Futhark Documentation

Release 0.1

HIPERFIT

May 25, 2017

Table of Contents

Welcome to the documentation for the Futhark compiler and language. For a basic introduction, please see our website. To get started, read the page on *Installation*. Once the compiler has been installed, you might want to take a look at *Language Overview* or *C Porting Guide*. Additional documentation can also be found in our publications.

CHAPTER 1

Installation

There are two ways to install the Futhark compiler: using a precompiled tarball or compiling from source. Both methods are discussed below. If you are using Windows, make sure to read *Installing Futhark on Windows*.

Compiling from source

We use the the Haskell Tool Stack to handle dependencies and compilation of the Futhark compiler, so you will need to install the `stack` tool. Fortunately, the `stack` developers provide ample documentation about installing Stack on a multitude of operating systems. If you're lucky, it may even be in your local package repository.

We do not presently issue source releases of Futhark, so the only way to compile from source is to perform a checkout of our Git repository:

```
$ git clone https://github.com/HIPERFIT/futhark.git
```

This will create a directory `futhark`, which you must enter:

```
$ cd futhark
```

To get all the prerequisites for building the Futhark compiler (including, if necessary, the appropriate version of the Haskell compiler), run:

```
$ stack setup
```

Note that this will not install anything system-wide and will have no effect outside the Futhark build directory. Now you can run the following command to build the Futhark compiler, including all dependencies:

```
$ stack build
```

The Futhark compiler and its tools will now be built. You can copy them to your `$HOME/.local/bin` directory by running:

```
$ stack install
```

Note that this does not install the Futhark manual pages.

Installing from a precompiled snapshot

We do not yet have any proper releases as such, but every day a program automatically clones the Git repository, builds the compiler, and packages a simple tarball containing the resulting binaries, built manpages, and a simple `Makefile` for installing. The implication is that these tarballs are not vetted in any way, nor more stable than Git HEAD at any particular moment in time. They are provided merely for users who are unable or unwilling to compile Futhark themselves.

At the moment, we build such snapshots only for a single operating system:

Linux (x86_64) futhark-nightly-linux-x86_64.tar.xz

In time, we hope to make snapshots available for more platforms, but we are limited by system availability. We also intend to make proper releases once the language matures.

Installing Futhark on Windows

While the Futhark compiler itself is easily installed on Windows via `stack` (see above), it takes a little more work to make the OpenCL and PyOpenCL backends functional. This guide was last updated on the 5th of May 2016, and is for computers using 64-bit Windows along with CUDA 7.5 and Python 2.7 (Anaconda preferred).

Also Git for Windows is required for its Linux command line tools. If you have not marked the option to add them to path, there are instructions below how to do so. The GUI alternative to `git`, Github Desktop is optional and does not come with the required tools.

Setting up Futhark and OpenCL

1. Clone the Futhark repository to your hard drive.
2. Install Stack using the 64-bit installer. Compile the Futhark compiler as described in *Installation*.
3. For editing environment variables it is strongly recommended that you install the Rapid Environment Editor
4. For a Futhark compatible C/C++ compiler, that you will also need to install pyOpenCL later, install MingWpy. Do this using the `pip install -i https://pypi.anaconda.org/carlkl/simple mingwpy` command.
5. Assuming you have the latest Anaconda distribution as your primary one, it will get installed to a place such as `C:\Users\UserName\Anaconda2\share\mingwpy`. The pip installation will not add its bin or include directories to path.

   To do so, open the Rapid Environment Editor and add `C:\Users\UserName\Anaconda2\share\mingwpy\bin` to the system-wide `PATH` variable.

   If you have other MingW or GCC distributions, make sure MingWpy takes priority by moving its entry above the other distributions. You can also change which Python distribution is the default one using the same trick should you need so.

   If have done so correctly, typing `where gcc` in the command prompt should list the aforementioned MingWpy installation at the top or show only it.

To finish the installation, add the `C:\Users\UserName\Anaconda2\share\mingwpy\include` to the `CPATH` environment variable (note: *not* `PATH`). Create the variable if necessary.

6. The header files and the .dll for OpenCL that comes with the CUDA 7.5 distribution also need to be installed into MingWpy. Go to `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\include` and copy the `CL` directory into the MingWpy `include` directory.

   Next, go to `C:\Program Files\NVIDIA Corporation\OpenCL` and copy the `OpenCL64.dll` file into the MingWpy `lib` directory (it is next to `include`).

   The CUDA distribution also comes with the static `OpenCL.lib`, but trying to use that one instead of the `OpenCL64.dll` will cause programs compiled with `futhark-opencl` to crash, so ignore it completely.

Now you should be able to compile `futhark-opencl` and run Futhark programs on the GPU.

Congratulations!

Setting up PyOpenCL

The following instructions are for how to setup the `futhark-pyopencl` backend.

First install Mako using `pip install mako`.

Also install PyPNG using `pip install pypng` (not stricly necessary, but some examples make use of it).

7. Clone the PyOpenCL repository to your hard drive. Do this instead of downloading the zip, as the zip will not contain some of the other repositories it links to and you will end up with missing header files.

8. If you have ignored the instructions and gotten Python 3.x instead 2.7, you will have to do some extra work.

   Edit `.\pyopencl\compyte\ndarray\gen_elemwise.py` and `.\pyopencl\compyte\ndarray\test_gpu_ndarray.py` and convert most Python 2.x style print statements to Python 3 syntax. Basically wrap print arguments in brackets "(..)" and ignore any lines containing StringIO >> operator.

   Otherwise just go to the next point.

9. Go into the repo directory and from the command line execute `python configure.py`.

   Edit `siteconf.py` to following:

```
CL_TRACE = false
CL_ENABLE_GL = false
CL_INC_DIR = ['c:\\Program Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v7.
↪5\\include']
CL_LIB_DIR = ['C:\\Program Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v7.
↪5\\lib\\x64']
CL_LIBNAME = ['OpenCL']
CXXFLAGS = ['-std=c++0x']
LDFLAGS = []
```

   Run the following commands:

```
> python setup.py build_ext --compiler=mingw32
> python setup.py install
```

If everything went in order, pyOpenCL should be installed on your machine now.

10. Lastly, Pygame needs to be installed. Again, not stricly necessary, but some examples make use of it. To do so on Windows, download `pygame-1.9.2a0-cp27-none-win_amd64.whl` from here. `cp27` means Python 2.7 and `win_amd64` means 64-bit Windows.

Go to the directory you have downloaded the file and execute `pip install pygame-1.9.2a0-cp27-none-win_amd64.whl` from the command line.

Now you should be able to run the Mandelbrot Explorer and and Game of Life examples.

11. To run the makefiles, first setup `make` by going to the `bin` directory of MingWpy and making a copy of `mingw32-make.exe`. Then simply rename `mingw32-make - Copy.exe` or similar to `make.exe`. Now you will be able to run the makefiles.

    Also, if you have not selected to add the optional Linux command line tools to `PATH` during the `Git for Windows` installation, add the `C:\Program Files\Git\usr\bin` directory to `PATH` manually now.

12. This guide has been written off memory, so if you are having difficulties - ask on the issues page. There might be errors in it.

CHAPTER 2

Basic Usage

Futhark contains several code generation backends. Each is provided as a full standalone compiler binary. For example, `futhark-c` compiles a Futhark program by translating it to sequential C code, while `futhark-pyopencl` generates Python and the PyOpenCL library. The different compilers all contain the same frontend and optimisation pipeline - only the code generator is different. They all provide roughly the same command line interface, but there may be minor differences and quirks due to characteristics of the specific backends.

Compiling to Executable

A Futhark program is stored in a file with the extension `.fut`. It can be compiled to an executable program as follows:

```
$ futhark-c prog.fut
```

This makes use of the `futhark-c` compiler, but any other will work as well. The result is an executable binary called `prog`. If we had used `futhark-py` instead of `futhark-c`, the `prog` file would instead have contained Python code, along with a shebang for easy execution. In general, when compiling file `foo.fut`, the result will be written to a file `foo` (i.e. the extension will be stripped off). This can be overridden using the `-o` option.

When a Futhark program is run, execution starts at the function named `main`. If the program has no `main` function, the compiler will fail with an error. If `main` takes any parameters, these will be read from standard input in Futhark syntax. **Note:** Tuple value syntax is not supported. Instead, pass the values comprising the tuple as if they were immediate parameters to the `main` function. We recommended using only primitive types and arrays of primitive types as parameter (and return) types in the `main` function.

Instead of compiling, there is also an interpreter, `futharki`. Be aware that it is very slow, and does not produce better error messages than the compiler. **Note:** If you run `futharki` without any options, you will see something that looks deceptively like a REPL, but it is not yet finished, and useless in its present state.

Compiling to Library

While compiling a Futhark program to an executable is useful for testing, it is not the intended use case. Instead, a Futhark program should be compiled into a reusable library in some target language, enabling integration into a larger program. At the moment, this work has only been done for the `futhark-py` and `futhark-pyopencl` backends, and only the latter of these generates sufficiently performant code for it to be worthwhile.

We can use `futhark-pyopencl` to translate the program `futlib.fut` into a Python module `futlib.py` with the following command:

```
futhark-pyopencl --library futlib.fut
```

This will create a file `futlib.py`, which contains Python code that defines a class named `futlib`. This class defines one method for each entry point function (see *Entry Points*) in the Futhark program. After the class has been instantiated, these methods can be invoked to run the corresponding Futhark function. The reason for why the module does does not expose functions directly, is that some complicated initialisation and stateful bookkeeping may be necessary for sophisticated backends, such as PyOpenCL.

