to an entry point. Since Futhark programs are purely functional (and therefore stateless), having the user to manually pass back the state returned by the previous call is the only way to accomplish this. Fortunately, we can assign these opaque types somewhat more readable names by type abbreviations:

```
type array_of_pairs = [](i32,i32)

entry pack (xs: []i32) (ys: []i32): array_of_pairs = zip xs ys

entry unpack (zs: array_of_pairs): ([]i32,[]i32) = unzip zs
```

Now, when compiled to C, we obtain a somewhat more readable name for the opaque type:

```
struct futhark_opaque_array_of_pairs ;

int futhark_free_opaque_array_of_pairs(struct futhark_context *ctx,
                                       struct futhark_opaque_array_of_
↪pairs *obj);

int futhark_entry_pack(struct futhark_context *ctx,
                       struct futhark_opaque_array_of_pairs **out0,␣
↪const
                       struct futhark_i32_1d *in0, const
                       struct futhark_i32_1d *in1);

int futhark_entry_unpack(struct futhark_context *ctx,
                         struct futhark_i32_1d **out0,
                         struct futhark_i32_1d **out1, const
                         struct futhark_opaque_array_of_pairs *in0);
```

We have to be careful to use the type abbreviation everywhere, as the compiler will generate the hash-named opaque in any place that we miss.

Chapter 6

A Parallel Cost Model for Futhark Programs

In this chapter we develop a more formal model for Futhark and provide an ideal cost model for the language in terms of the concepts of work and span. Before we present the cost model for the language, we present a simple type system for Futhark and an evaluation semantics. In the initial development, we shall not consider Futhark's more advanced features such as loops and uniqueness types, but we shall return to these constructs later in the chapter.

Futhark supports certain kinds of nested parallelism. For instance, Futhark can in many cases map two nested maps into fully parallel code. Consider the following Futhark function:

```
let multable (n : i32) : [n][n]i32 =
  map (\i ->
    map (\j -> i * j) (iota n))
      (iota n)
```

In the case of this program, Futhark will flatten the code to make a single flat kernel. We shall return to the concept of flattening in a later chapter.

When we shall understand how efficient an algorithm is, we shall build our analysis around the two concepts of work and span. These concepts are defined inductively over the various Futhark language constructs and we may therefore argue about work and span in a compositional way. For instance, if we want to know about the work required to execute the `multable` function, we need to know about how to compute the work for a call to the `map` SOAC, how to compute the work for the `iota` operation, how to compute the work for the multiply operation, and, finally, how to combine the work. The way to determine the work for a `map` SOAC instance is to multiply the size of the argument array with the work of the body of the argument function. Thus, we have

$$W(\texttt{map}\ (\texttt{\textbackslash j} \rightarrow \texttt{i * j})\ (\texttt{iota}\ n)) = n + 1$$

Applying a similar argument to the outer map, we get

$$W(\texttt{map}\ (\texttt{\textbackslash i} \rightarrow \cdots)\ (\texttt{iota}\ n)) = (n+1)^2$$

Most often we are interested in finding the asymptotical complexity of the algorithm we are analyzing, in which case we will simply write

$$W(\texttt{map}\ (\texttt{\textbackslash i} \rightarrow \cdots)(\texttt{iota}\ n) = O(n^2)$$

In a similar way we can derive that the span of a call `multable n`, written $S(\texttt{multable n})$, is $O(1)$.

6.1 Futhark - the Language

In this section we present a simplified version of the Futhark language in terms of syntax, a type system for the language, and a strict evaluation semantics.

We assume a countable infinite number of program variables, ranged over by x and f. Binary infix scalar operators, first-order built-in operations, and second order array combinators are given as follows:

$$\begin{aligned}
\textit{binop} ::= {} & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \cdots \\
\\
\textit{op} ::= {} & \texttt{-} \mid \texttt{abs} \mid \texttt{copy} \mid \texttt{concat} \mid \texttt{empty} \\
& \mid \texttt{iota} \mid \texttt{partition} \mid \texttt{rearrange} \\
& \mid \texttt{replicate} \mid \texttt{reshape} \\
& \mid \texttt{rotate} \mid \texttt{shape} \mid \texttt{scatter} \\
& \mid \texttt{split} \mid \texttt{transpose} \mid \texttt{unzip} \mid \texttt{zip} \\
\\
\textit{soac} ::= {} & \texttt{map} \mid \texttt{reduce} \mid \texttt{reduce_comm} \\
& \mid \texttt{scan} \mid \texttt{filter} \mid \texttt{partition}
\end{aligned}$$

In the grammar for the Futhark language below, we have eluded both the required explicit type annotations and the optional explicit type annotations. Also for simplicity, we are considering only

"unnested" pattern matching and we do not, in this section, consider uniqueness types.

$$p ::= \; x \mid (x_1, ..., x_n)$$

$$ps ::= \; p_1 \cdots p_n$$

$$F ::= \; \backslash ps \to e \mid e \; binop \mid binop \; e$$

$$P ::= \; \texttt{let} \; f \; ps = e \mid P_1 P_2 \mid \texttt{let} \; p = e$$

$$\begin{aligned} v ::= \; & \texttt{true} \mid \texttt{false} \mid n \mid r \\ & \mid \texttt{[}v_1, ..., v_n\texttt{]} \mid \texttt{(}v_1, ..., v_n\texttt{)} \end{aligned}$$

$$\begin{aligned} e ::= \; & x \mid v \mid \texttt{let} \; ps = e \; \texttt{in} \; e' \\ & \mid e\texttt{[}e'\texttt{]} \mid e\texttt{[}e'\texttt{:}e''\texttt{]} \\ & \mid \texttt{[}e_1, ..., e_n\texttt{]} \mid \texttt{(}v_1, ..., v_n\texttt{)} \\ & \mid f e_1 ... e_n \mid op \; e_1 ... e_n \mid e_1 \; binop \; e_2 \\ & \mid \texttt{loop} \; p_1 = e_1, \cdots, p_n = e_n \; \texttt{for} \; x < e \; \texttt{do} \; e' \\ & \mid \texttt{loop} \; p_1 = e_1, \cdots, p_n = e_n \; \texttt{while} \; e \; \texttt{do} \; e' \\ & \mid soac \; F \; e_1 \; \cdots \; e_n \end{aligned}$$

6.2 Futhark Type System

Without considering Futhark's uniqueness type system, Futhark's type system is simple. Types (τ) follow the following grammar-slightly simplified:

$$\begin{aligned} \tau ::= \; & \texttt{i32} \mid \texttt{f32} \mid \texttt{bool} \mid \texttt{[]}\tau \\ & \mid (\tau_1, \cdots, \tau_n) \mid \tau \to \tau' \mid \alpha \end{aligned}$$

We shall refer to the types `i32`, `f32`, and `bool` as *basic types*. Futhark supports more basic types than those presented here; consult *Basic Language Features* for a complete list.

In practice, Futhark requires a programmer to provide explicit parameter types and an explicit result type for top-level function declarations. Similarly, in practice, Futhark requires explicit types for top-level `let` bindings. In such explicit types, type variables are not allowed; at present, Futhark does not allow for a programmer to declare polymorphic functions.

Futhark's second order array combinators and some of its primitive operations do have *polymorphic* types, which we specify by introducing the concept of *type schemes*, ranged over by σ, which are basically quantified types with α and β ranging over ordinary types. When $\sigma = \forall\vec{\alpha}.\tau$ is some type scheme, we say that τ' is an instance of σ, written $\sigma \geq \tau'$ if there exists a substitution $[\vec{\tau}/\vec{\alpha}]$

such that $\tau[\vec{\tau}/\vec{\alpha}] = \tau'$. We require all substitutions to be *simple* in the sense that substitutions do not allow for function types, product types, or type variables to be substituted. Other restrictions may apply, which will be specified using a *type variable constraint* $\alpha \triangleright T$, where T is a set of basic types.

The type schemes for Futhark's second-order array combinators are as follows:

$$
\begin{array}{rcl}
\textit{soac} & : & \mathrm{TypeOf}(\textit{soac}) \\
\hline
\texttt{filter} & : & \forall\alpha.(\alpha \to \texttt{bool}) \to []\alpha \to []\alpha \\
\texttt{map} & : & \forall\alpha_1 \cdots \alpha_n\beta.(\alpha_1 \to \cdots \to \alpha_n \to \beta) \\
 & & \to []\alpha_1 \to \cdots \to []\alpha_n \to []\beta \\
\texttt{reduce} & : & \forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha \\
\texttt{scan} & : & \forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to []\alpha
\end{array}
$$

The type schemes for Futhark's built-in first-order operations are as follows:

$$
\begin{array}{rcl}
\textit{op} & : & \mathrm{TypeOf}(\textit{op}) \\
\hline
\texttt{concat} & : & \forall\alpha.[]\alpha \to \cdots \to []\alpha \to []\alpha \\
\texttt{empty} & : & \forall\alpha.[]\alpha \\
\texttt{iota} & : & \texttt{int} \to []\texttt{int} \\
\texttt{replicate} & : & \forall\alpha.\texttt{int} \to \alpha \to []\alpha \\
\texttt{rotate} & : & \forall\alpha.\texttt{int} \to []\alpha \to []\alpha \\
\texttt{transpose} & : & \forall\alpha.[][]\alpha \to [][]\alpha \\
\texttt{unzip} & : & \forall\alpha_1 \cdots \alpha_n.[](\alpha_1, \cdots, \alpha_n) \\
 & & \to ([]\alpha_1, \cdots, []\alpha_n) \\
\texttt{scatter} & : & \forall\alpha.[]\alpha \to []\texttt{int} \to []\alpha \to []\alpha \\
\texttt{zip} & : & \forall\alpha_1 \cdots \alpha_n.[]\alpha_1 \to \cdots \to []\alpha_n \\
 & & \to [](\alpha_1, \cdots, \alpha_n)
\end{array}
$$

The type schemes for Futhark's built-in infix scalar operations are as follows:

$$
\begin{array}{rcl}
\textit{binop} & : & \mathrm{TypeOf}(\textit{binop}) \\
\hline
\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \cdots & : & \forall\alpha \triangleright \{\texttt{i32}, \texttt{f32}\}.\alpha \to \alpha \to \alpha \\
\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=} & : & \forall\alpha \triangleright \{\texttt{i32}, \texttt{f32}\}.\alpha \to \alpha \to \texttt{bool}
\end{array}
$$

We use Γ to range over *type environments*, which are finite maps mapping variables to types. We use $\{\}$ to denote the empty type environment and $\{x : \tau\}$ to denote a singleton type environment. When Γ is some type environment, we write $\Gamma, x : \tau$ to denote the type environment with domain $\mathrm{Dom}(\Gamma) \cup \{x\}$ and values $(\Gamma, x : \tau)(y) = \tau$ if $y = x$ and $\Gamma(y)$, otherwise. Moreover, when Γ and Γ' are type environments, we write $\Gamma + \Gamma'$ to denote the type environment with domain $\mathrm{Dom}(\Gamma) \cup \mathrm{Dom}(\Gamma')$ and values $(\Gamma + \Gamma')(x) = \Gamma'(x)$ if $x \in \mathrm{Dom}(\Gamma')$ and $\Gamma(x)$, otherwise.

