Futhark Documentation

Release 0.7.0

DIKU

Jul 16, 2018

Table of Contents

Welcome to the documentation for the Futhark compiler and language. For a basic introduction, please see the Futhark website. To get started, read the page on *Installation*. Once the compiler has been installed, you might want to take a look at *Basic Usage*. This User's Guide contains a *Language Reference*, but new Futhark programmers are probably better served by reading Parallel Programming in Futhark first.

Documentation for the included basis library is also available online.

The particularly interested reader may also want to peruse the publications, or the development blog.

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*. If you are using macOS, read *Installing Futhark on macOS*.

1.1 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.

You can either retrieve a source release tarball or perform a checkout of our Git repository:

```
$ git clone https://github.com/diku-dk/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.

1.2 Installing from a precompiled snapshot

Tarballs of binary releases can be found online, but are available only for very few platforms (as of this writing, only GNU/Linux on x86_64).

Furthermore, 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 for users who wish to use the most recent code, but are unable 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.

1.3 Installing Futhark on macOS

Futhark is available on Homebrew, and the latest release can be installed via:

```
$ brew install futhark
```

macOS ships with one OpenCL platform and various devices. One of these devices is always the CPU, which is not fully functional, and is never picked by Futhark by default. You can still select it manually with the usual mechanisms (see *Executable Options*), but it is unlikely to be able to run most Futhark programs. Depending on the system, there may also be one or more GPU devices, and Futhark will simply pick the first one as always. On multi-GPU MacBooks, this is is the low-power integrated GPU. It should work just fine, but you might have better performance if you use the dedicated GPU instead.

1.4 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.

1.4.1 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!

1.4.2 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']
```

(continues on next page)

(continued from previous page)

```
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.

There are two main ways of compiling a Futhark program: to an executable (by using `--executable`, which is the default), and to a library (`--library`). Executables can be run immediately, but are useful mostly for testing and benchmarking. Libraries can be called from non-Futhark code.

2.1 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 compiler will automatically invoke `gcc` to produce 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. For more details on specific compilers, see their individual manual pages.

Executables generated by the various Futhark compilers share a common command-line interface, but may also individually support more options. When a Futhark program is run, execution starts at one of its *entry points*. By default, the entry point named `main` is run. An alternative entry point can be indicated by using the `-e` option. All entry point functions must be declared appropriately in the program (see *Entry Points*). If the entry point takes any parameters, these will be read from standard input in a subset of the Futhark syntax. A binary input format is also supported; see *Binary Data Format*. The result of the entry point is printed to standard output.

Only a subset of all Futhark values can be passed to an executable. Specifically, only primitives and arrays of primitive types are supported. In particular, nested tuples and arrays of tuples are not permitted. Non-nested tuples are supported are supported as simply flat values. This restriction is not present for Futhark programs compiled to libraries. If an

entry point *returns* any such value, its printed representation is unspecified. As a special case, an entry point is allowed to return a flat tuple.

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 only marginally useful in its present state.

2.1.1 Executable Options

All generated executables support the following options.

`-t FILE`

Print the time taken to execute the program to the indicated file, an integral number of microseconds. The time taken to perform tup or teardown, including reading the input or writing the sult, is not included in the measurement. See the documentation r specific compilers to see exactly what is measured.

`-r RUNS`

Run the specified entry point the given number of times (plus a warmup run). The program result is only printed once, after the last run. If combined with `-t`, one measurement is printed per run. This is a good way to perform benchmarking.

`-D`

Print debugging information on standard error. Exactly what is printed, and how it looks, depends on which Futhark compiler is used. This option may also enable more conservative (and slower) execution, such as frequently synchronising to check for errors.

`-b`

Print the result using the binary data format (*Binary Data Format*). For large outputs, this is significantly faster and takes up less space.

The following options are supported by executables generated by `futhark-opencl` and `futhark-pyopencl`:

`-p PLATFORM`

Pick the first OpenCL platform whose name contains the given string. The special string `#k`, where `k` is an integer, can be used to pick the *k*-th platform, numbered from zero.

`-d DEVICE`

Pick the first OpenCL device whose name contains the given string. The special string `#k`, where `k` is an integer, can be used to pick the *k*-th device, numbered from zero. If used in conjunction with `-p`, only the devices from matching platforms are considered.

`--dump-opencl FILE`

Dump the embedded OpenCL program to the indicated file. Useful if you want to see what is actually being executed.

`--load-opencl FILE`

Instead of using the embedded OpenCL program, load it from the indicated file. This is extremely unlikely to result in succesful execution unless this file is the result of a previous call to `--dump-opencl` (perhaps lightly modified).

There is rarely a need to use both `-p` and `-d`. For example, to run on the first available NVIDIA GPU, `-p NVIDIA` is sufficient, as there is likely only a single device associated with this platform. On *nix (including macOS), the clinfo tool (available in many package managers) can be used to determine which OpenCL platforms and devices are available on a given system. On Windows, CPU-z can be used.

2.2 Compiling to Library

While compiling a Futhark program to an executable is useful for testing, it is not suitable for production use. Instead, a Futhark program should be compiled into a reusable library in some target language, enabling integration into a larger program. Four of the Futhark compilers support this: `futhark-c`, `futhark-opencl`, `futhark-py`, and `futhark-pyopencl`.

2.2.1 General Concerns

Futhark entry points are mapped to some form of function or method in the target language. Generally, an entry point taking *n* parameters will result in a function taking *n* parameters. Extra parameters may be added to pass in context data, or *out*-parameters for writing the result, for target languages that do not support multiple return values from functions.

Not all Futhark types can be mapped cleanly to the target language. Arrays of tuples are the most common case. In such cases, *opaque types* are used in the generated code. Values of these types cannot be directly inspected, but can be passed back to Futhark entry points. In the general case, these types will be named with a random hash. However, you if you insert explicit type annotations (and the type name contains only characters valid for identifiers for the used backend), the indicated name will be used. Note that arrays contain brackets, which are usually not valid in identifiers. Defining a simple type alias is the best way around this.

2.2.2 Generating C

A Futhark program `futlib.fut` can be compiled to reusable C code using either:

```
$ futhark-c --library futlib.fut
```

Or:

```
$ futhark-c --library futlib.fut
```

This produces two files in the current directory: `futlib.c` and `futlib.h`. If we wish (and are on a Unix system), we can then compile `futlib.c` to a shared library like this:

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

However, details of how to link the generated code with other C code is highly system-dependent, and outside the scope of this manual.

The generated header file (here, `futlib.h`) specifies the API, and is intended to be human-readable. The basic usages revolves around creating a *configuration object*, which can then be used to obtain a *context object*, which must be passed whenever entry points are call.ed

The configuration object is created using the following function:

```
struct futhark_context_config *futhark_context_config_new();
```

Depending on the backend, various functions are generated to modify the configuration. The following is always available:

```
void futhark_context_config_set_debugging(struct futhark_context_config *cfg,
                                          int flag);
```

A configuration object can be used to create a context with the following function:

```
struct futhark_context *futhark_context_new(struct futhark_context_config *cfg);
```

Memory management is entirely manual. Deallocation functions are provided for all types defined in the header file. Everything returned by an entry point must be manually deallocated.

Functions that can fail return an integer: 0 on success and a non-zero value on error. A human-readable string describing the error can be retrieved with the following function:

```
char *futhark_context_get_error(struct futhark_context *ctx);
```

It is the callers responsibility to `free()` the returned string. Any subsequent call to the function returns `NULL`, until a new error occurs.

For now, many internal errors, such as failure to allocate memory, will cause the function to `abort()` rather than return an error code. However, all application errors (such as bounds and array size checks) will produce an error code.

The API functions are thread safe.

2.2.3 Generating Python

The `futhark-py` and `futhark-pyopencl` compilers both support generating reusable Python code, although the latter of these generates code of sufficient performance to be worthwhile. The following mentions options and parameters only available for `futhark-pyopencl`. You will need at least PyOpenCL version 2015.2.

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. The methods take one parameter for each parameter in the corresponding entry point, and return a tuple containing a value for every value returned by the entry point. For entry points returning a single (non-tuple) value, just that value is returned (that is, single-element tuples are not returned).

After the class has been instantiated, these methods can be invoked to run the corresponding Futhark function. The constructor for the class takes various keyword parameters:

`interactive=BOOL`

: If `True` (the default is `False`), show a menu of available OpenCL platforms and devices, and use the one chosen by the user.

`platform_pref=STR`

: Use the first platform that contains the given string. Similar to the `-p` option for executables.

`device_pref=STR`

: Use the first device that contains the given string. Similar to the `-d` option for executables.

Futhark arrays are mapped to either the Numpy `ndarray` type or the `` `pyopencl.array <https://documen.tician.de/pyopencl/array.html>_` `` type. Scalars are mapped to Numpy scalar types.

CHAPTER 3

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 (for example by applying rules of operator precedence).

This reference describes only the language itself. Documentation for the basis library is available elsewhere.

3.1 Identifiers and Keywords