CHAPTER 3

Language Overview

The Futhark programming language is a purely functional, call-by-value, mostly first-order language that permits bulk operations on arrays using *second-order array combinators* (SOACs).

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. It is also the plan to eventually design a rigorous cost model for Futhark, although this work has not yet been completed.

Lexical Syntax

The syntax of Futhark is derived from Haskell and Standard ML, although somewhat less flexible. Futhark is not whitespace-sensitive, and indentation is only used for readability. An identifier starts with a lowercase letter, followed by any number of letters, digits, apostrophes and underscores. An identifier may also start with an underscore, which is used to indicate that the name will not be used. A structure identifier starts with an uppercase letter. Numeric, string and character literals use the same notation as Haskell (which is very similar to C), including all escape characters. Comments are indicated with `--` and span to end of line. Block comments are not presently supported.

Simple Futhark

An Futhark program consists of a sequence of *function definitions*, of the following form:

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

A function must declare both its return type and the types of all its parameters. All functions (except for inline anonymous functions; see below) are defined globally. Futhark does not use type inference. As a concrete example, here is the a parallel definition of the factorial function in Futhark:

```
let fact(n: i32): i32 =
  reduce (*) 1 (map (1+) (iota n))
```

Indentation has no syntactical significance in Futhark, but recommended for readability.

The syntax for tuple types is a comma-separated list of types or values enclosed in parentheses, so `(i32, real)` is a pair of an integer and a floating-point number. Single-element and empty tuples are not permitted. Array types are written as the element type preceded by brackets, meaning that `[]i32` is a one-dimensional array of integers, and `[][][](i32, real)` is a three-dimensional array of tuples of integers and floats. An array value is written as a sequence of elements enclosed by brackets:

```
[1, 2, 3]       -- Array of type []i32.
[[1], [2], [3]] -- Array of type [][]i32.
```

All arrays must be *regular* (often termed *full*). This means that, for example, all rows of a two-dimensional array must have the same number of elements:

```
[[1, 2], [3]]      -- Compile-time error.
[iota(1), iota(2)] -- A run-time error if reached.
```

The restriction to regular arrays simplifies compilation.

Arrays are indexed using the common row-major notation, e.g., `a[i1, i2, i3...]`. An indexing is said to be *full* if the number of given indices is equal to the dimensionality of the array.

A `let`-expression can be used to refer to the result of a subexpression:

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

Recall that Futhark is eagerly evaluated, so the right-hand side of the `let` is evaluated exactly once, at the time it is first encountered. The `in` keyword is optional when it precedes another `let`. This means that 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.

Two-way `if-then-else` is the only branching construct in Futhark. Pattern matching is supported in a limited way for taking apart tuples, for example:

```
let sumpair((x, y): (i32, i32)): i32 = x + y
```

We can also add the type ascription on the tuple components:

```
let sumpair(x: i32, y: i32): i32 = x + y
```

Apart from pattern-matching, the components of a tuple can also be accessed by using the tuple projection syntax: `#i`, where `i` indicates the component to extract (0-indexed):

```
let sumpair(p: (i32, i32)): i32 = #0 p + #1 p
```

In most cases, pattern matching is better style.

Function calls are written as the function name with the arguments juxtaposed. All function calls must be fully saturated - currying is only permitted in *SOACs*.

Sequential Loops

Futhark has a built-in syntax for expressing certain tail-recursive functions. Consider the following tail-recursive formulation of a function for computing the Fibonacci numbers:

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

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

This is not valid Futhark, as Futhark does not presently allow recursive functions. Instead, we can write this using the `loop` construct:

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

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

```
loop (pat = initial) = for i < bound do loopbody
in body
```

Has the following the 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. Evaluate *body* with *pat* bound to its final value.

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`, this loop:

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

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

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

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

The purpose of `loop` 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.

Apart from the `i < n` form, which loops from zero, Futhark also supports the `v <= i < n` form which starts at `v`. We can also invert the order of iteration by writing `n > i` or `n > i >= v`, which loops down from the upper bound to the lower. Due to parser limitations, most non-atomic expressions will have to be parenthesised when used as the left-hand bound.

Apart from `for`-loops, Futhark also supports `while` loops. These 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 x
```

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 is easier to understand with an example. The loop:

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

This can sometimes make imperative code look more natural.

Shape Declarations

Optionally, the programmer may put *shape declarations* in the return type and parameter types of a function declaration. These can be used to express invariants about the shapes of arrays that are accepted or produced by the function, e.g:

```
let f (a: [#n]i32): [n]i32 =
  map (+1) a
```

The above declaration specifies a function that accepts an array containing `n` elements and returns an array likewise containing `n` elements. When prefixed with a `#` character, a name is *freshly bound*, whilst an unadorned name must be in scope. In the example above, we do not use a `#` in the return type, because we wish to refer to the `n` bound by the parameters. If we refer to the same freshly bound variable in multiple parameters (see below), each occurence must be prefixed with `#`. A shape declaration can also be an integer constant (with no suffix).

The same name can be used in several dimensions, or even in several parameters. This can be used to give a natural type to a function for computing dot products:

```
let dotProduct(a: [#n]i32, b: [#n]i32): i32 =
  reduce (+) 0 (map (*) a b)
```

Or matrix multiplication:

```
let matMult(x: [#n][#m]i32, y: [#m][#n]i32): [n][n]i32 =
  ...
```

The dimension names bound in a parameter shape declaration can be used as ordinary variables inside the scope of the parameter.

Shape declarations serve two main purposes:

1. They document the shape assumptions of the function in an easily understandable form.
2. More importantly, they help the compiler understand the invariants of the program, which it may otherwise have trouble figuring out.

Note that adding shape declarations is never unsafe - the compiler still inserts dynamic checks, so if an incorrect declaration is made, the result will merely be an abrubt but controlled termination as it collides with reality. Shape declarations matter most when used for the input parameters of the `main` function and for the return type of functions used to `map`.

In-Place Updates

In an array programming language, we tend to use bulk operations for most array manipulation. However, sometimes it is useful to directly replace some element. In a pure language, we cannot permit free mutation, but we can permit the creation of a duplicate array, where some elements have been changed. General modification of array elements is done using the `let-with` construct. In its most general form, it looks as follows:

```
let dest = src with [indexes] <- (value)
in body
```

This evaluates `body` with `dest` bound to the value of `src`, except that the element(s) at the position given by `indexes` take on the new value `value`. Due to parser limitations, the parenthesis around `value` are not optional. The given indexes need not be complete, but in that case, `value` must be an array of the proper size. As an example, here's how we could replace the third row of an `n * 3` array:

```
let b = a with [2] <- ([1,2,3]) in b
```

As a convenience, whenever `dest` and `src` are the same, we can write:

```
let dest[indexes] = value in body
```

as a shortcut. Note that this has no special semantic meaning, but is simply a case of normal name shadowing.

For example, this loop implements the "imperative" version of matrix multiplication of an `m * o` with an `o * n` matrix:

```
let matmult(a: [#m][#o]f32,  b: [#o][#n]f32): [m][n]f32 =
  let res = replicate(m, replicate(n,0f32)) in
  loop (res) = for i < m do
      loop (res) = for j < n do
          loop (partsum = 0f32) = for k < o do
            partsum + a[i,k] * b[k,j]
          let res[i,j] = partsum
          in res
```

```
    in res
in res
```

With the naive implementation based on copying the source array, executing the `let-with` expression would require memory proportional to the entire source array, rather than proportional to the slice we are changing. This is not ideal. Therefore, the `let-with` construct has some unusual restrictions to permit in-place modification of the `src` array, as described in *Uniqueness Types*. Simply put, we track that `src` is never used again. The consequence is that we can guarantee that the execution of a `let-with` expression does not involve any copying of the source array in order to create the newly bound array, and therefore the time required for the update is proportional to the section of the array we are updating, not the entire array. We can think of this as similar to array modification in an imperative language.

SOACs

The language presented in the previous section is in some sense "sufficient", in that it is Turing-complete, and can express imperative-style loops in a natural way with `do` and `while`-loops. However, Futhark is not intended to be used in this way - bulk operations on arrays should be expressed via one of the *second-order array combinators* (SOACs) shown below, as this maximises the amount of parallelism that the compiler is able to take advantage of.

```
e  ::=   ``map'' lambda e
         ``filter'' lambda e
         ``partition'' ``('' lambda '','' ... lambda '')'' e
         ``reduce'' lambda e e
         ``scan'' lambda e e
```

A lambda can be an anonymous function, the name of a function (with optional curried arguments), or an operator (possibly with one operand curried):

```
lambda  ::=   ``('' \ param... :  rettype ``->'' e '')''
              fname
              ``('' fname e ... e '')''
              ``('' op e '')''
              ``('' e op '')''
              ``('' op '')''
```

Parameter- and return type ascriptions are optional in anonymous functions. The semantics of the SOACs is identical to the similarly-named higher-order functions found in many functional languages. For specifics, see *Language Reference*.

The `scan` SOAC performs an inclusive prefix scan, and returns an array of the same outer size as the original array. The functions given to `reduce` and `scan` must be binary associative operators, and the value given as the initial value of the accumulator must be the neutral element for the function. These properties are not checked by the Futhark compiler, and are the responsibility of the programmer.

Uniqueness Types

While Futhark is uncompromisingly a 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 = iota(n) in
  -- Fill array with Fibonacci numbers.
  loop (arr) = for i < n-2 do
                 let arr[i+2] = arr[i] + arr[i+1]
                 in arr
  in arr
```

If the array `arr` is copied for each iteration of the loop, we are going to put enormous pressure on memory, and 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)* becomes *(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, we will want to update the array *in-place*, that is, with a static guarantee that the operation will not require any additional memory allocation, such as copying the entire array. With an in-place modification, a `let-with` can modify the array in time proportional to the slice 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 performed a full copy. In order to perform the update without violating referential transparency, we need to 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 Clean. Alongside a (relatively) simple aliasing analysis in the type checker, this is sufficient to determine at compile time whether an in-place modification is safe, and signal a compile time error if `let-with` is used in 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 =
  let a[i] = a[i] + x in
  a
```

The function call `modify(a,i,x)` returns `a`, but where the element at index `i` has been increased by `x`. Note the asterisks in the parameter declaration `*[]i32 a`. This means that the function `modify` has been given "ownership" of the array `a`, meaning that the caller of `modify` will never reference array `a` after the call. As a consequence, `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) in
..
```

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`.