Type judgments for values take the form $\vdash v : \tau$, which are read "the value v has type τ." Type judgments for expressions take the form $\Gamma \vdash e : \tau$, which are read "in the type environment Γ, the expression e has type τ." Finally, type judgments for programs take the form $\Gamma \vdash P : \Gamma'$, which are read "in the type environment Γ, the program P has type environment Γ'."

Values $\boxed{\vdash v : \tau}$

$$\frac{}{\vdash r : \texttt{f32}}$$

$$\frac{}{\vdash n : \texttt{i32}}$$

$$\frac{}{\vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{\vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\vdash v_i : \tau_i \quad i = [1; n]}{\vdash \ (v_1, \cdots, v_n) \ : \ (\tau_1, \cdots, \tau_n)}$$

$$\frac{\vdash v_i : \tau \quad i = [1; n]}{\vdash \ [v_1, \cdots, v_n] \ : \ [] \tau}$$

Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : []\tau \quad \Gamma \vdash e_i : \texttt{int} \ \ i = [1, 2]}{\Gamma \vdash e \, [e_1 : e_2] \, : []\tau}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \textbf{let} \ x = e \ \textbf{in} \ e' : \tau'}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \ (\tau_1, \cdots, \tau_n) \\ \Gamma, x_1 : \tau_1, \cdots, x_n : \tau_n \vdash e' : \tau\end{array}}{\Gamma \vdash \textbf{let} \ \ (x_1, \cdots, x_n) \ = e \ \textbf{in} \ e' : \tau}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad i = [1; n]}{\Gamma \vdash \ (e_1, \cdots, e_n) \ : \ (\tau_1, \cdots, \tau_n)}$$

$$\frac{\Gamma \vdash e_i : \tau \quad i = [1; n]}{\Gamma \vdash \ [e_1, \cdots, e_n] \ : \ [] \tau}$$

$$\frac{\vdash v : \tau}{\Gamma \vdash v : \tau}$$

$$\frac{\Gamma(f) = (\tau_1, \cdots, \tau_n) \to \tau \quad \Gamma \vdash e_i : \tau_i \;\; i = [1; n]}{\Gamma \vdash f\ e_1 \cdots e_n : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_i : \tau_i \;\; i = [1; 2] \\ \text{TypeOf}(\textit{binop}) \geq \tau \quad \tau = \tau_1 \to \tau_2 \to \tau'\end{array}}{\Gamma \vdash e_1\ \textit{binop}_\tau\ e_2 : \tau'}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_i : \tau_i \;\; i = [1; n] \\ \text{TypeOf}(\textit{op}) \geq \tau \\ \tau = \tau_1 \to \cdots \to \tau_n \to \tau'\end{array}}{\Gamma \vdash \textit{op}_\tau\ e_1\ \cdots\ e_n : \tau'}$$

$$\frac{\Gamma \vdash e : []\tau \qquad \Gamma \vdash e' : \texttt{int}}{\Gamma \vdash e\,[e']\ : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash F : \tau_\text{f} \quad \Gamma \vdash e_i : \tau_i \;\; i = [1; n] \\ \text{TypeOf}(\textit{soac}) \geq \tau_\text{f} \to \tau_1 \to \cdots \to \tau_n \to \tau\end{array}}{\Gamma \vdash \textit{soac}\ F\ e_1\ \cdots\ e_n : \tau}$$

Functions $\boxed{\Gamma \vdash F : \tau}$

$$\frac{\Gamma, x_1 : \tau_1\ \cdots\ x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \backslash x_1 : \tau_1\ \cdots\ x_n : \tau_n \to e : \tau_1 \to \cdots \to \tau_n \to \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \\ \text{TypeOf}(\textit{binop}) \geq \tau_1 \to \tau_2 \to \tau\end{array}}{\Gamma \vdash e\ \textit{binop} : \tau_2 \to \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_2 \\ \text{TypeOf}(\textit{binop}) \geq \tau_1 \to \tau_2 \to \tau\end{array}}{\Gamma \vdash \textit{binop}\ e : \tau_1 \to \tau}$$

Programs $\boxed{\Gamma \vdash P : \Gamma'}$

$$\frac{\Gamma \vdash e : \tau \qquad x \notin \text{Dom}(\Gamma)}{\Gamma \vdash \texttt{let}\ x = e : \{x : \tau\}}$$

$$\frac{\Gamma \vdash P_1 : \Gamma_1 \quad \Gamma + \Gamma_1 \vdash P_2 : \Gamma_2}{\Gamma \vdash P_1\ P_2 : \Gamma_1 + \Gamma_2}$$

$$\frac{\Gamma, x_1 : \tau_1, \cdots, x_n : \tau_n \vdash e : \tau \quad f \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash \texttt{let}\ f\ (x_1, \cdots, x_n) = e : \{f : (\tau_1, \cdots, \tau_n) \to \tau\}}$$

For brevity, we have eluded some of the typing rules and we leave it to the reader to create typing rules for `rearrange`, `shape`, `reshape`, `loop-for`, `loop-while`, and array ranging (`e[i:j:o]`).

6.3 Futhark Evaluation Semantics

In this section we develop a simple evaluation semantics for Futhark programs. The semantics is presented as a *big step* evaluation function that takes as parameter an expression and gives as a result a value. A *soundness property* states that if a program P is well-typed and contains a function *main* of type $() \to \tau$, then, if evaluation of the program results in a value v, the value v has type τ.

To ease the presentation, we treat the evaluation function as being implicitly parameterised by the program P.

The semantics of types yields their natural set interpretations:

$$\begin{aligned}
[\![\,\texttt{i32}\,]\!] &= Z \\
[\![\,\texttt{i32}\,]\!] &= R \\
[\![\,\texttt{bool}\,]\!] &= \{\texttt{true}, \texttt{false}\} \\
[\![\,(\tau_1, \cdots, \tau_n)\,]\!] &= [\![\,\tau_1\,]\!] \times \cdots \times [\![\,\tau_n\,]\!] \\
[\![\,[]\tau\,]\!] &= N \to [\![\,\tau\,]\!] \\
[\![\,\tau_1 \to \tau_2\,]\!] &= [\![\,\tau_1\,]\!] \to [\![\,\tau_2\,]\!]
\end{aligned}$$

For ease of presentation, we consider a syntactic vector value $[v_1, \cdots, v_n]$ equal to the projection function on the vector, returning a default value of the underlying type for indexes greater than $n - 1$ (zero-based interpretation).

For built-in operators op_τ, annotated with their type instance τ according to the typing rules, we assume a semantic function $[\![\,op_\tau\,]\!] : [\![\,\tau\,]\!]$. As an examples, we assume $[\![\,+_{\texttt{i32}\to\texttt{i32}\to\texttt{i32}}\,]\!] : Z \to Z \to Z$.

When e is some expression, we write $e[v_1/x_1, \cdots, v_n/x_n]$ to denote the simultaneous substitution of $v_1, \cdots, v_n$ for $x_1, \cdots, x_n$ (after appropriate renaming of bound variables) in e.

Evaluation of an expression e is defined by an evaluation function $[\![\,\cdot\,]\!] : \mathrm{Exp} \to \mathrm{Val}$. The function is defined in a mutually recursive fashion with an auxiliary utility function $\langle\, F \,\rangle$ for extracting

SOAC function parameters. We first give the definition for $[\![\cdot]\!]$:

$$
\begin{aligned}
[\![f\ e_1\ \cdots\ e_n]\!] &= [\![e[[\![e_1]\!]/x_1 \cdots [\![e_n]\!]/x_n]]\!] \\
&\quad \text{where } \texttt{let}\ f\ x_1\ \cdots x_n = e \in P \\
[\![v]\!] &= v \\
[\![e\,\texttt{[}e'\texttt{]}]\!] &= [\![e]\!]([\![e']\!]) \\
[\![\texttt{let}\ x = e\ \texttt{in}\ e']\!] &= [\![e'[[\![e]\!]/x]]\!] \\
[\![\texttt{let}\ (x_1, \cdots, x_n) = e\ \texttt{in}\ e']\!] &= [\![e'[v_1/x_1 \cdots v_n/x_n]]\!] \\
&\quad \text{where } [\![e]\!] = (v_1, \cdots, v_n) \\
[\![\texttt{[}e_1, \cdots, e_n\texttt{]}]\!] &= \texttt{[}[\![e_1]\!], \cdots, [\![e_n]\!]\texttt{]} \\
[\![(e_1, \cdots, e_n)]\!] &= ([\![e_1]\!], \cdots, [\![e_n]\!]) \\
[\![e_1\ \mathit{binop}_\tau\ e_2]\!] &= [\![\mathit{binop}_\tau]\!]\ [\![e_1]\!]\ [\![e_2]\!] \\
[\![\mathit{op}_\tau\ e_1 \cdots e_n]\!] &= [\![\mathit{op}_\tau]\!]\ [\![e_1]\!]\ \cdots\ [\![e_n]\!] \\
[\![\texttt{map}\ F\ e_1 \cdots e_m]\!] &= [\![\texttt{[}e'[v_1^1/x_1 \cdots v_1^m/x_m], \cdots, e'[v_n^1/x_n \cdots v_n^m/x_m]\texttt{]}]\!] \\
&\quad \text{where } \lambda x_1 \cdots x_m.e' = \langle\, F \,\rangle \\
&\quad \text{and } \texttt{[}v_1^i, \cdots, v_n^i\texttt{]} = [\![e_i]\!] \quad i = [1..m]
\end{aligned}
$$

Given a SOAC function parameter F, we define the utility *extraction function*, $\langle\, F \,\rangle$, as follows:

$$
\begin{aligned}
\langle\, \backslash x_1 \cdots x_n \to e \,\rangle &= \lambda x_1 \cdots x_n.e \\
\langle\, \mathit{binop}\ e \,\rangle &= \lambda x.x\ \mathit{binop}\ v \\
&\quad \text{where } v = [\![e]\!] \\
\langle\, e\ \mathit{binop} \,\rangle &= \lambda x.v\ \mathit{binop}\ x \\
&\quad \text{where } v = [\![e]\!]
\end{aligned}
$$

Type soundness is expressed by the following proposition:

Proposition: Futhark Type Soundness

If $\vdash P : \Gamma$ and $\Gamma(\mathit{main}) = () \to \tau$ and $[\![\ \mathit{main}\ \texttt{()}\ ]\!] = v$ then $\vdash v : \tau$.

Notice that we have glanced over the concept of bounds checking by assuming that arrays with elements of type τ are implemented as total functions from N to $[\![\tau]\!]$.

6.4 Work and Span

In this section we give a cost model for Futhark in terms of functions for determining the total *work* done by a program, in terms of operations done by the big-step evaluation semantics, and the *span* of the program execution, in terms of the maximum depth of the computation, assuming an infinite amount of parallelism in the SOAC computations. The functions for work and span, denoted by

$W : \text{Exp} \to N$ and $S : \text{Exp} \to N$ are given below. The functions are defined independently, although they make use of the evaluation function $[\![\cdot]\!]$. We have given the definitions for the essential SOAC functions, namely `map` and `reduce`. The definitions for the remaining SOACs follow the same lines as the definitions for `map` and `reduce`.