```
id          ::=   letter (letter | "_" | "'")* | "_" id
quals       ::=   (id ".")+
qualid      ::=   id | quals id
binop       ::=   opstartchar opchar*
qualbinop   ::=   binop | quals binop | "`" qualid "`"
fieldid     ::=   decimal | id
```

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*. Note that a *fieldid* can be a decimal number. Futhark has three distinct name spaces: terms, module types, and types. Modules (including parametric modules) and values both share the term namespace.

3.2 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, the type of the literal will be inferred based on its use. If the use is not constrained, integral literals will be assigned type `i32`, and decimal literals type `f64`. Hexadecimal literals are supported by prefixing with `0x`, and binary literals by prefixing with `0b`.

Floats can also be written in hexadecimal format such as `0x1.fp3`, instead of the usual decimal notation. Here, `0x1.f` evaluates to `1 15/16` and the `p3` multiplies it by `2^3 = 8`.

```
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
exponentfloat     ::=   (intpart | pointfloat) exponent
hexadecimalfloat  ::=   0 ("x" | "X") hexintpart hexfraction ("p"|"P") ["+" | "-"] decdigit
intpart           ::=   decdigit (decdigit |"_")*
fraction          ::=   "." decdigit (decdigit |"_")*
hexintpart        ::=   hexdigit (hexdigit | "_")*
hexfraction       ::=   "." hexdigit (hexdigit |"_")*
exponent          ::=   ("e" | "E") ["+" | "-"] decdigit+
```

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

3.2.1 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 | function_type | type type_arg
array_type     ::=   "[" [dim] "]" type
tuple_type     ::=   "(" ")" | "(" type ("[" "," type "]")* ")"
record_type    ::=   "{" "}" | "{" fieldid ":" type ("," fieldid ":" type)* "}"
```

```
function_type  ::=  param_type "->" type
param_type     ::=  type | "(" id ":" type ")"
type_arg       ::=  "[" [dim] "]" | type
dim            ::=  qualid | decimal
```

An array value is written as a sequence of zero or more 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 `[]`, and its type is inferred from its use. When writing Futhark values for such uses as `futhark-test` (but not when writing programs), the syntax `empty(t)` can be used to denote an empty array with row type `t`.

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.

Functions are classified via function types, but they are not fully first class. See *Higher-order functions* for the details.

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

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

3.3 Declarations

A Futhark file or module consists of a sequence of declarations. Each declaration is processed in order, and a declaration can only refer to names bound by preceding declarations.

```
dec  ::=  fun_bind | val_bind | type_bind | mod_bind | mod_type_bind
        | "open" mod_exp
        | "import" stringlit
        | "local" dec
```

The `open` declaration brings names defined in another module into scope (see also *Module System*). For the meaning of `import`, see *Referring to Other Files*. If a declaration prefixed with `local`, whatever names it defines will *not* be visible outside the current module. In particular `local open` is used to bring names from another module into scope, without making those names available to users of the module being defined. In most cases, using module type ascription is a better idea.

3.3.1 Declaring Functions and Values

```
fun_bind  ::=   ("let" | "entry") (id | "(" binop ")") 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, and body of the function:

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

Full type inference is not supported, but the return type can be elided. 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.

3.3.2 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.

An infix operator can also be defined with prefix notation, like an ordinary function, by enclosing it in parentheses:

```
let (+) (x: i32) (y: i32) = x - y
```

This is necessary when defining operators that take type or shape parameters.

3.3.3 Entry Points

Apart from declaring a function with the keyword `let`, 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.

3.3.4 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.

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

We use a *shape parameter*, `[n]`, to explicitly quantify the names of shapes. The `[n]` parameter need not be explicitly passed when calling `f`. Rather, its value is implicitly deduced from the arguments passed for the value parameters. Any size parameter must be used in a value parameter. 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
```

3.3.6 Type Abbreviations

```
type_bind   ::=   "type" id type_param* "=" type
type_param  ::=   "[" id "]" | "'" 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 enclosed in square brackets. Example:

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

let x: 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 x: two_vecs [2] i32 = (iota 2, replicate 2 0)
```

A type parameter prefixed with `'^` is a *lifted type parameter*. These may be instantiated with types that may be functions. On the other hand, values of such types are subject to the same restrictions as function types (cannot be put in an arrays, returned from `if`, or used as a `loop` parameter; see *Higher-order functions*).