In general, when a value is passed as a unique-typed argument in a function call, we consider that value to be *consumed*, and neither it nor any of its aliases can be used again. Otherwise, we would break the contract that gives the function liberty to manipulate the argument however it wants. Note 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, i.e. overlap in memory. As the most trivial case, after evaluating the binding `let 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] in
... -- b is aliased to a (assuming a is not one-dimensional)
```

In *Sharing Analysis* below, we will cover sharing and sharing analysis in greater detail.

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

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

Note that the argument, `a`, is non-unique, and hence we cannot modify it. 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 ''let-with'', 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 passed in the place of a unique parameter in a function call, or used as the source in a `let-with` expression, neither that value, nor any value that aliases it, may be used on any execution path following the function call. An example violation:

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

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. An example violation:

```
let broken(a: [][]i32, i: i32): *[]i32 =
  a[i] -- 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 in this chapter is employed as part of a static analysis. *All* violations of the uniqueness rules will be discovered at compile time during type-checking, thus leaving the code generator and runtime system at liberty to exploit them for low-level optimisation.

Sharing Analysis

Whenever the memory regions for two values overlap, we say that they are *aliased*, or that *sharing* is present. As an example, if you have a two-dimensional array `a` and extract its first row as the one-dimensional array `b`, we say that `a` and `b` are aliased. While the Futhark compiler may do a deep copy if it wishes, it is not required, and this operation thus holds the potential for sharing memory. Sharing analysis is necessarily conservative, and merely imposes an upper bound on the amount of sharing happening at runtime. The sharing analysis in Futhark has been carefully designed to make the bound as tight as possible, but still easily computable.

In Futhark, the only values that can have any sharing are arrays - everything else is considered "primitive". Tuples are special, in that they are not considered to have any identity beyond their elements. Therefore, when we store sharing information for a tuple-typed expression, we do it for each of its element types, rather than the tuple value as a whole.

Most operations produce arrays without any aliases. You can think of these as producing fresh arrays. The exceptions are `split`, `reshape`, `transpose`, `rearrange`, `zip` and `unzip`, as well as function calls and `if` expressions (depending on types). You can use `copy` from `/futlib/array` to "break" sharing by forcing the argument to be manifested freshly in memory.

CHAPTER 4

Language Reference

This reference seeks to describe every construct in the Futhark language. It is not presented in a tutorial fashion, but rather intended for quick lookup and documentation of subtleties. For this reason, it is not written in a bottom-up manner, and some concepts may be used before they are fully defined. It is a good idea to have a basic grasp of Futhark (or some other functional programming language) before reading this reference. An ambiguous grammar is given for the full language. The text describes how ambiguities are resolved in practice (things like operator precedence).

Identifiers and Keywords

```
id         ::=   letter (letter | ``_'' | ``''')* | ``_'' id
quals      ::=   (id ''.'')+
qualid     ::=   id | quals id
binop      ::=   symbol+
qualbinop  ::=   binop | quals binop
fieldid    ::=   decimal | id
symbol     ::=   ``+'' | ``-'' | ``*'' | ``/'' | ``%'' | ``='' | ''!'' | ``>'' | ``<'' | ``
```

Many things in Futhark are named. When we are defining something, we give it an unqualified name (*id*). When referencing something inside a module, we use a qualified name (*qualid*). The fields of a record are named with *fieldid*'s. Note that a '*fieldid* can be decimal numbers.

Primitive Types and Values

```
literal  ::=   intnumber | floatnumber | ``true'' | ``false''
```

Boolean literals are written `true` and `false`. 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`, as well as `bool`. An `f32` is always a single-precision float and a `f64` is a double-precision float.

```
int_type    ::=   ``i8'' | ``i16'' | ``i32'' | ``i64'' | ``u8'' | ``u16'' | ``u32'' | ``u64
float_type  ::=   ``f8'' | ``f16'' | ``f32'' | ``f64''
```

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, integer literals are of type `i32`, and decimal literals are of type `f64`. Hexadecimal literals are supported by prefixing with `0x`, and binary literals by prefixing with `0b`.

```
intnumber    ::=   (decimal | hexadecimal | binary) [int_type]
decimal      ::=   decdigit (decdigit |''_'')*
hexadecimal  ::=   0 (``x'' | ``X'') hexdigit (hexdigit |''_'')*
binary       ::=   0 (``b'' | ``B'') bindigit (bindigit | ``_'')*
```

```
floatnumber    ::=   (pointfloat | exponentfloat) [float_type]
pointfloat     ::=   [intpart] fraction | intpart ''.''
exponentfloat  ::=   (intpart | pointfloat) exponent
intpart        ::=   decdigit (decdigit |''_'')*
fraction       ::=   ''.'' decdigit (decdigit |''_'')*
exponent       ::=   (``e'' | ``E'') [''+'' | ``-``] decdigit+
```

```
decdigit  ::=   ``0''...''9''
hexdigit  ::=   decdigit | ``a''...''f'' | ``A''...''F''
bindigit  ::=   ``0'' | ``1''
```

Numeric values can be converted between different types by using the desired type name as a function. E.g., `i32(1.0f32)` would convert the floating-point number `1.0` to a 32-bit signed integer. Conversion from floating-point to integers is done by truncation.

These can also be converted to numbers (1 for true, 0 for false) by using the desired numeric type as a function.

Compound Types and Values

All primitive 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 `()`.

```
type         ::=   qualid | array_type | tuple_type | record_type | type type_arg
array_type   ::=   ``['' [dim] ``]'' type
tuple_type   ::=   ``('' '')'' | ``('' type (``['' '','' type ``]'')* '')''
record_type  ::=   ``{'' ``}'' | ``{'' fieldid '':''  type ('','' fieldid '':''  type)* ``
type_arg     ::=   ``['' [dim] ``]'' | type
dim          ::=   qualid | decimal | ``#'' id
```

An array value is written as a nonempty 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

typically 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 as `empty(t)`, where `t` is the element type.

Multi-dimensional arrays are supported in Futhark, but they must be *regular*, meaning that all inner arrays must 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 come up with integers `m` and `n` such that `[m][n]i32` describes the array. The restriction to regular arrays is rooted in low-level concerns about efficient compilation. However, 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.

Records are mappings from field names to values, with the field names known statically. A tuple behaves in all respects like a record with numeric field names, and vice versa. It is an error for a record type to name the same field twice.

A parametric type abbreviation can be applied by juxtaposing its name and its arguments. The application must provide as many arguments as the type abbreviation has parameters - partial application is presently not allowed. See *Type Abbreviations* for further details.

String literals are supported, but only as syntactic sugar for arrays of `i32` values. There is no `char` type in Futhark.

```
stringlit   ::=   `''' stringchar `'''
stringchar  ::=   <any source character except ``\'' or newline or quotes>
```

Expressions

Expressions are the basic construct of any Futhark program. An expression has a statically determined *type*, and produces a *value* at runtime. Futhark is an eager/strict language ("call by value").

The basic elements of expressions are called *atoms*, for example literals and variables, but also more complicated forms.

```
atom      ::=   literal
              | qualid
              | stringlit
              | ``empty'' ``('' type '')''
              | ``('' '')''
              | ``('' exp '')''
              | ``('' exp (','' exp)* '')''
              | ``{'' ``}''
              | ``{'' field (','' field)* ``}''
              | qualid ``['' index (','' index)* ``]''
              | ``('' exp '')'' ``['' index (','' index)* ``]''
              | ``['' exp (','' exp)* ``]''
              | ``#'' fieldid exp
exp       ::=   atom
              | exp qualbinop exp
              | exp exp
              | exp '':''  type
              | ``if'' exp ``then'' exp ``else'' exp
              | ``let'' type_param* pat ``='' exp ``in'' exp
              | ``let'' id ``['' index (','' index)* ``]'' ``='' exp ``in'' exp
              | ``let'' id type_param* pat+ ['':''  type] ``='' exp ``in'' exp
              | ``loop'' ``('' type_param* pat [(``='' exp)] '')'' ``='' loopform ``do''
```

```
                | ``reshape'' exp exp
                | ``rearrange'' ``('' nat_int+ '')'' exp
                | ``rotate'' [''@'' nat_int] exp exp
                | ``split'' [''@'' nat_int] exp exp
                | ``concat'' [''@'' nat_int] exp+
                | ``zip'' [''@'' nat_int] exp+
                | ``unzip'' exp
                | ``unsafe'' exp
                | exp ``with'' ``['' index (','' index)* ``]'' ``<-'' exp
                | ``map'' fun exp+
                | ``reduce'' fun exp exp
                | ``reduce_comm'' fun exp exp
                | ``reduce'' fun exp exp
                | ``scan'' fun exp exp
                | ``filter'' fun exp
                | ``partition'' ``('' fun+ '')'' exp
                | ``stream_map'' fun exp
                | ``stream_map_per'' fun exp
                | ``stream_red'' fun exp exp
                | ``stream_map_per'' fun exp exp
                | ``stream_seq'' fun exp exp
field     ::=   fieldid ``='' exp
                | exp
pat       ::=   id
                | ``_''
                | ``('' '')''
                | ``('' pat '')''
                | ``('' pat (','' pat)+ '')''
                | ``{'' ``}''
                | ``{'' fieldid ``='' pat [','' fieldid ``='' pat] ``}''
                | pat '':''  type
loopform  ::=   ``for'' id ``<'' exp
                | ``for'' atom ``<='' id ``<'' exp
                | ``for'' atom ``>'' id ``>='' exp
                | ``for'' atom ``>'' id
                | ``while'' exp
```

Some of the built-in expression forms have parallel semantics, but it is not guaranteed that the the parallel constructs in Futhark are evaluated in parallel, especially if they are nested in complicated ways. Their purpose is to give the compiler as much freedom and information is possible, in order to enable it to maximise the efficiency of the generated code.