Work (W)

$$
\begin{aligned}
W(v) &= 1 \\
W(\texttt{let}\ x = e\ \texttt{in}\ e') &= W(e) + W(e'[[\![e]\!]/x]) + 1 \\
W(\texttt{let}\ (x_1, ..., x_n) = e\ \texttt{in}\ e') &= \text{let } [v_1, ..., v_n] = [\![e]\!] \\
&\quad \text{in } W(e) + W(e'[v_1/x_1, \cdots, v_n/x_n]) + 1 \\
W(\texttt{[}e_1, \cdots, e_n\texttt{]}) &= W(e_1) + \ldots + W(e_n) + 1 \\
W(\texttt{(}e_1, \cdots, e_n\texttt{)}) &= W(e_1) + \ldots + W(e_n) + 1 \\
W(f\ e_1 \cdots e_n) &= W(e_1) + \ldots + W(e_n) + W(e[[\![e_1]\!]/x_1, \cdots [\![e_n]\!]/x_n]) + 1 \\
&\quad \text{where}(\texttt{let}\ f\ x_1\ \cdots\ x_n\ =\ e)\ \in\ P \\
W(e_1 \mathit{binop} e_2) &= W(e_1) + W(e_2) + 1 \\
W(\texttt{map}\ F\ e) &= \text{let } [v_1, \cdots, v_n] = [\![e]\!] \\
&\quad \lambda x.e' = \langle\, F \,\rangle \\
&\quad \text{in } W(e) + W(e'[v_1/x]) + \ldots + W(e'[v_n/x]) \\
W(\texttt{reduce}\ F\ e'\ e'') &= \text{let } [v_1, \cdots, v_n] = [\![e'']\!] \\
&\quad \lambda x\ x'.e = \langle\, F \,\rangle \\
&\quad \text{in } W(e') + W(e'') + W(e[v_1/x, v_n/x']) \times n + 1 \\
&\quad \text{assuming } W(e[v_1/x, v_n/x']) \text{ indifferent to } v_1 \text{ and } v_n \\
W(\texttt{iota}\ e) &= W(e) + n \quad \text{where } n = [\![e]\!]
\end{aligned}
$$

Span (S)

$$
\begin{aligned}
S(v) &= 1 \\
S(\texttt{let}\ x\ =\ e\ \texttt{in}\ e') &= S(e) + S(e'[[\![\, e \,]\!]/x]) + 1 \\
S(\texttt{let}\ (x_1, ..., x_n)\ =\ e\ \texttt{in}\ e') &= \text{let } [v_1, ..., v_n] = [\![\, e \,]\!] \\
&\quad \text{in } S(e) + S(e'[v_1/x_1, \cdots, v_n/x_n]) \\
S([e_1, \cdots, e_n]) &= S(e_1) + \ldots + S(e_n) + 1 \\
S((e_1, \cdots, e_n)) &= S(e_1) + \ldots + S(e_n) + 1 \\
S(f e_1 \cdots e_n) &= S(e_1) + \ldots + S(e_n) + S(e[[\![\, e_1 \,]\!]/x_1, \cdots [\![\, e_n \,]\!]/x_n]) + 1 \\
&\quad \text{where } (\texttt{let}\ f\ x_1\ \cdots\ x_n\ =\ e)\ \in\ P \\
S(e_1\ \textit{binop}\ e_2) &= S(e_1) + S(e_2) + 1 \\
S(\texttt{map}\ F\ e) &= \text{let } [v_1, \cdots, v_n] = [\![\, e \,]\!] \\
&\quad \lambda x.e' = \langle\, F \,\rangle \\
&\quad \text{in } S(e) + \textit{max}(S(e'[v_1/x]), \ldots, S(e'[v_n/x])) + 1 \\
S(\texttt{reduce}\ F\ e'\ e'') &= \text{let } [v_1, \cdots, v_n] = [\![\, e'' \,]\!] \\
&\quad \lambda x\ x'.e = \langle\, F \,\rangle \\
&\quad \text{in } S(e') + S(e'') + S(e[v_1/x, v_n/x']) \times \ln n + 1 \\
&\quad \text{assuming } S(e[v_1/x, v_n/x']) \text{ indifferent to } v_1 \text{ and } v_n \\
S(\texttt{iota}\ e) &= S(e) + 1
\end{aligned}
$$

6.5 Reduction by Contraction

In this section, we shall investigate an implementation of reduction using the general concept of *contraction*, which is the general algorithmic trick of solving a particular problem by first making a *contraction step*, which simplifies the problem size, and then repeating the contraction algorithm until a final result is reached *[Org16]*.

The reduction algorithm that we shall implement assumes an associative reduction operator $\oplus : A \to A \to A$, a neutral element of type A, and a vector v of size 2^n, containing elements of type A. If $\text{size}(v) = 1$, the algorithm returns the single element. Otherwise, the algorithm performs a contraction by splitting the vector in two and applies the reduction operator elementwise on the two subvectors, thereby obtaining a contracted vector, which is then used as input to a recursive call to the algorithm. In Futhark, the function can be implemented as follows:

```
let red (xs : []i32) : i32 =
  let xs = loop xs=padpow2 0 xs while length xs > 1 do
    let n = length xs / 2
    in map2 (+) xs[0:n] xs[n:2*n]
```

The function specializes the reduction operator $\oplus$ to be `+` and the neutral element to be `0`. The function first pads the argument vector `xs` with neutral elements to ensure that its size is a power of two. It then implements a sequential loop with the contraction step as its loop body, implemented by a parallel `map` over an appropriately split input vector.

The auxiliary function for padding the input vector is implemented by the following code:

```
-- Find the smallest power of two greater than n
let nextpow2 (n:i32) : i32 =
  loop a=2 while a < n do 2*a

-- Pad a vector to make its size a power of two
let padpow2 [n] (ne: i32) (v:[n]i32) : []i32 =
  concat v (replicate (nextpow2 n - n) ne)
```

6.5.1 Determining Work and Span

To determine the work and span of the algorithm `red`, we first determine the work and span for `padpow2`, for which we again need to determine the work and span for `nextpow2`. From simple inspection we have $W(\texttt{nextpow2 n}) = S(\texttt{nextpow2 n}) = O(\log \texttt{n})$. Now, from the definition of W and S and because $\texttt{nextpow2 n} \leq 2\,\texttt{n}$, we have

$$W(\texttt{padpow2 ne v}) = W(\texttt{concat}\ v\ \texttt{(replicate (nextpow2 n - n) ne)}) = O(\texttt{n})$$

and

$$S(\texttt{padpow2 ne v}) = O(\log \texttt{n})$$

where $\texttt{n} = \text{size}\ \texttt{v}$.

Each loop iteration in has span $O(1)$. Because the loop is iterated at-most $\log(2\,\texttt{n})$ times, we have (where $\texttt{n} = \text{size}\,\texttt{v}$)

$$\begin{aligned} W(\texttt{red v}) &= O(\texttt{n}) + O(\texttt{n}/2) + O(\texttt{n}/4) + \cdots + O(1) \;=\; O(\texttt{n}) \\ S(\texttt{red v}) &= O(\log \texttt{n}) \end{aligned}$$

It is an exercise for the reader to compare the performance of the reduction code to the performance of Futhark's built-in `reduce` SOAC (see *Benchmarking*).

6.6 Radix-Sort by Contraction

Another example of a contraction-based algorithm is radix-sort. Radix-sort is a non-comparison based sorting routine, which implements sorting by iteratively moving elements with a particular bit set to the beginning (or end) in the array. It turns out that this move of elements with the same

bit set can be parallelised. Thus, for arrays containing 32-bit unsigned integers, the sorting routine needs only 32 loop-iterations to sort the array. A central property of each step is that elements with identical bit values will not shift position. Depending on whether the algorithm consistently moves elements with the bit set to the end of the array or to the beginning of the array results in the array being sorted in either ascending or descending order.

6.6.1 Radix-Sort in Futhark

A radix-sort algorithm that sorts the argument vector in ascending order is shown below:

```
let rsort_step [n] (xs: [n]u32, bitn: i32): [n]u32 =
  let bits1 = map (\x -> (i32.u32 (x >> u32.i32 bitn)) & 1) xs
  let bits0 = map (1-) bits1
  let idxs0 = map2 (*) bits0 (scan (+) 0 bits0)
  let idxs1 = scan (+) 0 bits1
  let offs  = reduce (+) 0 bits0
  let idxs1 = map2 (*) bits1 (map (+offs) idxs1)
  let idxs  = map2 (+) idxs0 idxs1
  let idxs  = map (\x->x-1) idxs
  in scatter (copy xs) idxs xs

-- Radix sort algorithm, ascending
let rsort [n] (xs: [n]u32): [n]u32 =
  loop (xs) for i < 32 do rsort_step(xs,i)
```

The function `rsort_step` implements the contraction step that takes care of moving all elements with the `bitn` set to the end of the array. The main function `rsort` takes care of iterating the contraction step until the array is sorted (i.e., when the contraction step has been executed for all bits.) To appreciate the purpose of each data-parallel operation in the function, the table below illustrates how `rsort_step` takes care of moving elements with a particular bit set (bit 2) to the end of the array. The example assumes the current array (`xs`) contains the array `[2,0,6,4,2,1,5,9]`. Notice that the last three values all have their 0-bit set whereas the first five values have not.

Variable								
`xs`	$2^\dagger$	0	$2^\dagger$	4	$2^\dagger$	1	5	9
`bits1`	1	0	1	0	1	0	0	0
`bits0`	0	1	0	1	0	1	1	1
`scan (+) 0 bits0`	0	1	1	2	2	3	4	5
`idxs0`	0	1	0	2	0	3	4	5
`idxs1`	1	1	2	2	3	3	3	3
`idxs1'`	6	6	7	7	8	8	8	8
`idxs1''`	6	0	7	0	8	0	0	0
`idxs`	6	1	7	2	8	3	4	5
`map (-1) idxs`	5	0	6	1	7	2	3	4

By a straightforward analysis, we can argue that $W(\texttt{rsort v}) = O(\texttt{n})$, where $n = \text{length v}$; each of the operations in has work $O(\texttt{n})$ and `rsort_step` is called a constant number of times (i.e., 32 times). Similarly, we can argue that $S(\texttt{rsort v}) = O(\log \texttt{n})$, dominated by the SOAC calls in `rsort_step`.

6.7 Counting Primes

A variant of a contraction algorithm is an algorithm that first solves a smaller problem, recursively, and then uses this result to provide a solution to the larger problem. One such algorithm is a version of the Sieve of Eratosthenes that, to find the primes smaller than some n, first calculates the primes smaller than $\sqrt{n}$. It then uses this intermediate result for sieving away the integers in the range $\sqrt{n}$ up to n that are multiples of the primes smaller than $\sqrt{n}$.