Resolving Ambiguities

The above grammar contains some ambiguities, which in the concrete implementation is resolved via a combination of lexer and grammar transformations. For ease of understanding, they are presented here in natural text.

- A type ascription (*exp* `:` *type*) cannot appear as an array index, as it collides with the syntax for slicing.
- In `f [x]`, there is am ambiguity between indexing the array `f` at position `x`, or calling the function `f` with the singleton array `x`. We resolve this the following way:
  - If there is a space between `f` and the opening bracket, it is treated as a function application.
  - Otherwise, it is an array index operation.

- The following table describes the precedence and associativity of infix operators. All operators in the same row have the same precedence. The rows are listed in increasing order of precedence. Note that not all operators listed here are used in expressions; nevertheless, they are still used for resolving ambiguities.

Associativity	Operators
left	`,`
left	`:`
left	`||`
left	`&&`
left	`<= >= > < == !=`
left	`& ^ |`
left	`<< >> >>>`
left	`+ -`
left	`* / % // %%`
right	`->`

Semantics

literal

Evaluates to itself.

qualid

A variable name; evaluates to its value in the current environment.

stringlit

Evaluates to an array of type `[]i32` that contains the string characters as integers.

`empty(t)`

Create an empty array whose row type is `t`. For example, `empty(i32)` creates a value of type `[]i32`. The row type can contain shape declarations, e.g., `empty([2]i32)`. Any dimension without an annotation will be of size 0, as will the outermost dimension.

`()`

Evaluates to an empty tuple.

`( e )`

Evaluates to the result of `e`.

`(e1, e2, ..., eN)`

Evaluates to a tuple containing `N` values. Equivalent to `(1=e1, 2=e2, ..., N=eN)`.

`{f1, f2, ..., fN}`

A record expression consists of a comma-separated sequence of *field expressions*. A record expression is evaluated by creating an empty record, then processing the field expressions from left to right. Each field expression adds fields to the record being constructed. A field expression can take one of two forms:

`f = e`: adds a field with the name `f` and the value resulting from evaluating `e`.

`e`: the expression `e` must evaluate to a record, whose fields are added to the record being constructed.

If a field expression attempts to add a field that already exists in the record being constructed, the new value for the field supercedes the old one.

`a[i]`

Return the element at the given position in the array. The index may be a comma-separated list of indexes instead of just a single index. If the number of indices given is less than the rank of the array, an array is returned.

The array `a` must be a variable name or a parenthesized expression. Futhermore, there *may not* be a space between `a` and the opening bracket. This disambiguates the array indexing `a[i]`, from `a [i]`, which is a function call with a literal array.

`a[i:j:s]`

Return a slice of the array `a` from index `i` to `j`, the latter inclusive and the latter exclusive, taking every `s`-th element. The `s` parameter may not be zero. If `s` is negative, it means to start at `i` and descend by steps of size `s` to `j` (not inclusive).

It is generally a bad idea for `s` to be non-constant. Slicing of multiple dimensions can be done by separating with commas, and may be intermixed freely with indexing.

If `s` is elided it defaults to `1`. If `i` or `j` is elided, their value depends on the sign of `s`. If `s` is positive, `i` become `0` and `j` become the length of the array. If `s` is negative, `i` becomes the length of the array minus one, and `j` becomes minus one. This means that `a[::-1]` is the reverse of the array `a`.

`[x, y, z]`

Create an array containing the indicated elements. Each element must have the same type and shape. At least one element must be provided - empty arrays must be constructed with the `empty` construct. This restriction is due to limited type inference in the Futhark compiler, and will hopefully be fixed in the future.

`#f e`

Access field `f` of the expression `e`, which must be a record or tuple.

`x` *binop* `y`

Apply an operator to `x` and `y`. Operators are functions like any other, and can be user-defined. Futhark pre-defines certain "magical" *overloaded* operators that work on many different types. Overloaded functions cannot be defined by the user. Both operands must have the same type. The predefined operators and their semantics are:

`**`

Power operator, defined for all numeric types.

`//`, `%%`

Division and remainder on integers, with rounding towards zero.

`*`, `/`, `%`, `+`, `-`

The usual arithmetic operators, defined for all numeric types. Note that `/` and `%` rounds towards negative infinity when used on integers - this is different from in C.

`^`, `&`, `|`, `>>`, `<<`, `>>>`

Bitwise operators, respectively bitwise xor, and, or, arithmetic shift right and left, and logical shift right. Shift amounts must be non-negative and the operands must be integers. Note that, unlike in C, bitwise operators have *higher* priority than arithmetic operators. This means that `x & y == z` is understood as `(x & y) == z`, rather than `x & (y == z)` as it would in C. Note that the latter is a type error in Futhark anyhow.

`==`, `!=`

Compare any two values of builtin or compound type for equality.

`<`, `<=`. `>`, `>=`

Company any two values of numeric type for equality.

`x && y`

Short-circuiting logical conjunction; both operands must be of type `bool`.

`x || y`

Short-circuiting logical disjunction; both operands must be of type `bool`.

`f x y z`

Apply the function `f` to the arguments `x`, `y` and `z`. Any number of arguments can be passed.

`e : t`

Annotate that `e` is expected to be of type `t`, failing with a type error if it is not. If `t` is an array with shape declarations, the correctness of the shape declarations is checked at run-time.

Due to ambiguities, this syntactic form cannot appear as an array index expression unless it is first enclosed in parentheses.

`! x`

Logical negation of `x`, which must be of type `bool`.

`- x`

Numerical negation of `x`, which must be of numeric type.

`. x`

Bitwise negation of `x`, which must be of integral type.

`if c then a else b`

If `c` evaluates to `True`, evaluate `a`, else evaluate `b`.

`let pat = e in body`

Evaluate `e` and bind the result to the pattern `pat` while evaluating `body`. The `in` keyword is optional if `body` is a `let` or `loop` expression. See also *Shape Declarations*.

`let a[i] = v in body`

Write `v` to `a[i]` and evaluate `body`. The given index need not be complete and can also be a slice, but in these cases, the value of `v` must be an array of the proper size. Syntactic sugar for `let a = a with [i] <- v in a`.

`let f params... = e in body`

Bind `f` to a function with the given parameters and definition (`e`) and evaluate `body`. The function will be treated as aliasing any free variables in `e`. The function is not in scope of itself, and hence cannot be recursive. See also *Shape Declarations*.

`loop (pat = initial) = for i < bound do loopbody in body`

The name `i` is bound here and initialised to zero.

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, increasing `i` by one after each iteration.
3. Evaluate `body` with `pat` bound to its final value.

The `= initial` can be left out, in which case initial values for the pattern are taken from equivalently named variables in the environment. I.e., `loop (x) = ...` is equivalent to `loop (x = x) = ...`.

See also *Shape Declarations*.

`loop (pat = initial) = while cond do loopbody in body`

1. Bind `pat` to the initial values given in `initial`.
2. While `cond` evaluates to true, evaluate `loopbody`, rebinding `pat` to be the value returned by the body.
3. Evaluate `body` with `pat` bound to its final value.

See also *Shape Declarations*.

iota n

An array of the integers from `0` to `n-1`. The `n` argument can be any integral type. The elements of the array will have the same type as `n`.

shape a

The shape of array `a` as an integer array. It is often more readable to use shape declaration names instead of `shape`.

replicate n x

An array consisting of `n` copies of `a`. The `n` argument must be of type `i32`.

reshape (d_1, ..., d_n) a

Reshape the elements of `a` into an `n`-dimensional array of the specified shape. The number of elements in `a` must be equal to the product of the new dimensions.

rearrange (d_1, ..., d_n) a

Permute the dimensions in the array, returning a new array. The `d_i` must be *static* integers, and constitute a proper length-`n` permutation.

For example, if `b==rearrange (2,0,1) a`, then `b[x,y,z] = a[y,z,x]`.

rotate@d i a

Rotate dimension `d` of the array `a` left by `i` elements. Intuitively, you can think of it as subtracting `i` from every index (modulo the size of the array).

For example, if `b==rotate 1 i a`, then `b[x,y+1] = a[x,y]`.

split (i_1, ..., i_n) a

Partitions the given array `a` into `n+1` disjoint arrays `(a[0...i_1-1], a[i_1...i_2-1], ..., a[i_n...])`, returned as a tuple. The split indices must be weakly ascending, ie `i_1 <= i_2 <= ... <= i_n`.