Unfortunately, Futhark does not presently support recursion, thus, one needs to use a **`loop`** construct instead to implement the sieve. A Futhark program calculating the number of primes below some number n, also denoted in the literature as the π function, is shown below:

```
import "/futlib/math"

-- Find the first n primes
let primes (n:i32) : []i32 =
  let (acc, _) = loop (acc,c) = ([],2) while c < n+1 do
        let c2 = i32.min (c * c) (n+1)
        let is = map (+c) (iota(c2-c))
        let fs = map (\i ->
                      let xs = map (\p -> if i%p==0 then 1
                                          else 0) acc
                      in reduce (+) 0 xs) is
        -- apply the sieve
        let new = filter (\i -> 0 == unsafe fs[i-c]) is
        in (concat acc new, c2)
```

(continues on next page)

(continued from previous page)

```
  in acc

-- Return the number of primes less than n
let main (n:i32) : i32 =
  let ps = primes n in length ps
```

Notice that the algorithm applies a parallel sieve for each step, using a combination of maps and reductions. The work done by the algorithm is $O(n \log \log n)$ and the span is $O(\log \log n)$. The $\log \log n$ factor appears because the size of the problem is squared at each step, which is identical to doubling the exponent size at each step (i.e., the sequence $2^2, 2^4, 2^8, 2^{16}, \ldots, n$, where $n = 2^{2^m}$, for some positive m, has $m = \log \log n$ elements.)

Chapter 7

Fusion and List Homomorphisms

In this chapter, we outline the general SOAC reasoning principles that lie behind both the philosophy of programming with arrays in Futhark and the techniques used for allowing certain programs to have efficient parallel implementations. We shall discuss the reasoning principles in terms of Futhark constructs but introduce a few higher-order concepts that are important for the reasoning.

We first discuss the concept of *fusion*, which aims at eliminating intermediate arrays while still allowing the Futhark programmer to express an algorithm using simple SOACs and their associated reasoning principles.

We then introduce the concept of list homomorphism through a few examples.

7.1 Fusion

Fusion aims at reducing the overhead of unnecessary repeated control-flow or unnecessary temporary storage. In essence, fusion is defined in terms of a number of *fusion rules*, which specify how a Futhark (intermediate) expression can be transformed into a semantically equivalent expression.

The rules make use of the auxiliary higher-order functions for, for instance, function composition, presented in *Higher-Order Functions*.

The *first fusion rule*, $F1$, which says that the result of mapping an arbitrary function `f` over the result of mapping another arbitrary function `g` over some array `a` is identical to mapping the composed function `f <-< g` over the array `a`. The first fusion rule is also called map-map fusion and can simply be written

```
map f <-< map g = map (f <-< g)
```

Given that `f` and `g` denote the Futhark functions `\x -> e` and `\y -> e'`, respectively (possibly after renaming of bound variables), the *function product* of `f` and `g`, written `f <*> g`, is defined as `\(x,y) -> (f x, g y)`.

Now, given functions `f:a->b` and `g:a->c`, the *second fusion rule*, $F2$, which denotes horizontal fusion, is given by the following equation:

```
(map f <*> map g) <-< dup = map ((f <*> g) <-< dup)
```

Here `dup` is the Futhark function `\x -> (x,x)`.

The fusion rules that we have presented here generalise to functions that take multiple arguments by applying zipping, unzipping, currying, and uncurrying strategically. Notice that due to Futhark's strategy of automatically transforming arrays of tuples into tuples of arrays, the applications of zipping, unzipping, currying, and uncurrying have no effect at runtime.

Futhark applies a number of other fusion rules, which are based on the fundamental property that Futhark's internal representation is based on a number of composed constructs (e.g., named `scanomap` and `redomap`). These constructs turn out to fuse well with `map`.

7.2 Parallel Utility Functions

For use by other algorithms, a set of utility functions for manipulating and managing arrays is an important part of the tool box. We present a number of utility functions here, ranging from finding elements in an array to finding the maximum element and its index in an array.

7.2.1 Finding the Index of an Element in an Array

We device two different functions for finding an index in an array for which the content is identical to some given value. The first function, `find_idx_first`, takes a value `e` and an array `xs` and returns the smallest index `i` into `xs` for which `xs[i] = e`:

```
-- Return the first index i into xs for which xs[i] == e
let find_idx_first [n] (e:i32) (xs:[n]i32) : i32 =
  let es = map2 (\x i -> if x==e then i else n) xs (iota n)
  let res = reduce i32.min n es
  in if res == n then -1 else res
```

The second function, `find_idx_last`, also takes a value and an array but returns the largest index `i` into `xs` for which `xs[i] = e`:

```
-- Return the last index i into xs for which xs[i] == e
let find_idx_last [n] (e:i32) (xs:[n]i32) : i32 =
  let es = map2 (\x i -> if x==e then i else -1) xs (iota n)
  in reduce i32.max (-1) es
```

The above two functions make use of the auxiliary functions `i32.max` and `i32.min`.

7.2.2 Finding the Largest Element and its Index in an Array

Futhark allows for reduction operators to take tuples as arguments. This feature is exploited in the following function, which implements a homomorphism for finding the largest element and its index in an array:

```
-- Find the largest integer and its index in an array
let MININT : i32 = -10000000

let mx (m1:i32,i1:i32) (m2:i32,i2:i32) : (i32,i32) =
  if m1 > m2 then (m1,i1) else (m2,i2)

let maxidx [n] (xs: [n]i32) : (i32,i32) =
  reduce mx (MININT,-1) (zip xs (iota n))
```

The function is a *homomorphism [Bir87]*: For any x and y, and with $+\!\!+$ denoting array concatenation, there exists an associative operator $\oplus$ such that

$$\texttt{maxidx}(x +\!\!+ y) = \texttt{maxidx}(x) \oplus \texttt{maxidx}(y)$$

The operator $\oplus = \texttt{mx}$. We will leave it up to the reader to verify that the `maxidx` function will operate efficiently on large inputs.

7.3 Radix Sort Revisited

A simple radix sort algorithm was presented already in *Radix-Sort in Futhark*. In this section, we present two generalized versions of radix sort, one for ascending sorting and one for descending sorting. As a bonus, the sorting routines return both the sorted array and an index array that can be used to sort an array with respect to a permutation obtained by sorting another array. The generalised ascending radix sort is as follows:

```
-- Store elements for which bitn is not set first
let rs_step_asc [n] ((xs:[n]u32,is:[n]i32),bitn:i32) : ([n]u32,[n]i32)␣
↪=
  let bits1 = map (\x -> (i32.u32 (x >> u32.i32 bitn)) & 1) xs
  let bits0 = map (1-) bits1
  let idxs0 = map2 (*) bits0 (scan (+) 0 bits0)
  let idxs1 = scan (+) 0 bits1
  let offs  = reduce (+) 0 bits0    -- store idxs1 last
  let idxs1 = map2 (*) bits1 (map (+offs) idxs1)
  let idxs  = map (\x->x-1) (map2 (+) idxs0 idxs1)
  in (scatter (copy xs) idxs xs,
      scatter (copy is) idxs is)
```

(continues on next page)

(continued from previous page)

```
-- Radix sort - ascending
let rsort_asc [n] (xs: [n]u32) : ([n]u32,[n]i32) =
  let is = iota n
  in loop (p : ([n]u32,[n]i32)) = (xs,is) for i < 32 do
    rs_step_asc(p,i)
```

And the descending version as follows:

```
-- Store elements for which bitn is set first
let rs_step_desc [n] ((xs:[n]u32,is:[n]i32),bitn:i32) : ([n]u32,
↪[n]i32) =
  let bits1 = map (\x -> (i32.u32 (x >> u32.i32 bitn)) & 1) xs
  let bits0 = map (1-) bits1
  let idxs1 = map2 (*) bits1 (scan (+) 0 bits1)
  let idxs0 = scan (+) 0 bits0
  let offs  = reduce (+) 0 bits1    -- store idxs0 last
  let idxs0 = map2 (*) bits0 (map (+offs) idxs0)
  let idxs  = map (\x->x-1) (map2 (+) idxs1 idxs0)
  in (scatter (copy xs) idxs xs,
      scatter (copy is) idxs is)

-- Radix sort - descending
let rsort_desc [n] (xs: [n]u32) : ([n]u32,[n]i32) =
  loop (p : ([n]u32,[n]i32)) = (xs,iota n) for i < 32 do
    rs_step_desc(p,i)
```

Notice that in case of identical elements in the source vector, one cannot simply implement the ascending version by reversing the arrays resulting from calling the descending version.

7.4 Finding the Longest Streak

In this section, we shall demonstrate how to write a function for finding the longest streak of increasing numbers. Here is one possible implementation of the function:

```
-- Longest streak of increasing numbers
let streak [n] (xs: [n]i32) : i32  =
  -- find increments
  let ys = rotate 1 xs
  let is = (map2 (\x y -> if x < y then 1 else 0) xs ys)[0:n-1]
  -- scan increments
  let ss = scan (+) 0 is
  -- nullify where there is no increment
  let ss1 = map2 (\s i -> s*(1-i)) ss is
```

(continues on next page)

(continued from previous page)

```
  let ss2 = scan max 0 ss1
  -- subtract from increment scan
  let ss3 = map2 (-) ss ss2
  let res = reduce max 0 ss3
  in res
```

The following derivation shows how the algorithm works for a particular input, namely when `stream` is given the argument array `[1,5,3,4,2,6,7,8]`, in which case the algorithm should return the value 3:

Variable									
xs	=	1	5	3	4	2	6	7	8
ys	=	5	3	4	2	6	7	8	1
is	=	1	0	1	0	1	1	1	
ss	=	1	1	2	2	3	4	5	
ss	=	0	1	0	2	0	0	0	
ss2	=	0	1	1	2	2	2	2	
ss3	=	1	0	1	0	1	2	3	
res	=	3							

In *Finding the Longest Streak Using Segmented Scan* we present a simpler algorithm, which builds directly on the concept of a so-called segmented scan.

Chapter 8

Regular Flattening

In this chapter, we introduce the concept of regular *moderate flattening [HSE+17]*, which is the essential technique used for making regular nested parallel Futhark programs run efficiently in practice on parallel hardware such as GPUs.

We first introduce a number of parallel segmented operations, which are essential for dealing with nested parallelism. The segmented operations, it turns out, can be implemented using Futhark's standard SOAC parallel array combinators. In particular, it turns out that the `scan` operator is of critical importance in that it can be used to develop the notion of a *segmented scan* operation, an operation that, in its own right, is essential to many parallel algorithms. Based on the segmented scan operation and the other Futhark SOAC operations, we present a set of utility functions as well as their parallel implementations. The functions are used by the moderate flattening transformation presented in *Moderate Flattening*, but are also useful, as we shall see in *Irregular Flattening*, for the programmer to manage irregular parallelism through flattening transformations, performed manually by the programmer.

8.1 Segmented Scan

As mentioned, the segmented scan operation is quite essential for Futhark to flatten nested regular parallelism and for the programmer to flatten irregular nested parallel problems. The operation can be implemented with a simple scan using an associative function that operates on pairs of values *[Sch80][Ble90]*. Here is the definition of the segmented scan operation, hardcoded to work with addition:

```
-- Segmented scan with integer addition
let sgm_scan_add [n] (vals:[n]i32) (flags:[n]bool) : [n]i32 =
  let pairs = scan ( \ (v1,f1) (v2,f2) ->
                       let f = f1 || f2
                       let v = if f2 then v2 else v1+v2
```

(continues on next page)

(continued from previous page)

```
                        in (v,f) ) (0,false) (zip vals flags)
  let (res,_) = unzip pairs
  in res
```

We can make use of Futhark's support for higher-order functions and polymorphism to define a generic version of segmented scan that will work for other monoidal structures than addition on `i32` values:

```
-- Generic version of segmented scan
let sgm_scan 't [n] (g:t->t->t) (ne:t) (vals:[n]t) (flags:[n]bool) :␣
↪[n]t =
  let pairs = scan ( \ (v1,f1) (v2,f2) ->
                       let f = f1 || f2
                       let v = if f2 then v2 else g v1 v2
                       in (v,f) ) (ne,false) (zip vals flags)
  let (res,_) = unzip pairs
  in res
```

We leave it up to the reader to prove that, given an associative function `g`, (1) the operator passed to `scan` is associative and (2) `(ne, false)` is a neutral element for the operator.

8.1.1 Finding the Longest Streak Using Segmented Scan

In this section we revisit the problem of *Finding the Longest Streak* for finding the longest streak of increasing numbers. We show how we can make direct use of a segmented scan operation for solving the problem:

```
-- Longest streak of increasing numbers
let sgm_streak [n] (xs: [n]i32) : i32  =
  let ys = rotate 1 xs
  let is = (map2 (\x y -> if x < y then 1 else 0) xs ys)[0:n-1]
  let fs = map (==0) is
  let ss = sgm_scan_add is fs
  let res = reduce max 0 ss
  in res
```

The algorithm first constructs the `is` array, as in the previous algorithm, and then uses a segmented scan over a negation of this array over the unit-array to create the `ss3` vector directly. Here is a derivation of how the segmented-scan based algorithm works:

Variable									
xs	=	1	5	3	4	2	6	7	8
ys	=	5	3	4	2	6	7	8	1
is	=	1	0	1	0	1	1	1	
fs	=	0	1	0	1	0	0	0	
ss	=	1	0	1	0	1	2	3	
res	=	3							

The morale here is that the segmented scan operation provides us with a great abstraction.

8.2 Replicated Iota

The first utility function that we will present is called `replicated_iota`. Given an array of natural numbers specifying repetitions, the function returns an array of weakly increasing indices (starting from 0) and with each index repeated according to the repetition array. As an example, `replicated_iota [2,3,1,1]` returns the array `[0,0,1,1,1,2,3]`. The function is defined in terms of other parallel operations, including `scan`, `map`, `scatter`, and `segmented_scan`:

```
let replicated_iota [n] (reps:[n]i32) : []i32 =
  let s1 = scan (+) 0 reps
  let s2 = map (\i -> if i==0 then 0 else unsafe s1[i-1]) (iota n)
  let tmp = scatter (replicate (unsafe s1[n-1]) 0) s2 (iota n)
  let flags = map (>0) tmp
  in segmented_scan (+) 0 flags tmp
```

An example evaluation of a call to the function `replicated_iota` is provided below. Notice that in order to use this Futhark code with `futhark-opencl`, we need to prefix the array indexing in line 3 and line 4 with the `unsafe` keyword.

Args/Result								
reps	=	2	3	1	1			
s1	=	2	5	6	7			
s2	=	0	2	5	6			
replicate	=	0	0	0	0	0	0	0
tmp	=	0	0	1	0	0	2	3
flags	=	0	0	1	0	0	1	1
segmented_scan	=	0	0	1	1	1	2	3

8.3 Segmented Replicate

Another useful utility function is called `segmented_replicate`. Given a one-dimensional replication array containing natural numbers and a data array of the same shape, `segmented_replicate` returns an array of size equal to the sum of the values in the replication array with values from the data array replicated according to the corresponding replication values. As an example, a call `segmented_replicate [2,1,0,3,0] [5,6,9,8,4]` result in the array `[5,5,6,8,8,8]`. Here is the code that implements the function `segmented_replicate`:

```
let segmented_replicate [n] (reps:[n]i32) (vs:[n]i32) : []i32 =
  let idxs = replicated_iota reps
  in map (\i -> unsafe vs[i]) idxs
```

The `segmented_replicate` function makes use of the previously defined function `replicated_iota`. Notice the use of the `unsafe` keyword in the last line; it is necessary because Futhark cannot prove that the index `i` will always be within bounds of the array `vs`.

8.4 Segmented Iota

Another useful utility function is the function `segmented_iota` that, given a array of flags (i.e., booleans), returns an array of index sequences, each of which is reset according to the booleans in the array of flags. As an example, the expression:

```
segmented_iota [false,false,false,true,false,false,false]
```

returns the array `[0,1,2,0,1,2,3]`. The `segmented_iota` function can be implemented with the use of a simple call to `segmented_scan` followed by a call to `map`:

```
let segmented_iota [n] (flags:[n]bool) : [n]i32 =
  let iotas = segmented_scan (+) 0 flags (replicate n 1)
  in map (\x -> x-1) iotas
```

8.5 Indexes to Flags

Many segmented operations, such as `segmented_scan` takes as argument an array of boolean flags for specifying when new segments start. Often, only the sizes of segments are known, which means that it may come in useful to be able to transform an array of segment sizes to a corresponding array of boolean flags. Here is one possible parallel implementation of such an `idxs_to_flags` function:

```
let idxs_to_flags [n] (is : [n]i32) : []bool =
  let vs = segmented_replicate is (iota n)
  in map2 (!=) vs ([0] ++ vs[:length vs-1])
```

As an example use of the function, the expression `idxs_to_flags [2,1,3]` evaluates to the flag array `[false,false,true,true,false,false]`. Notice that the implementation also works in case some segments are of size zero.

8.6 Moderate Flattening

The flattening rules that we shall introduce here allow the Futhark compiler to generate parallel kernels for various code block patterns. In contrast to the general concept of flattening as introduced by Blelloch *[BHS+94]*, Futhark applies a technique called *moderate flattening [HSE+17]*, which does not cover arbitrary nested parallelism, but does cover well many regular nested parallel patterns. We shall come back to the issue of flattening irregular nested parallelism in *Irregular Flattening*.

In essence, moderate flattening works by matching compositions of fused constructs against a number of flattening rules. The aim is to merge (i.e., flatten) nested parallel operations into sequences of parallel operations. Although, such flattening is often possible, in particular due to an integrated transformation called vectorisation, there are situations where choices needs to be made. In particular, when a map is nested on top of a loop, we may choose to parallelise the outer map and sequentialise the inner loop, which on the GPU will amount to all threads running sequential loops in parallel. An alternative, when possible, will be to interchange the outer map and the loop and then sequentialise the outer loop (on the host) and parallelise the inner map, which will then be executed multiple times. It turns out that Futhark can make some guesses about which strategy to pursue based on possible information about the sizes of the arrays. An extension to the static concept moderate flattening, Futhark also supports a notion of flattening that generates multiple versions of flattened code, guarded by parameters that may be autotuned to achieve good performance for a range of different data sets *[HTEO19]*.

In the following we shall focus on the transformations performed by moderate flattening.

8.6.1 Vectorisation

Assuming `e'` contains SOACs, transform the expression

```
map (\x -> let y = e in e') xs
```

into the expression

```
let ys = map (\x -> e) xs
in map (\(x,y) -> e') (zip xs ys)
```

This transformation does not itself capture any nested parallelism but may enable other transformations by eliminating the inner `let`-expression.

8.6.2 Map-Map Nesting

Nested applications of `map` constructs are in essence transformed into a single `map` construct by (1) flattening the argument array, (2) applying the inner function on the flattened array, and (3) unflattening the concatenated results. This process can be repeated for multiple nested `map` constructs. It turns out that the administrative operations can be implemented with zero overhead.

8.6.3 Map-Scan Nesting

In case of an expression made up from a `map` construct appearing on top of a `scan` operation, the expression is transformed into a regular segmented scan operation. That is, the expression:

```
map (\xs -> scan f ne xs) xss
```

is transformed into the expression:

```
regular_segmented_scan f ne xss
```

Notice here that we assume the availability of a regular segmented scan operation of type:

```
val regular_segmented_scan 't [n] [m]: (t->t->t) -> t -> [n][m]t ->␣
↪[n][m]t
```

Internally, this function will use the inner size of the multi-dimensional argument array (i.e., `m`) to construct an appropriate flag vector suitable for the segmented scan. Again, for an in-depth discussion of how to implement a segmented scan operation on top of an ordinary scan operation, please consult *Segmented Scan*.

8.6.4 Map-Reduce Nesting

In case of a `map` construct appearing on top of a `reduce` operation, this expression is transformed into a regular segmented reduction *[LH17]*. That is, the expression:

```
map (\xs -> reduce f ne xs) xss
```

is transformed into the expression:

```
regular_segmented_reduce f ne xss
```

Notice here that we assume the availability of a regular segmented reduction operation of type:

```
val regular_segmented_reduce 't [n] : (t->t->t) -> t -> [n][]t -> [n]t
```

Internally, this function can be implemented based on the function `regular_segmented_scan` discussed above. Here is a simple definition::

```
let regular_segmented_reduce = map last <-< regular_segmented_scan
```

8.6.5 Map-Iota Nesting

A `map` over an `iota` expression can be transformed to the composition of the `segmented_iota` function defined in *Segmented Iota* and a function `idxs_to_flags`, which converts an array of indices to an array `fs` of boolean flags of size equal to the sum of the values in `xs` and with `true`-values in indexes specified by the prefix sums of the index values.

As an example, the expression `idxs_to_flags [2,1,3]` evaluates to the flag array `[false,false,true,true,false,false]`. Notice that the expression `idxs_to_flags [2,0,4]` evaluates to the same boolean vector as `idxs_to_flags [2,4]`. We shall not here give a definition of the `idxs_to_flags` function, but refer the reader to *Indexes to Flags*.

All in all, an expression of the form:

```
map iota xs
```

is transformed into:

```
(segmented_iota <-< idxs_to_flags) xs
```

8.6.6 Map-Replicate Nesting

Recall that `replicate` has the type:

```
val replicate 't : (n:i32) -> t -> [n]t
```

A `map` over a `replicate` expression takes the form:

```
map (\x -> replicate n x) xs
```

where `n` is invariant to `x`. Such an expression can be transformed into the expression:

```
segmented_replicate (replicate (length xs) n) xs
```

As an example, consider the expression `map (replicate 2) [8,5,1]`. This expression is transformed into the expression:

```
segmented_replicate (replicate 3 2) [8,5,1]
```

which evaluates to `[8,8,5,5,1,1]`. Notice that the subexpression `replicate 3 2` evaluates to `[2,2,2]`.

Chapter 9

Pseudo-Random Numbers and Monte Carlo Sampling Methods

Pseudo-random number generation and Monte Carlo sampling are concepts that apply to a large number of application areas. In a data-parallel setting, these concepts require special treatment beyond the usual sequential methods. In this chapter, we first present a Futhark package, called `cpprandom` for generating pseudo-random numbers in parallel. We then present a Futhark package, called `sobol`, for generating Sobol sequences, which are examples of so-called low-discrepancy sequences, sequences that make numerical multi-dimensional integration converge faster than if pseudo-random numbers were used.

9.1 Generating Pseudo-Random Numbers

The `cpprandom` package is inspired by the C++ library `<random>`, which is very elaborate, but also very flexible. Due to Futhark's purity, it is up to the programmer to explicitly manage the state of the pseudo-random number engine (the RNG state). In particular, it is the programmer's responsibility to ensure that the same state is not used more than once (unless that is what is desired).