Example: `split (1,1,3) [5,6,7,8] == ([5],[],[6,7],[8])`

split@i (i_1, ..., i_n) a

Splits an array across dimension `i`, with the outermost dimension being `0`. The `i` must be a compile-time integer constant, i.e. `i` cannot be a variable.

concat a_1 ..., a_n

Concatenate the rows/elements of several arrays. The shape of the arrays must be identical in all but the first dimension. This is equivalent to `concat@0` (see below).

concat@i a_1 ... a_n

Concatenate arrays across dimension `i`, with the outermost dimension being `0`. The `i` must be a compile-time integer constant, i.e. `i` cannot be a variable.

zip x y z

Zips together the elements of the outer dimensions of arrays `x`, `y`, and `z`. Static or runtime check is performed to check that the sizes of the outermost dimension of the arrays are the same. If this property is not true, program execution stops with an error. Any number of arrays may be passed to `unzip`. If *n* arrays are given, the result will be a single-dimensional array of *n*-tuples (where the the tuple components may themselves be arrays).

zip@i x y z

Like `zip`, but operates within `i+1` dimensions. Thus, `zip@0` is equivalent to unadorned `zip`. This form is useful when zipping multidimensional arrays along the innermost dimensions.

unzip a

If the type of `a` is `[(t_1, ..., t_n)]`, the result is a tuple of *n* arrays, i.e., `([t_1], ..., [t_n])`, and otherwise a type error.

unsafe e

Elide safety checks (such as bounds checking) for operations lexically with `e`. This is useful if the compiler is otherwise unable to avoid bounds checks (e.g. when using indirect indexes), but you really do not want them here.

a with [i] <- e

Return `a`, but with the element at position `i` changed to contain the result of evaluating `e`. Consumes `a`.

map f a_1 ... a_n

Apply `f` to every element of `a_1 ... a_n` and return the resulting array. Differs from `map f (zip a_1 ... a_n)` in that `f` is called with `n` arguments, where in the latter case it is called with a single `n`-tuple argument. In other languages, this form of `map` is often called `zipWith`.

reduce f x a

Left-reduction with `f` across the elements of `a`, with `x` as the neutral element for `f`. The function `f` must be associative. If it is not, the return value is unspecified.

reduce_comm f x a

Like `reduce`, but with the added guarantee that the function `f` is *commutative*. This lets the compiler generate more efficient code. If `f` is not commutative, the return value is unspecified. You do not need to explicitly use `reduce_comm` with built-in operators like `+` - the compiler already knows that these are commutative.

scan f x a

Inclusive prefix scan. Has the same caveats with respect to associativity as `reduce`.

filter f a

Remove all those elements of `a` that do not satisfy the predicate `f`.

partition (f_1, ..., f_n) a

Divide the array `a` into disjoint partitions based on the given predicates. Each element of `a` is called with the predicates `f_1` to `f_n` in sequence, and as soon as one as one of them returns `True`, the element is added to the corresponding partition. If none of the functions return `True`, the element is added to a catch-all partition that is returned last. Always returns a tuple with *n+1* components. The partitioning is stable, meaning that elements of the partitions retain their original relative positions.

scatter as is vs

This `scatter` expression calculates the equivalent of this imperative code:

```
for index in 0..shape(is)[0]-1:
  i = is[index]
  v = vs[index]
  as[i] = v
```

The `is` and `vs` arrays must have the same outer size. `scatter` acts in-place and consumes the `as` array, returning a new array that has the same type and elements as `as`, except for the indices in `is`. If `is` contains duplicates (i.e. several writes are performed to the same location), the result is unspecified. It is not guaranteed that one of the duplicate writes will complete atomically - they may be interleaved.

Shape Declarations

Whenever a pattern occurs (in `let`, `loop`, and function parameters), as well as in return types, *shape declarations* may be used to express invariants about the shapes of arrays that are accepted or produced by the function. For example:

```
let f (a: [#n]i32) (b: [#n]i32): [n]i32 =
  map (+) a b
```

When prefixed with a `#` character, a name is *freshly bound*, whilst an unadorned name must be in scope. In the example above, `#` is not used in the return type, because we wish to refer to the `n` bound by the parameters. If we refer to the same freshly bound variable in multiple parameters (as above), each occurence must be prefixed with `#`. The sizes can also be explicitly quantified:

```
let f [n] (a: [n]i32) (b: [n]i32): [n]i32 =
  map (+) a b
```

This has the same meaning as above. It is an error to mix explicit and implicit sizes. Note that the `[n]` parameter need not be explicitly passed when calling `f`. Any explicitly bound size must be used in a parameters. This is an error:

```
let f [n] (x: i32) = n
```

A shape declaration can also be an integer constant (with no suffix). The dimension names bound can be used as ordinary variables within the scope of the parameters. If a function is called with arguments, or returns a value, that does not fulfill the shape constraints, the program will fail with a runtime error. Likewise, if a pattern with shape declarations is attempted bound to a value that does not fulfill the invariants, the program will fail with a runtime error. For example, this will fail:

```
let x: [3]i32 = iota 2
```

While this will succeed and bind `n` to `2`:

```
let [n] x: [n]i32 = iota 2
```

Declarations

```
dec  ::=  fun_bind | val_bind | type_bind | mod_bind | mod_type_bind
        | ``open'' mod_exp+
        | default_dec
        | ``import'' stringlit
```

Declaring Functions and Values

```
fun_bind  ::=  (``let'' | ``entry'') id type_param* pat+ ['':''  type] ``='' exp
             | (``let'' | ``entry'') pat binop pat ['':''  type] ``='' exp

val_bind  ::=  ``let'' id ['':''  type] ``='' exp
```

Functions and values must be defined before they are used. A function declaration must specify the name, parameters, return type, and body of the function:

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

Type inference is not supported, and functions are fully monomorphic. A parameter is written as `(name: type)`. Functions may not be recursive. Optionally, the programmer may put *shape declarations* in the return type and parameter types; see *Shape Declarations*. A function can be *polymorphic* by using type parameters, in the same way as for *Type Abbreviations*:

```
let reverse [n] 't (xs: [n]t): [n]t = xs[::-1]
```

Shape and type parameters are not passed explicitly when calling function, but are automatically derived.

User-Defined Operators

Infix operators are defined much like functions:

```
let (p1: t1) op (p2: t2): rt = ...
```

For example:

```
let (a:i32,b:i32) +^ (c:i32,d:i32) = (a+c, b+d)
```

A valid operator name is a non-empty sequence of characters chosen from the string `"+-*/%=!><&^"`. The fixity of an operator is determined by its first characters, which must correspond to a built-in operator. Thus, `+^` binds like `+`, whilst `*^` binds like `*`. The longest such prefix is used to determine fixity, so `>>=` binds like `>>`, not like `>`.

It is not permitted to define operators with the names `&&` or `||` (although these as prefixes are accepted). This is because a user-defined version of these operators would not be short-circuiting. User-defined operators behave exactly like functions, except for syntactically.

A built-in operator can be shadowed (i.e. a new `+` can be defined). This will result in the built-in polymorphic operator becoming inaccessible, except through the `Intrinsics` module.

Entry Points

Apart from declaring a function with the keyword `fun`, it can also be declared with `entry`. When the Futhark program is compiled any function declared with `entry` will be exposed as an entry point. If the Futhark program has been compiled as a library, these are the functions that will be exposed. If compiled as an executable, you can use the `--entry-point` command line option of the generated executable to select the entry point you wish to run.

Any function named `main` will always be considered an entry point, whether it is declared with `entry` or not.

Value Declarations

A named value/constant can be declared as follows:

```
let name: type = definition
```

The definition can be an arbitrary expression, including function calls and other values, although they must be in scope before the value is defined. The type annotation can be elided if the value is defined before it is used.

Values can be used in shape declarations, except in the return value of entry points.

Type Abbreviations

```
type_bind   ::=   ``type'' id type_param* ``='' type
type_param  ::=   ``['' id ``]'' | ``'''' id
```

Type abbreviations function as shorthands for purpose of documentation or brevity. After a type binding `type t1 = t2`, the name `t1` can be used as a shorthand for the type `t2`. Type abbreviations do not create new unique types. After the previous binding, the types `t1` and `t2` are entirely interchangeable.

A type abbreviation can have zero or more parameters. A type parameter enclosed with square brackets is a *shape parameter*, and can be used in the definition as an array dimension size, or as a dimension argument to other type abbreviations. When passing an argument for a shape parameter, it must be encloses in square brackets. Example:

```
type two_intvecs [n] = ([n]i32, [n]i32)

let (a,b): two_intvecs [2] = (iota 2, replicate 2 0)
```

Shape parameters work much like shape declarations for arrays. Like shape declarations, they can be elided via square brackets containing nothing.

A type parameter prefixed with a single quote is a *type parameter*. It is in scope as a type in the definition of the type

abbreviation. Whenever the type abbreviation is used in a type expression, a type argument must be passed for the parameter. Type arguments need not be prefixed with single quotes:

```
type two_vecs [n] 't = ([n]t, [n]t)
type two_intvecs [n] = two_vecs [n] i32
let (a,b): two_vecs [2] i32 = (iota 2, replicate 2 0)
```

When using uniqueness attributes with type abbreviations, inner uniqueness attributes are overrided by outer ones:

```
type unique_ints = *[]i32
type nonunique_int_lists = []unique_ints
type unique_int_lists = *nonunique_int_lists

-- Error: using non-unique value for a unique return value.
let f (p: nonunique_int_lists): unique_int_lists = p
```

Module System

```
mod_bind       ::=   ``module'' id mod_param+ ``='' ['':''  mod_type_exp] ``='' mod_exp
mod_param      ::=   ``('' id '':''  mod_type_exp '')''
mod_type_bind  ::=   ``module'' ``type'' id type_param* ``='' mod_type_exp
```

Futhark supports an ML-style higher-order module system. *Modules* can contain types, functions, and other modules. *Module types* are used to classify the contents of modules, and *parametric modules* are used to abstract over modules (essentially module-level functions). In Standard ML, modules, module types and parametric modules are called structs, signatures, and functors, respectively.

Named modules are declared as:

```
module name = module expression
```

A named module type is defined as:

```
module type name = module type expression
```

Where a module expression can be the name of another module, an application of a parametric module, or a sequence of declarations enclosed in curly braces:

```
module Vec3 = {
  type t = ( f32 , f32 , f32 )
  let add(a: t) (b: t): t =
    let (a1, a2, a3) = a in
    let (b1, b2, b3) = b in
    (a1 + b1, a2 + b2 , a3 + b3)
}

module AlsoVec3 = Vec3
```

Functions and types within modules can be accessed using dot notation:

```
type vector = Vec3.t
let double(v: vector): vector = Vec3.add v v
```

We can also use `open Vec3` to bring the names defined by `Vec3` into the current scope. Multiple modules can be opened simultaneously by separating their names with spaces. In case several modules define the same names, the

ones mentioned last take precedence. The first argument to `open` may be a full module expression.

Named module types are defined as:

```
module type ModuleTypeName = module type expression
```

A module type expression can be the name of another module type, or a sequence of *specifications*, or *specs*, enclosed in curly braces. A spec can be a *value spec*, indicating the presence of a function or value, an *abstract type spec*, or a *type abbreviation spec*. For example:

```
module type Addable = {
  type t                  -- abstract type spec
  type two_ts = (t,t)     -- type abbreviation spec
  val add: t -> t -> t    -- value spec
}
```

This module type specifies the presence of an *abstract type* `t`, as well as a function operating on values of type `t`. We can use *module type ascription* to restrict a module to what is exposed by some module type:

```
module AbstractVec = Vec3 : Addable
```

The definition of `AbstractVec.t` is now hidden. In fact, with this module type, we can neither construct values of type `AbstractVec.T` or convert them to anything else, making this a rather useless use of abstraction. As a derived form, we can write `module M: S = e` to mean `module M = e : S`.

Parametric modules allow us to write definitions that abstract over modules. For example:

```
module Times(M: Addable) = {
  let times (x: M.t) (k: int): M.t =
    loop (x' = x) = for i < k do
      T.add x' x
    in x'
}
```

We can instantiate `Times` with any module that fulfills the module type `Addable` and get back a module that defines a function `times`:

```
module Vec3Times = Times(Vec3)
```

Now `Vec3Times.times` is a function of type `Vec3.t -> int -> Vec3.t`.

Module Expressions

```
mod_exp  ::=  qualid
            | mod_exp ''':''  mod_type_exp
            | ``\'' ``('' id ''':''  mod_type_exp '')'' [''':''  mod_type_exp] ``='' mod_e
            | mod_exp mod_exp
            | ``('' mod_exp '')''
            | ``{'' dec* ``}''
            | ``import'' stringlit
```

Module Type Expressions

```
mod_type_exp  ::=  qualid
                 | ``{'' spec* ``}''
                 | mod_type_exp ``with'' qualid ``='' type
                 | ``('' mod_type_exp '')''
                 | ``('' id '':''  mod_type_exp '')'' ``->'' mod_type_exp
                 | mod_type_exp ``->'' mod_type_exp


spec       ::=  ``val'' id type_param* '':''  spec_type
              | ``val'' binop '':''  spec_type
              | ``type'' id type_param* ``='' type
              | ``type id type_param*
              | ``module'' id '':''  mod_type_exp
              | ``include'' mod_type_exp
spec_type  ::=  type | type ``->'' spec_type
```

Referring to Other Files

You can refer to external files in a Futhark file like this:

```
import "module"
```

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

You can also include files from subdirectories:

```
include "path/to/a/file"
```

The above will include the file `path/to/a/file.fut`. When importing a nonlocal file (such as the standard library or the compiler search path), the path must begin with a forward slash.

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

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

Literal Defaults

```
default_dec  ::=  ``default'' (int_type)
                | ``default'' (float_type)
                | ``default'' (int_type, float_type)
```

By default, Futhark interprets integer literals as `i32` values, and decimal literals (integer literals containing a decimal point) as `f64` values. These defaults can be changed using the Haskell-inspired `default` keyword.

To change the `i32` default to e.g. `i64`, type the following at the top of your file:

```
default(i64)
```

To change the `f64` default to `f32`, type the following at the top of your file:

```
default(f32)
```

To change both, type:

```
default(i64,f32)
```

CHAPTER 5

C Porting Guide

This short document contains a collection of tips and tricks for porting simple numerical C code to futhark. Futhark's sequential fragment is powerful enough to permit a rather straightforward translation of sequential C code that does not rely on pointer mutation. Additionally, we provide hints on how to recognise C coding patterns that are symptoms of C's weak type system, and how better to organise it in Futhark.

One intended audience of this document is a programmer who needs to translate a benchmark application written in C, or needs to use a simple numerical algorithm that is already available in the form of C source code.

Where This Guide Falls Short

Some C code makes use of unstructured returns and nonlocal exits (`return` inside loops, for example). These are not easy to express in Futhark, and will require massaging the control flow a bit. C code that uses `goto` is likewise not easy to port.

Types

Futhark provides scalar types that match the ones commonly used in C: `u8`/`u16`/`u32`/`u64` for the unsigned integers, `i8`/`i16`/`i32`/`i64` for the signed, and `f32`/`f64` for `float` and `double` respectively. In contrast to C, Futhark does not automatically promote types in expressions - you will have to manually make sure that both operands to e.g. a multiplication are of the exact same type. This means that you will need to understand exactly which types a given expression in original C program operates on, which generally boils down to converting the type of the (type-wise) smaller operand to that of the larger. Note that the Futhark `bool` type is not considered a number.

Operators

Most of the C operators can be found in Futhark with their usual names. Note however that the Futhark `/` and `%` operators for integers round towards negative infinity, whereas their counterparts in C round towards zero. You can

write `//` and `%%` if you want the C behaviour. There is no difference if both operands are non-zero, but `//` and `%%` may be slightly faster. For unsigned numbers, they are exactly the same.

Variable Mutation

As a sequential language, most C programs quite obviously rely heavily on mutating variables. However, in many programs, this is done in a static manner without indirection through pointers (except for arrays; see below), which is conceptually similar to just declaring a new variable of the same name that shadows the old one. If this is the case, a C assignment can generally be translated to just a `let`-binding. As an example, let us consider the following function for computing the modular multiplicative inverse of a 16-bit unsigned integer (part of the IDEA encryption algorithm):

```
static uint16_t ideaInv(uint16_t a) {
  uint32_t b;
  uint32_t q;
  uint32_t r;
  int32_t t;
  int32_t u;
  int32_t v;

  b = 0x10001;
  u = 0;
  v = 1;

  while(a > 0)
    {
      q = b / a;
      r = b % a;

      b = a;
      a = r;

      t = v;
      v = u - q * v;
      u = t;
    }

  if(u < 0)
    u += 0x10001;

  return u;
}
```

Each iteration of the loop mutates the variables `a`, `b`, `v`, and `u` in ways that are visible to the following iteration. Conversely, the "mutations" of `q`, `r`, and `t` are not truly mutations, and the variable declarations could be moved inside the loop if we wished. Presumably, the C programmer left them outside for aesthetic reasons. When translating such code, it is important to determine exactly how much *true* mutation is going on, and how much is just reuse of variable space. This can usually be done by checking whether a variable is read before it is written in any given iteration - if not, then it is not true mutation. The variables that change value from one iteration of the loop to the next will need to be maintained as *merge parameters* of the Futhark `do`-loop.

The Futhark program resulting from a straightforward port looks as follows:

```
let main(a: u16): u16 =
  let b = 0x10001u32
  let u = 0i32
  let v = 1i32
```

```
  loop ((a,b,u,v)) = while a > 0u16 do
    let q = b / u32(a)
    let r = b % u32(a)

    let b = u32(a)
    let a = u16(r)

    let t = v
    let v = u - i32(q) * v
    let u = t
    in (a,b,u,v)

  in u16(if u < 0 then u + 0x10001 else u)
```

Note the heavy use of type conversion and type suffixes for constants. This is necessary due to Futhark's lack of implicit conversions. Note also the conspicuous way in which the `do`-loop is written - the result of a loop iteration consists of variables whose names are identical to those of the merge parameters.

This program can still be massaged to make it more idiomatic Futhark - for example, the variable `t` only serves to store the old value of `v` that is otherwise clobbered. This can be written more elegantly by simply inlining the expressions in the result part of the loop body.

Arrays

Dynamically sized multidimensional arrays are somewhat awkward in C, and are often simulated via single-dimensional arrays with explicitly calculated indices:

```
a[i * M + j] = foo;
```

This indicates a two-dimensional array `a` whose *inner* dimension is of size `M`. We can usually look at where `a` is allocated to figure out what the size of the outer dimension must be:

```
a = malloc(N * M * sizeof(int));
```

We see clearly that `a` is a two-dimensional integer array of size `N` times `M` - or of type `[N][M]i32` in Futhark. Thus, the update expression above would be translated as:

```
let a[i,j] = foo in
...
```

C programs usually first allocate an array, then enter a loop to provide its initial values. This is not possible in Futhark - consider whether you can write it as a `replicate`, an `iota`, or a `map`. In the worst case, use `replicate` to obtain an array of the desired size, then use a `do`-loop with in-place updates to initialise it (but note that this will run stricly sequentially).