The following program constructs a uniform distribution of single precision floats using `minstd_rand` as the underlying RNG engine.

```
module dist = uniform_real_distribution f32 minstd_rand

let rng = minstd_rand.rng_from_seed [123]
let (rng, x) = dist.rand (1,6) rng
```

The `dist` module is constructed at the program top level, while we use it at the expression level. We use the `minstd_rand` module for initialising the random number state using a seed, and then we pass that state to the `rand` function in the generated distribution module, along with a

description of the distribution we desire. We get back not just the random number, but also the new state of the engine.

The `dist.rand` function, coming from `uniform_real_distribution`, simply takes a pair of numbers describing the range. Consider instead the following code:

```
module norm_dist = normal_distribution f32 minstd_rand

let (rng, y) = norm_dist.rand {mean=50, stddev=25} rng
```

In contrast to `dist.rand`, the `norm_dist.rand` function, coming from `normal_distribution` takes a record specifying the mean and the standard deviation. Since both `dist` and `norm_dist` have been initialised with the same underlying `rng_engine`, we can reuse the same RNG state. Such reuse is often convenient when a program needs to generate random numbers from several different distributions, as we still only have to manage a single RNG state.

9.1.1 Parallel random numbers

Random number generation is inherently sequential. The `rand` functions take an RNG state as input and produce a new RNG state. This dependence creates challenges when we wish to `map` a function `f` across some array `xs`, and each application of the function must produce some random numbers. We generally don't want to pass the exact same state to every application, as that means each element will see the exact same stream of random numbers. The common procedure is to use `split_rng`, which creates any number of RNG states from one, and then pass one to each application of `f`:

```
let rngs = minstd_rand.split_rng n rng
let (rngs, ys) = unzip (map2 f rngs xs)
let rng = minstd.rand.join_rngs rngs
```

We assume here that the function `f` returns not just the result, but also the new RNG state. Generally, all functions that accept random number states should behave like this. We subsequently use `join_rngs` to combine all resulting states back into a single state. Thus, parallel programming with random numbers involves frequently splitting and rejoining RNG states. For most RNG engines, these operations are generally very cheap.

9.2 Low-Discrepancy Sequences

The Futhark package `sobol` is a package for generating Sobol sequences, which are examples of so-called *low-discrepancy sequences*, sequences that, when combined with Monte-Carlo methods, make numeric integration converge faster than if ordinary pseudo-random numbers are used and are more flexible than if uniform sampling techniques are used. Sobol sequences may be multi-dimensional and a key property of using Sobol sequences is that we can freely choose the number

of points that should span the multi-dimensional space. In contrast, if we set out to use a simpler uniform sampling technique for spanning two dimensions, we can only span the space properly if we choose the number of points to be on the form x^2, for some natural number x. This spanning problem becomes worse for higher dimensions.

As an example, we shall see how we can use Sobol sequences together with Monte-Carlo simulation to compute the value of π. We shall also see that doing so will result in faster convergence towards the true value of π compared to if pseudo-random numbers are used.

To calculate an approximation to the value of π, we will use a simple dart-throwing approach. We will throw darts at a 2 by 2 square, centered around the origin, and then establish the ratio between the number of darts hitting within the unit circle with the number of darts hitting the square. This ratio multiplied with 4 will be our approximation of π. The more darts we throw, the better our approximation, assuming that the darts we throw hit the board somewhat evenly. To calculate whether a particular dart, thrown at the point (x, y), is within the unit circle, we can apply the standard Pythagoras formula:

$$\pi \approx \frac{4}{N} \sum_{i=1}^{N} \begin{cases} 1 & \text{if } x_i^2 + y_i^2 < 1 \\ 0 & \text{otherwise} \end{cases}$$

For the actual throwing of darts, we need to establish N pairs of numbers, each in the interval [-1;1]. Now, it turns out that it matters significantly how we choose to throw the darts. Some obvious choices would be to throw the darts in a regular grid (i.e., *uniform sampling*), or to choose points using a pseudo-random number generator.

The Futhark package makes essential use of an *independent formula* for calculating, independently, the n'th Sobol number. However, even though such a formula is essential for achieving parallelism, it performs poorly compared to the more efficient *recurrent formula*, which makes it possible to calculate the n'th Sobol number if we know the previous Sobol number. The Futhark package makes essential use of both formulas. The calculation of a sequence of Sobol numbers depends on a set of direction vectors, which are also provided by the package.

The key functionality of the package comes in the form of a higher-order module *Sobol*, which takes as arguments a direction vector module and a module specifying the dimensionality of the generated Sobol numbers:

```
module type sobol_dir  = { ... }
module sobol_dir       : sobol_dir  -- file sobol-dir-50, e.g.

module type sobol = {
  val D : i32
  val norm : f64
  val independent : i32 -> [D]u32
  val recurrent   : i32 -> [D]u32 -> [D]u32
  val sobol       : (n: i32) -> [n][D]f64
}
module Sobol : (DM : sobol_dir) -> (X : { val D : i32 }) -> sobol
```

For estimating the value of π, we will need a two-dimensional Sobol sequence, thus we apply the *Sobol* higher-order module to the direction vector module that works for up-to 50 dimensions and a module specifying a dimensionality of two:

```
import "lib/github.com/diku-dk/sobol/sobol-dir-50"
import "lib/github.com/diku-dk/sobol/sobol"

module sobol = Sobol sobol_dir { let D = 2 }
```

We can now complete the program by writing a *main* function that computes an array of Sobol numbers of a size given by the parameter given to *main* and feed this array into a function that will compute the estimation of π using the function shown above:

```
let sqr (x:f64) = x * x

let in_circle (p:[2]f64) : bool =
  sqr p[0] + sqr p[1] < 1.0f64

let pi_arr [n] (arr: [n][2]f64) : f64 =
  let bs = map (i32.bool <-< in_circle) arr
  let sum = reduce (+) 0 bs
  in 4f64 * r64 sum / f64.i32 n

let main (n:i32) : f64 =
  sobol.sobol n |> pi_arr
```

The use of Sobol numbers for estimating π turns out to be about three times slower than using a uniform grid on a standard GPU. However, it converges towards π equally well (with increasing N) and is superior for larger dimensions *[HEO18]*. In general, there are other good reasons to avoid uniform sampling in relation to Monte-Carlo methods.

Chapter 10

Irregular Flattening

In this chapter, we investigate a number of challenging irregular algorithms, which cannot be dealt with directly using Futhark's moderate flattening technique discussed in *Moderate Flattening*.

10.1 Flattening by Expansion

For dealing with large non-regular problems, we need ways to regularise the problems so that they become tractable with the regular parallel techniques that we have seen demonstrated previously. One way to regularise a problem is by *padding* data such that the data fits a regular parallel schema. However, by doing so, we run the risk that the program will use too many parallel resources for computations on the padding data. This problem will arise, in particular, if the data is very irregular. As a simple, and also visualisable, example, consider the task of determining the points that make up a number of line segments given by sets of two points in a 2D grid. Whereas we may easily devise an algorithm for determining the grid points that make up a single line segment, it is not immediately obvious how we can efficiently regularise the problem of drawing multiple line segments, as each line segment will end up being represented by a different number of points. If we choose to implement a padding regularisation scheme by introducing a notion of ''an empty point'', each line can be represented as the same number of points, which will allow us to map over an array of such line points for processing the lines using regular parallelism. However, the cost we pay is that even the smallest line will be represented as the same number of points as the longest line.

Another strategy for regularisation is to *flatten* the irregular parallelism into regular parallelism and use segmented operations to process each particular object. It turns out that there, in many cases, is a simple approach to implement such flattening, using, as we shall see, a technique called *expansion*, which will take care of all the knitty gritty details of the flattening. The expansion approach is centered around a function that we shall call `expand`, which, as the name suggests, expands a source array into a longer target array, by expanding each individual source element into multiple target elements, which can then be processed in parallel.

For implementing the `expand` function using only parallel operations, we shall make use of the segmented helper functions defined in *Regular Flattening*. In particular, we shall make use of the functions `replicated_iota`, `segmented_replicate`, and `segmented_iota`.

Here is the generic type of the `expand` function:

```
val expand 'a 'b : (sz: a -> i32) -> (get: a -> i32 -> b) -> []a -> []b
```

The function expands a source array into a target array given (1) a function that determines, for each source element, how many target elements it expands to and (2) a function that computes a particular target element based on a source element and the target element number associated with the source. As an example, the expression `expand (\x->x) (*) [2,3,1]` returns the array `[0,2,0,3,6,0]`. The function is defined as follows:

```
let expand 'a 'b (sz: a -> i32) (get: a -> i32 -> b) (arr:[]a) : []b =
  let szs = map sz arr
  let idxs = replicated_iota szs
  let iotas = segmented_iota (map2 (!=) idxs (rotate (i32.negate 1)␣
↪idxs))
  in map2 (\i j -> unsafe get arr[i] j) idxs iotas
```

10.2 Drawing Lines

In this section we demonstrate how to apply the flattening-by-expansion technique for obtaining a work efficient line drawing routine that draws lines fully in parallel. The technique resembles the development by Blelloch *[Ble90]* with the difference that it makes use of the `expand` function defined in the previous section. Given a number of line segments, each defined by its end points (x_1, y_1) and (x_2, y_2), the algorithm will find the set of all points constituting all the line segments.

We first present an algorithm that will find all points that constitutes a single line segment. For computing this set, observe that the number of points that make up the constituting set is the maximum of $|x_2 - x_1|$ and $|y_2 - y_1|$, the absolute values of the difference in x-coordinates and y-coordinates, respectively. Using this observation, the algorithm can independently compute the constituting set by first calculating the proper direction and slope of a line, relative to a particular starting point.