CHAPTER 6

Hacking on the Futhark Compiler

The Futhark compiler is a significant body of code with a not entirely straightforward design. The main reference is the documentation of the compiler internals that is automatically generated by Haddock. If you feel that it is incomplete, or lacks an explanation, then feel free to report it is an an issue on the Github page. Documentation bugs are bugs too.

The Futhark compiler is built using Stack. It's a good idea to familiarise yourself with how it works. As a starting point, here are a few hints:

- To test with different GHC versions, point the `STACK_YAML` environment variable at another file. For example, to build using the Stack LTS 1.15 snapshot (GHC 7.8), we would run:

```
$ STACK_YAML=stack-lts-1.15.yaml stack build
```

- When testing, pass `--fast` to `stack` to disable the GHC optimiser. This speeds up builds considerably (although it still takes a while). The resulting Futhark compiler will run slower, but it is not something you will notice for small test programs.
- When debugging, pass `--profile` to `stack`. This will build the Futhark compiler with debugging information (not just profiling). In particular, hard crashes will print a stack trace. You can also get actual profiling information by passing `+RTS -pprof-all -RTS` to the Futhark compiler. This asks the Haskell runtime to print profiling information to a file. For more information, see the Profiling chapter in the GHC User Guide.

Debugging Internal Type Errors

The Futhark compiler uses a typed core language, and the type checker is run after every pass. If a given pass produces a program with inconsistent typing, the compiler will report an error and abort. While not every compiler bug will manifest itself as a core language type error (unfortunately), many will. To write the erroneous core program to a file in case of type error, pass `-v filename` to the compiler. This will also enable verbose output, so you can tell which pass fails. The `-v` option is also useful when the compiler itself crashes, as you can at least tell where in the pipeline it got to.

Checking Generated Code

Hacking on the compiler will often involve inspecting the quality of the generated code. The recommended way to do this is to use *futhark-c* or *futhark-opencl* to compile a Futhark program to an executable. These backends insert various forms of instrumentation that can be enabled by passing run-time options to the generated executable.

- As a first resort, use `-t` option to use the built-in runtime measurements. A nice trick is to pass `-t /dev/stderr`, while redirecting standard output to `/dev/null`. This will print the runtime on the screen, but not the execution result.
- Optionally use `-r` to ask for several runs, e.g. `-r 10`. If combined with `-t`, this will cause several runtimes to be printed (one per line). The *futhark-bench* tool itself uses `-t` and `-r` to perform its measurements.
- Pass `-m` to have the program print information on allocation and deallocation of memory.
- (*futhark-opencl* only) Use the `-s` option to enable synchronous execution. `clFinish()` will be called after most OpenCL operations, and a running log of kernel invocations will be printed. At the end of execution, the program prints a table summarising all kernels and their total runtime and average runtime.

Using the `futhark` Tool

For debugging specific compiler passes, there is a tool simply called *futhark*, which allows you to tailor your own compilation pipeline using command line options. It is also useful for seeing what the AST looks like after specific passes.

CHAPTER 7

Binary Data Format

Futhark programs compiled to an executable support both textual and binary input. Both are read via standard input, and can be mixed, such that one argument to an entry point may be binary, and another may be textual. The binary input format takes up significantly less space on disk, and can be read much faster than the textual format. This chapter describes the binary input format and its current limitations. The input formats (whether textual or binary) are not used for Futhark programs compiled to libraries, which instead use whichever format is supported by their host language.

Currently reading binary input is only supported for the C programs generated by `futhark-c` and `futhark-opencl`.

Currently there is no way to convert input from the textual format to the binary format, but you can generate random data in the binary format with *futhark-dataset*.

Currently there is no way to output data in the binary format from Futhark programs.

Specification

Elements that are bigger than one byte are always stored using little endian – we mostly run our code on x86 hardware so this seemed like a reasonable choice.

When reading input for an argument to the entry function, we need to be able to differentiate between text and binary input. If the first non-whitespace character of the input is a `b` we will parse this argument as binary, otherwise we will parse it in text format. Allowing preceding whitespace characters makes it easy to use binary input for some arguments, and text input for others.

The general format has this header:

```
b <version> <num_dims> <type>
```

Where `version` is a byte containing the version of the binary format used for encoding (currently 1), `num_dims` is the number of dimensions in the array as a single byte (0 for scalar), and `type` is a 4 character string describing the type of the values(s) – see below for more details.

Encoding a scalar value is done by appending the binary little endian representation of it:

```
b <version> 0 <type> <value>
```

To encode an array we must encode the number of dimensions `n` as a single byte, each dimension `dim_i` as an unsigned 64-bit little endian integer, and finally all the values in their binary little endian representation:

```
b <version> <n> <type> <dim_1> <dim_2> ... <dim_n> <values>
```

Type Values

A type is identified by a 4 character ASCII string (four bytes). Valid types are:

```
"  i8"
" i16"
" i32"
" i64"
" f32"
" f64"
"bool"
```

CHAPTER 8

futhark

SYNOPSIS

futhark [options...] infile

DESCRIPTION

This is a Futhark compiler development tool, intentionally undocumented and intended for use in developing the Futhark compiler, not for programmers writing in Futhark. To compile Futhark code, use one of the compilers, e.g. `futhark-c` or `futhark-opencl`.

For documentation on the Futhark language itself, see:

```
http://futhark.readthedocs.io
```

SEE ALSO

futhark-c(1), futhark-opencl(1)

CHAPTER 9

futhark-c

SYNOPSIS

futhark-c [-V] [-o outfile] infile

DESCRIPTION

`futhark-c` translates a Futhark program to sequential C code, and then compiles that C code with gcc(1) to an executable binary program. The standard Futhark optimisation pipeline is used, and GCC is invoked with `-O3`, `-lm`, and `-std=c99`.

The resulting program will read the arguments to the `main` function from standard input and print its return value on standard output. The arguments are read and printed in Futhark syntax, just like futharki(1).

OPTIONS

-o outfile Where to write the resulting binary. By default, if the source program is named 'foo.fut', the binary will be named 'foo'.

-v verbose Enable debugging output. If compilation fails due to a compiler error, the result of the last successful compiler step will be printed to standard error.

-h Print help text to standard output and exit.

-V Print version information on standard output and exit.

SEE ALSO

futharki(1), futhark-test(1)

CHAPTER 10

futhark-opencl

SYNOPSIS

futhark-opencl [-V] [-o outfile] infile

DESCRIPTION

`futhark-opencl` translates a Futhark program to C code invoking OpenCL kernels, and then compiles that C code with gcc(1) to an executable binary program. The standard Futhark optimisation pipeline is used, and GCC is invoked with `-O3`. The first device of the first OpenCL platform is used.

The resulting program will otherwise behave exactly as one compiled with `futhark-c`.

OPTIONS

-o outfile	Where to write the resulting binary. By default, if the source program is named 'foo.fut', the binary will be named 'foo'.
-v verbose	Enable debugging output. If compilation fails due to a compiler error, the result of the last successful compiler step will be printed to standard error.
-h	Print help text to standard output and exit.
-V	Print version information on standard output and exit.

SEE ALSO

futharki(1), futhark-test(1), futhark-c(1)

CHAPTER 11

futhark-py

SYNOPSIS

futhark-py [-V] [-o outfile] infile

DESCRIPTION

`futhark-py` translates a Futhark program to sequential Python code.

The resulting program will read the arguments to the `main` function from standard input and print its return value on standard output. The arguments are read and printed in Futhark syntax, just like futharki(1).

The generated code is very slow, and likely not very useful. It might be more interesting to use this commands big brother, `futhark-pyopencl`.

OPTIONS

-o outfile	Where to write the resulting binary. By default, if the source program is named 'foo.fut', the binary will be named 'foo'.
-v verbose	Enable debugging output. If compilation fails due to a compiler error, the result of the last successful compiler step will be printed to standard error.
-h	Print help text to standard output and exit.
-V	Print version information on standard output and exit.

SEE ALSO

futhark-pyopencl(1)

CHAPTER 12

futhark-pyopencl

SYNOPSIS

futhark-pyopencl [-V] [-o outfile] infile

DESCRIPTION

`futhark-pyopencl` translates a Futhark program to Python code invoking OpenCL kernels. By default, the program uses the first device of the first OpenCL platform - this can be changed by passing `-p` and `-d` options to the generated program (not to `futhark-pyopencl` itself).

The resulting program will otherwise behave exactly as one compiled with `futhark-py`. While the sequential host-level code is pure Python and just as slow as in `futhark-py`, parallel sections will have been compiled to OpenCL, and runs just as fast as when using `futhark-c(1)`. The kernel launch overhead is significantly higher, however, so a good rule of thumb when using `futhark-pyopencl` is to aim for having fewer but longer-lasting parallel sections.

OPTIONS

-o outfile Where to write the resulting binary. By default, if the source program is named 'foo.fut', the binary will be named 'foo'.

-v verbose Enable debugging output. If compilation fails due to a compiler error, the result of the last successful compiler step will be printed to standard error.