The simple line drawing routine is given as follows:

```
-- Finding points on a line
type point = (i32,i32)
type line = (point,point)
type points = []point

let compare (v1:i32) (v2:i32) : i32 =
  if v2 > v1 then 1 else if v1 > v2 then -1 else 0
```

(continues on next page)

(continued from previous page)

```
let slope ((x1,y1):point) ((x2,y2):point) : f32 =
  if x2==x1 then if y2>y1 then 1f32 else -1f32
                 else r32(y2-y1) / f32.abs(r32(x2-x1))

let linepoints ((x1,y1):point, (x2,y2):point) : points =
  let len = 1 + i32.max (i32.abs(x2-x1)) (i32.abs(y2-y1))
  let xmax = i32.abs(x2-x1) > i32.abs(y2-y1)
  let (dir,sl) =
    if xmax then (compare x1 x2, slope (x1,y1) (x2,y2))
    else (compare y1 y2, slope (y1,x1) (y2,x2))
  in map (\i -> if xmax
                then (x1+i*dir,
                      y1+i32.f32(f32.round(sl*r32(i))))
                else (x1+i32.f32(f32.round(sl*r32(i))),
                      y1+i*dir)) (iota len)
```

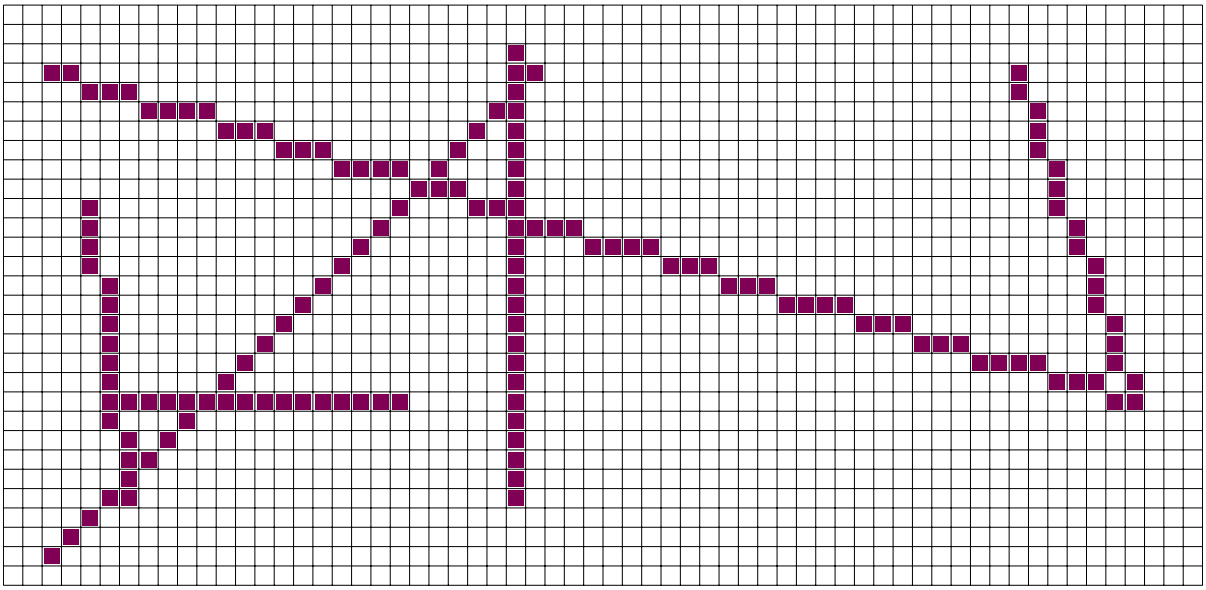
Futhark code that uses the `linepoints` function for drawing concrete lines is shown below:

```
-- Write to grid
let update [h] [w] [n] (grid: [h][w]i32)(xs:[n]i32)(ys:[n]i32):␣
↪[h][w]i32 =
  let is = map2 (\x y -> w*y+x) xs ys
  let flatgrid = flatten grid
  let ones = map (\ _ -> 1) is
  in unflatten h w (scatter (copy flatgrid) is ones)

-- Sequential algorithm for drawing multiple lines
let drawlines [h] [w] [n] (grid: *[h][w]i32) (lines:[n]line) :␣
↪[h][w]i32 =
  loop (grid) for i < n do -- find points for line i
    let (xs,ys) = unzip (linepoints (lines[i]))
    in update grid xs ys

-- Draw lines on a 70 by 30 grid
let main : [][]i32 =
  let height:i32 = 30
  let width:i32 = 70
  let grid : *[][]i32 = replicate height (replicate width 0)
  let lines = [((58,20),(2,3)),((27,3),(2,28)),((5,20),(20,20)),
               ((4,10),(6,25)),((26,25),(26,2)),((58,20),(52,3))]
  in drawlines grid lines
```

The function `main` sets up a grid and calls the function `drawlines`, which takes care of sequentially updating the grid with constituting points for each line, computed using the `linepoints` function. The resulting points look like this:



An unfortunate problem with the line drawing routine shown above is that it draws the lines sequentially, one by one, and therefore makes only very limited use of a GPU's parallel cores. There are various ways one may mitigate this problem. One way could be to use `map` to draw lines in parallel. However, such an approach will require some kind of padding to ensure that the map function will compute data of the same length, no matter the length of the line. A more resource aware approach will apply a flattening technique for computing all points defined by all lines simultaneously. Using the `expand` function defined in the previous section, all we need to do to implement this approach is to provide (1) a function that determines for a given line, the number of points that make up the line and (2) a function that determines the `n`'th point of a particular line, given the index `n`. The code for such an approach looks as follows:

```
-- Parallel flattened algorithm for turning lines into
-- points, using expansion.

let points_in_line ((x1,y1),(x2,y2)) =
  i32.(1 + max (abs(x2-x1)) (abs(y2-y1)))

let get_point_in_line ((p1,p2):line) (i:i32) =
  if i32.abs(p1.1-p2.1) > i32.abs(p1.2-p2.2)
  then let dir = compare (p1.1) (p2.1)
       let sl = slope p1 p2
       in (p1.1+dir*i,
           p1.2+i32.f32(f32.round(sl*r32 i)))
    else let dir = compare (p1.2) (p2.2)
         let sl = slope (p1.2,p1.1) (p2.2,p2.1)
         in (p1.1+i32.f32(f32.round(sl*r32 i)),
             p1.2+i*dir)

let drawlines [h][w][n] (grid:*[h][w]i32)
```

(continues on next page)

(continued from previous page)

```
                          (lines:[n]line) :[h][w]i32 =
  let (xs,ys) = expand points_in_line get_point_in_line lines
              |> unzip
  in update grid xs ys
```

Notice that the function `get_point_in_line` distinguishes between whether the number of points in the line is counted by the x-axis or the y-axis. Notice also that the flattening technique can be applied only because all lines have the same color. Otherwise, when two lines intersect, the result would be undefined, due to the fact that `scatter` results in undefined behaviour when multiple values are written into the same location of an array.

10.3 Drawing Triangles

Another example of an algorithm worthy of flattening is an algorithm for drawing triangles. The algorithm that we present here is based on the assumption that we already have a function for drawing multiple horizontal lines in parallel. Luckily, we have such a function! The algorithm is based on the property that any triangle can be split into an *upper triangle* with a horizontal baseline and a *lower triangle* with a horizontal ceiling. Just as the algorithm for drawing lines makes use of the `expand` function defined earlier, so will the flattened algorithm for drawing triangles. A triangle is defined by the three points representing the corners of the triangle:

```
type triangle = (point, point, point)
```

We shall make the assumption that the three points that define the triangle have already been sorted according to the y-axis. Thus, we can assume that the first point is the top point, the third point is the lowest point, and the second point is the middle point (according to the y-axis).

The first function we need to pass to the `expand` function is a function that determines the number of horizontal lines in the triangle:

```
let lines_in_triangle ((p,_,r):triangle) : i32 =
  r.2 - p.2 + 1
```

The second function we need to pass to the `expand` function is somewhat more involved. We first define a function `dxdy`, which computes the inverse slope of a line between two points:

```
let dxdy (a:point) (b:point) : f32 =
  let dx = b.1 - a.1
  let dy = b.2 - a.2
  in if dy == 0 then f32.i32 0
     else f32.i32 dx f32./ f32.i32 dy
```

We can now define the function that, given a triangle and the horizontal line number in the triangle (counted from the top), returns the corresponding line:

```
let get_line_in_triangle ((p,q,r):triangle) (i:i32) =
  let y = p.2 + i
  in if i <= q.2 - p.2 then      -- upper half
       let sl1 = dxdy p q
       let sl2 = dxdy p r
       let x1 = p.1 + i32.f32(f32.round(sl1 * f32.i32 i))
       let x2 = p.1 + i32.f32(f32.round(sl2 * f32.i32 i))
       in ((x1,y),(x2,y))
     else                         -- lower half
       let sl1 = dxdy r p
       let sl2 = dxdy r q
       let dy = (r.2 - p.2) - i
       let x1 = r.1 - i32.f32(f32.round(sl1 * f32.i32 dy))
       let x2 = r.1 - i32.f32(f32.round(sl2 * f32.i32 dy))
       in ((x1,y),(x2,y))
```

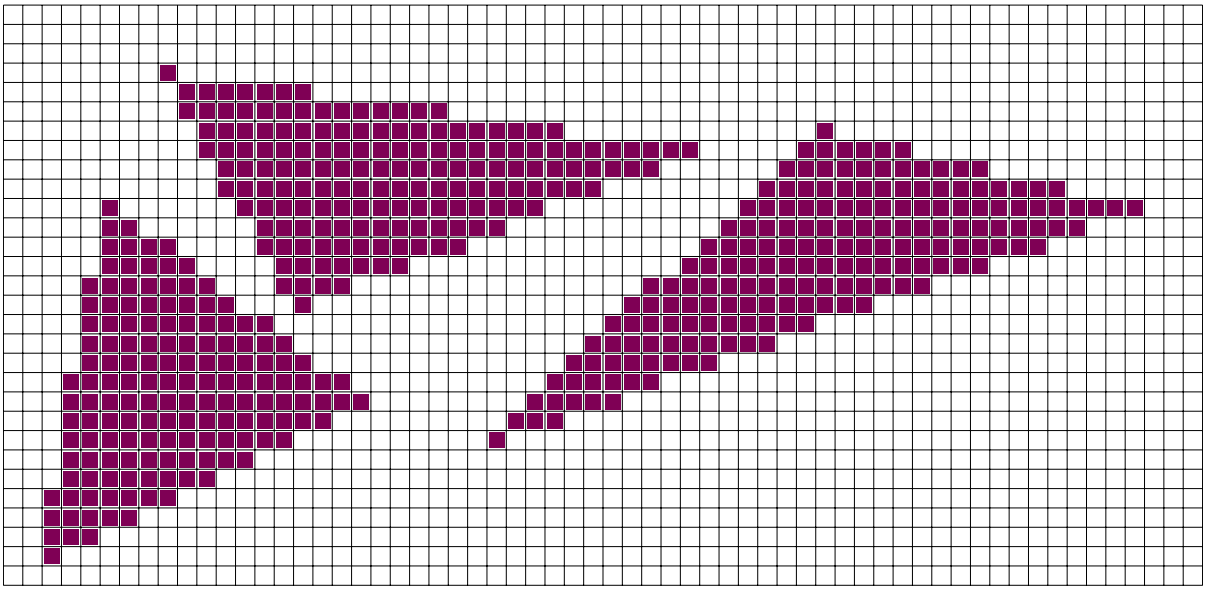
The function distinguishes between whether the line to compute resides in the upper or the lower subtriangle. Finally, we can define a parallel, work-efficient function that converts a number of triangles into lines:

```
let lines_of_triangles (xs:[]triangle) : []line =
  expand lines_in_triangle get_line_in_triangle
         (map normalize xs)
```

To see the code in action, here is a function that draws three triangles on a grid of height 30 and width 62:

```
let draw (height:i32) (width:i32) : [][]i32 =
  let grid : *[][]i32 = replicate height (replicate width 0)
  let triangles = [((5,10),(2,28),(18,20)),
                   ((42,6),(58,10),(25,22)),
                   ((8,3),(15,15),(35,7))]
  let lines = lines_of_triangles triangles
  in drawlines grid lines
```

The function makes use of both the `lines_of_triangles` function that we have defined here and the work efficient `drawlines` function defined previously. Here is a plot of the result:



10.4 Complex Flattening

Unfortunately, the flattening-by-expansion technique does not suit all irregular problems. We shall now investigate how we can flatten a highly irregular algorithm such as quick-sort. The Quick-sort algorithm can be presented very elegantly in a functional language. The function `qsort` that we will define has the following type:

```
val qsort 't [n] : (t -> t -> bool) -> [n]t -> [n]t
```

Given a comparison function (<=) and an array of elements `xs`, `qsort (<=) xs` returns an array with the elements in `xs` sorted according to <=. Consider the following pseudo-code, which, unfortunately, is not immediately Futhark code:

```
let qsort (<=) xs =
  if length xs < 2 then xs
  else let (left,middle,right) = partition (<=) xs[length xs / 2] xs
       in qsort (<=) left ++ middle ++ qsort (<=) right
```

Here the function `partition` returns three arrays with the first array containing elements smaller than the *pivot* element `xs[length xs / 2]`, the second array containing elements equal to the pivot element, and the third array containing elements that are greater than the pivot element. There are multiple problems with this code. First, the code makes use of recursion, which is not supported by Futhark. Second, the kind of recursion used is not tail-recursion, which means that it is not directly obvious how to eliminate the recursion. Third, it is not clear how the code can avoid using an excessive amount of memory instead of making use of inplace-updates for the sorting. Finally, it seems that the code is inherently task-parallel in nature and not particularly data-parallel.