--library Instead of compiling to an executable program, generate a Python module that can be ported by other Python code. The module will contain a class of the same name as the Futhark source file with `.fut` removed. Objects of the class define one method per entry point in the Futhark program, with matching parameters and return value.

-h Print help text to standard output and exit.

-V Print version information on standard output and exit.

SEE ALSO

futhark-py(1), futhark-openlk(1)

CHAPTER 13

futharki

SYNOPSIS

futharki [infile]

DESCRIPTION

When run with no options, start an interactive futharki session. This will let you interactively enter expressions which are then immediately interpreted. You cannot enter function definitions and the like directly, but you can load Futhark source files using the `:load` command. Use the `:help` command to see a list of commands. All commands are prefixed with a colon.

When `futharki` is run with a Futhark program as the command line option, the program is executed by evaluating the `main` function, and the result printed on standard output. The parameters to `main` are read from standard input.

Futharki will run the standard Futhark optimisation pipeline before execution, but the interpreter is still very slow.

OPTIONS

-e NAME	Run the given entry point instead of `main`.
-h	Print help text to standard output and exit.
-V	Print version information on standard output and exit.

BUGS

Input editing is not yet implemented; we recommend running `futharki` via `rlwrap(1)`.

SEE ALSO

futhark-c(1), futhark-test(1)

CHAPTER 14

futhark-test

SYNOPSIS

futhark-test [-c | -C | -t | -i] infiles...

DESCRIPTION

This program is used to integration-test the Futhark compiler itself. You must have futhark-c(1) and futharki(1) in your `PATH` when running `futhark-test`. If a directory is given, all contained files with a `.fut` extension are considered.

A Futhark test program is an ordinary Futhark program, with at least one test block describing input/output test cases and possibly other options. A test block consists of commented-out text with the following overall format:

```
description
==
cases...
```

The `description` is an arbitrary (and possibly multiline) human-readable explanation of the test program. It is separated from the test cases by a line containing just ==. Any comment starting at the beginning of the line, and containing a line consisting of just ==, will be considered a test block. The format of a test case is as follows:

```
[tags { tags... }]
[entry: name]
[compiled|nobench] input {
  values...
}
output { values... } | error: regex
```

If `compiled` is present before the `input` keyword, this test case will never be passed to the interpreter. This is useful for test cases that are annoyingly slow to interpret. The `nobench` keyword is for data sets that are too small to be worth benchmarking, and only has meaning to futhark-bench(1).

After the `input` block, the expected result of the test case is written as either another block of values, or an expected run-time error, in which a regular expression can be used to specify the exact error message expected. If no regular expression is given, any error message is accepted. If neither `output` nor `error` is given, the program will be expected to execute succesfully, but its output will not be validated.

Alternatively, instead of input-output pairs, the test cases can simply be a description of an expected compile time type error:

```
error: regex
```

This is used to test the type checker.

By default, both the interpreter and compiler is run on all test cases (except those that have specified `compiled`), although this can be changed with command-line options to `futhark-test`.

Tuple syntax is not supported when specifying input and output values. Instead, you can write an N-tuple as its constituent N values. Beware of syntax errors in the values - the errors reported by `futhark-test` are very poor.

An optional tags specification is permitted in the first test block. This section can contain arbitrary tags that classify the benchmark:

```
tags { names... }
```

Tag are sequences of alphanumeric characters, with each tag seperated by whitespace. Any program with the `disable` tag is ignored by `futhark-test`.

Another optional directive is `entry`, which specifies the entry point to be used for testing. This is useful for writing programs that test libraries with multiple entry points. The `entry` directive affects subsequent input-output pairs in the same comment block, and may only be present immediately preceding these input-output pairs. If no `entry` is given, the default of `main` is assumed. See below for an example.

For many usage examples, see the `tests` directory in the Futhark source directory. A simple example can be found in `EXAMPLES` below.

OPTIONS

--nobuffer Print each result on a line by itself, without buffering.

--exclude=tag Ignore benchmarks with the specified tag.

-c Only run compiled code - do not run any interpreters.

-i Only interpret - do not run any compilers.

-C Compile the programs, but do not run them.

-t Type-check the programs, but do not run them.

--compiler=program The program used to compile Futhark programs. This option can be passed multiple times, with the last taking effect. The specified program must support the same interface as `futhark-c`.

--interpreter=program Like `--compiler`, but for interpretation.

--typechecker=program Like `--compiler`, but for when execution has been disabled with `-t`.

--pass-option=opt Pass an option to benchmark programs that are being run. For example, we might want to run OpenCL programs on a specific device:

```
futhark-bench prog.fut --compiler=futhark-opencl --pass-
↪option=-dHawaii
```

EXAMPLES

The following program tests simple indexing and bounds checking:

```
-- Test simple indexing of an array.
-- ==
-- tags { firsttag secondtag }
-- input {
--   [4,3,2,1]
--   1
-- }
-- output {
--   3
-- }
-- input {
--   [4,3,2,1]
--   5
-- }
-- error: Assertion.*failed

let main(a: []i32:, i: i32): i32 =
  a[i]
```

The following program contains two entry points, both of which are tested:

```
let add(x: i32, y: i32): i32 = x + y

-- Test the add1 function.
-- ==
-- entry: add1
-- input { 1 } output { 2 }

entry add1(x: i32): i32 = add x 1

-- Test the sub1 function.
-- ==
-- entry: sub1
-- input { 1 } output { 0 }

entry sub1(x: i32): i32 = add x (-1)
```

SEE ALSO

futhark-c(1), futharki(1)

CHAPTER 15

futhark-bench

SYNOPSIS

futhark-bench [–runs=count | –compiler=program | –json | –no-validate] programs...

DESCRIPTION

This program is used to benchmark Futhark programs. In addition to the notation used by `futhark-test(1)`, this program also supports the dataset keyword `nobench`. This is used to indicate datasets that are worthwhile for testing, but too small to be worth benchmarking.

Programs are compiled using the specified compiler (`futhark-c` by default), then run a number of times for each data set, and the average runtime printed on standard output. A program will be ignored if it contains no data sets - it will not even be compiled. Only data sets that use the default entry point (`main`) are considered.

If compilation or running fails, an error message will be printed and benchmarking will continue, but a non-zero exit code will be returned at the end.

OPTIONS

--runs=count The number of runs per data set.

--compiler=program The program used to compile Futhark programs. This option can be passed multiple times, resulting in multiple compilers being used for each test case. The specified program must support the same interface as `futhark-c`.

--json=file Write raw results in JSON format to the specified file.

--pass-option=opt Pass an option to benchmark programs that are being run. For example, we might want to run OpenCL programs on a specific device:

```
futhark-bench prog.fut --compiler=futhark-opencl --pass-
↪option=-dHawaii
```

--timeout=seconds If the runtime for a dataset exceeds this integral number of seconds, it is aborted. Note that the time is allotted not *per run*, but for *all runs* for a dataset. A twenty second limit for ten runs thus means each run has only two seconds (minus initialisation overhead).

A negative timeout means to wait indefinitely.

EXAMPLES

The following program benchmarks how quickly we can sum arrays of different sizes:

```
-- How quickly can we reduce arrays?
--
-- ==
-- nobench input { 0 }
-- output { 0 }
-- input { 100 }
-- output { 4950 }
-- compiled input { 100000 }
-- output { 704982704 }
-- compiled input { 100000000 }
-- output { 887459712 }

let main(n: i32): i32 =
  reduce (+) 0 (iota n)
```

SEE ALSO

futhark-c(1), futhark-test(1)

CHAPTER 16

futhark-dataset

SYNOPSIS

futhark-dataset options...

DESCRIPTION

Generate random values in Futhark syntax, which can be useful when generating input datasets for program testing. All Futhark primitive types are supported. Tuples are not supported. Arrays of specific (non-random) sizes can be generated. You can specify maximum and minimum bounds for values, as well as the random seed used when generating the data. The generated values are written to standard output.

OPTIONS

-g type, --generate type Generate a value of the indicated type, e.g. `-g i32` or `-g [10]f32`.

-s int Set the seed used for the RNG. Zero by default.

–T-bounds=min:max Set inclusive lower and upper bounds on generated values of type `T`. `T` is any primitive type, e.g. `i32` or `f32`. The bounds apply to any following uses of the `-g` option.

You can alter the output format using the following flags. To use them, add them before data generation (–generate):

--text Output data in text format (must precede –generate). Default.

-b –binary Output data in binary Futhark format (must precede –generate).

--binary-no-header Output data in binary Futhark format without header (must precede –generate).

--binary-only-header Only output binary Futhark format header for data (must precede –generate).

EXAMPLES

Generate an array of floating-point numbers and an array of indices into that array:

```
futhark-dataset -g [10]f32 --i32-bounds=0:9 -g [100]i32
```

To generate binary data, the `--binary` must come before the `--generate`:

```
futhark-dataset --binary --generate=[42]i32
```

It is possible to generate a single file containing a payload of values, and use custom headers to make different interpretations as e.g. a 2D array. For example we can generate a file only containing 256 `i32` values by:

```
futhark-dataset --binary-no-header --generate=[256]i32 > 256.dat
```

Then we can run our program with different 2D-array configurations, without generating the array elements for each of them:

```
futhark-dataset --binary-only-header --generate=[1][256]i32 | cat - 256.dat |
↪<Futhark program>
futhark-dataset --binary-only-header --generate=[2][128]i32 | cat - 256.dat |
↪<Futhark program>
futhark-dataset --binary-only-header --generate=[4][64]i32 | cat - 256.dat | <Futhark␣
↪program>
```

SEE ALSO

futhark-test(1), futhark-bench(1)