The solution is to solve a slightly more general problem. More precisely, we shall set out to sort a number of segments, simultaneously, where each segment comprises a part of the array. Notice that we are interested in supporting a notion of *partial segmentation*, for which the segments of interest are disjoint but do not necessarily together span the entire array. In particular, the algorithm does not need to sort segments containing previously chosen pivot values. Such segments are already located in the correct positions, which means that they need not be moved around by the segmented quick sort implementation.

We first define a type `sgm` that specifies a segment of an underlying one-dimensional array of values:

```
type sgm = {start:i32,sz:i32}  -- segment descriptor
```

At top-level, the function `qsort` is defined as follows, assuming a function `step` of type `(t -> t -> bool) -> *[n]t -> []sgm -> (*[n]t,[]sgm)`:

```
let qsort [n] 't ((<=): t -> t -> bool) (xs:[n]t) : [n]t =
  if n < 2 then xs
  else (loop (xs,mms) = (copy xs,[{start=0,sz=n}])
        while length mms > 0 do
          step (<=) xs mms).1
```

The `step` function is called initially with the array to be sorted as argument together with a singleton array containing a segment denoting the entire array to be sorted. The `step` function is called iteratively until the returned array of segments is empty. The job of the `step` function is to divide each segment into three new segments based on pivot values found for each segment. After the step function has reordered the values in the segments, the middle segment (containing values equal to a pivot) need not be dealt with again in the further process. A new array of segment descriptors is then defined and after removing empty segment descriptors, the resulting array of non-empty segment descriptors is returned by the `step` function together with the reordered value array.

Before we can define the `step` function, we first define a few helper functions. Using the functions `segmented_iota` and `segmented_replicate`, defined earlier, we can define a function for finding all the indexes represented by an array of segments:

```
let idxs_values (sgms:[]sgm) : []i32 =
  let sgms_szs : []i32 = map (\sgm -> sgm.sz) sgms
  let is1 = segmented_replicate sgms_szs (map (\x -> x.start) sgms)
  let fs = map2 (!=) is1 (rotate (i32.negate 1) is1)
  let is2 = segmented_iota fs
  in map2 (+) is1 is2
```

We also define a function `info` that, given an ordering function and two elements, returns `-1` if the first element is less than the second element, `0` if the elements are identical, and `1` if the first element is greater than the second element:

```
let info 't ((<=): t -> t -> bool) (x:t) (y:t) : i32 =
  if x <= y then if y <= x then 0 else -1
  else 1
```

The following two functions `tripit` and `tripadd` are used for converting the classification of elements into subsegments:

```
let tripit x = if x < 0 then (1,0,0)
               else if x > 0 then (0,0,1) else (0,1,0)

let tripadd (a1:i32,e1:i32,b1:i32) (a2,e2,b2) =
  (a1+a2,e1+e2,b1+b2)
```

We can now define the function `step` that, besides from an ordering function, takes as arguments (1) the array containing values and (2) an array of segments to be sorted. The function returns a pair of a reordered array of values and a new array of segments to be sorted:

```
let step [n] 't ((<=): t -> t -> bool) (xs:*[n]t) (sgms:[]sgm)
  : (*[n]t,[]sgm) =
  -- find a pivot for each segment
  let pivots : []t = map (\sgm -> unsafe xs[sgm.start + sgm.sz/2]) sgms

  -- find index into the segment that a value belongs to
  let idxs : []i32 = replicated_iota (map (\sgm -> sgm.sz) sgms)

  let is = idxs_values sgms

  -- for each value, how does it compare to the pivot associated
  -- with the segment?
  let infos : []i32 =
    map2 (\idx i -> unsafe info (<=) xs[i] pivots[idx]) idxs is
  let orders : [](i32,i32,i32) = map tripit infos

  -- compute segment descriptor
  let flags =
    [true] ++ (map2 (!=) idxs (rotate (i32.negate 1) idxs))[1:]

  -- compute partition sizes for each segment
  let pszs : [](i32,i32,i32) =
    segmented_reduce tripadd (0,0,0) flags orders

  -- compute the new segments
  let sgms' =
    map2 (\(sgm:sgm) (a,e,b) -> [{start=sgm.start,sz=a},
                                 {start=sgm.start+a+e,sz=b}]) sgms pszs
    |> flatten
```

(continues on next page)

(continued from previous page)

```
  |> filter (\sgm -> sgm.sz > 1)

-- compute the new positions of the values in the present segments
let newpos : []i32 =
  let where : [](i32,i32,i32) =
    segmented_scan tripadd (0,0,0) flags orders
  in map3 (\i (a,e,b) info ->
           let (x,y,_) = unsafe pszs[i]
           let s = unsafe sgms[i].start
           in if info < 0 then s+a-1
              else if info > 0 then s+b-1+x+y
              else s+e-1+x) idxs where infos

let vs = map (\i -> unsafe xs[i]) is
let xs' = scatter xs newpos vs
in (xs',sgms')
```

The algorithm has best case work complexity $O(n)$ (when all elements are identical), worst case work complexity $O(n^2)$, and an average case work complexity of $O(n\log n)$. It has best depth complexity $O(1)$, worst depth complexity $O(n)$ and average depth complexity $O(\log n)$.

Chapter 11

Conclusion

In this book, we have aimed at providing a practical guide to writing data-parallel programs in Futhark. Futhark is quite an extensive language even though its semantics is pure. It does however have limitations. In particular, Futhark does not currently support recursion and it has no built-in support for algebraic datatypes. Support for some of these concepts are currently being investigated.

On the performance side, there are, of course, always room for improvements. In particular, a number of low-level optimisations, such as register tiling, could turn out helpful for certain kinds of applications. However, even with the current performance level, Futhark may turn out fruitful for serious prototyping and quick time-to-market development.

The Futhark web site at http://futhark-lang.org contains a list of research papers, which will serve as a suggestion for further reading.

Bibliography

[ABB+16] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E Oancea. Finpar: a parallel financial benchmark. In *ACM TACO*. 2016.

[Ann18] Danil Annenkov. *Adventures in Formalisation: Financial Contracts, Modules, and Two-Level Type Theory*. PhD thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, April 2018.

[Bir87] R. S. Bird. An Introduction to the Theory of Lists. In *NATO Inst. on Logic of Progr. and Calculi of Discrete Design*, 5–42. 1987.

[Ble90] Guy E Blelloch. *Vector models for data-parallel computing*. Volume 75. MIT press Cambridge, 1990.

[BHS+94] Guy E Blelloch, Jonathan C Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[Ell03] Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003. URL: http://conal.net/papers/functional-images/.

[Els98] Martin Elsman. Polymorphic equality—no tags required. In *Second International Workshop on Types in Compilation (TIC'98)*. March 1998.

[Els99] Martin Elsman. Static interpretation of modules. In *Procedings of Fourth International Conference on Functional Programming (ICFP'99)*, 208–219. ACM Press, September 1999.

[Els05] Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*. September 2005.

[EHAO18] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in Futhark: functional GPU programming in the large. *Proc. ACM Program. Lang.*, 2(ICFP):97:1–97:30, July 2018. URL: http://doi.acm.org/10.1145/3236792, doi:10.1145/3236792.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions of Computers*, 21(9):948–960, September 1972.

[GHK+11] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, September 2011. Second Edition.

[Hen17] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Universitetsparken 5, DK-2100 Copenhagen, 11 2017.

[HDU+16] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. APL on GPUs—a TAIL from the past, scribbled in Futhark. In *Proceedings of the 5th ACM SIGPLAN workshop on Functional High-Performance Computing (FHPC'16)*. ACM, September 2016.

[HEO14] Troels Henriksen, Martin Elsman, and Cosmin E Oancea. Size slicing: a hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional High-Performance Computing (FHPC'14)*, 31–42. ACM, 2014.

[HEO18] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. Modular acceleration: tricky cases of functional high-performance computing. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2018. New York, NY, USA, 2018. ACM.

[HLO16] Troels Henriksen, Ken Friis Larsen, and Cosmin E Oancea. Design and GPGPU performance of Futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 17–24. ACM, 2016.

[HO14] Troels Henriksen and Cosmin E Oancea. Bounds checking: an instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*, 88. ACM, 2014.

[HO13] Troels Henriksen and Cosmin Eugen Oancea. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional High-Performance Computing (FHPC'13))*, 47–58. ACM, 2013.

[HSE+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 556–571. New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3062341.3062354, doi:10.1145/3062341.3062354.

[HTEO19] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin E. Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '19. ACM, February 2019.

[Hov18] Anders Kiel Hovgaard. Higher-order functions for a high-performance programming language for GPUs. Master's thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, Universitetsparken 5, DK-2100 Copenhagen, May 2018.

[HHE18] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming (TFP'18)*. September 2018.

[Ken04] Andrew J. Kennedy. Functional pearl: pickler combinators. *Jounal of Functional Programming*, 14(6):727–739, November 2004.

[LH17] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, 42–52. New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3122948.3122952, doi:10.1145/3122948.3122952.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[Moo75] Gordon E. Moore. Progress in Digital Integrated Electronics. In *Technical Digest 1975*, 11–13. IEEE, 1975. International Electron Devices Meeting.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. URL: http://doi.acm.org/10.1145/1365490.1365500, doi:10.1145/1365490.1365500.

[Org16] Course Organizers. *Algorithm Design: Parallel and Sequential*. Carnegie Mellon University, September 2016. Course Book Draft Edition. Course Taught Fall 2016 by Umut Acar and Robert Harper.

[PJ93] John Peterson and Mark Jones. Implementing type classes. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, 227–236. New York, NY, USA, 1993. ACM. URL: http://doi.acm.org/10.1145/155090.155112, doi:10.1145/155090.155112.

[Sch80] Jacob T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, October 1980. URL: http://doi.acm.org/10.1145/357114.357116, doi:10.1145/357114.357116.

[vN45] John von Neumann. First draft of a report on the EDVAC. Technical Report, Moore School of Electrical Engineering, University of Pennsylvania, June 1945.