When using uniqueness attributes with type abbreviations, inner uniqueness attributes are overridden 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
```

3.4 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 ("." fieldid)*
              | stringlit
              | "(" ")"
              | "(" exp ")" ("." fieldid)*
              | "(" exp ("," exp)* ")"
              | "{" "}"
              | "{" field ("," field)* "}"
              | qualid "[" index ("," index)* "]"
              | "(" exp ")" "[" index ("," index)* "]"
              | quals "." "(" exp ")"
              | "[" exp ("," exp)* "]"
              | "[" exp [".." exp] "..." exp "]"
              | "(" qualbinop ")"
              | "(" exp qualbinop ")"
              | "(" qualbinop exp ")"
              | "(" ( "." field )+ ")"
              | "(" "." "[" index ("," index)* "]" ")"
exp       ::=   atom
              | exp qualbinop exp
              | exp exp
```

```
                | exp ":" type
                | exp [ ".." exp ] "..." exp
                | exp [ ".." exp ] "..<" exp
                | exp [ ".." exp ] "..>" exp
                | "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
                | "(" "\" type_param* pat+ [":" type] "->" exp ")"
                | "loop" type_param* pat [("=" exp)] loopform "do" exp
                | "unsafe" exp
                | "assert" atom atom
                | exp "with" "[" index ("," index)* "]" "<-" exp
field     ::=   fieldid "=" exp
                | id
pat       ::=   id
                | "_"
                | "(" ")"
                | "(" pat ")"
                | "(" pat ("," pat)+ ")"
                | "{" "}"
                | "{" fieldid ["=" pat] ["," fieldid ["=" pat]] "}"
                | pat ":" type
loopform  ::=   "for" id "<" exp
                | "for" pat "in" exp
                | "while" exp
index     ::=   exp [":" [exp]] [":" [exp]]
                | [exp] ":" exp [":" [exp]]
                | [exp] [":" exp] ":" [exp]
nat_int   ::=   decdigit+
```

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.

3.4.1 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.

- An expression `x.y` may either be a reference to the name `y` in the module `x`, or the field `y` in the record `x`. Modules and values occupy the same name space, so this is disambiguated by the type of `x`.
- A type ascription (`exp : type`) cannot appear as an array index, as it conflicts 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.
- An expression `(-x)` is parsed as the variable `x` negated and enclosed in parentheses, rather than an operator section partially applying the infix operator `-`.

- 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	`* / % // %%`
left	`|>`
right	`<|`
right	`->`
left	juxtaposition

3.4.2 Semantics of Simple Expressions

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.

`()`

Evaluates to an empty tuple.

`( e )`

Evaluates to the result of `e`.

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

Evaluates to a tuple containing `N` values. Equivalent to the record literal `{1=e1, 2=e2, ..., N=eN}`.

{f1, f2, ..., fN}

A record expression consists of a comma-separated sequence of *field expressions*. Each field expression defines the value of a field in the record. A field expression can take one of two forms:

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

`f`: defines a field with the name `f` and the value of the variable `f` in scope.

Each field may only be defined once.

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 former 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.

x..y...z

Construct an integer array whose first element is `x` and which proceeds stride of `y-x` until reaching `z` (inclusive). The `..y` part can be elided in which case a stride of 1 is used. The stride may not be zero. An empty array is returned in cases where `z` would never be reached or `x` and `y` are the same value.

x..y..<z

Construct an integer array whose first elements is `x`, and which proceeds upwards with a stride of `y` until reaching `z` (exclusive). The `..y` part can be elided in which case a stride of 1 is used. An empty array is returned in cases where `z` would never be reached or `x` and `y` are the same value.

`x..y..>z`

Construct an integer array whose first elements is `x`, and which proceeds downwards with a stride of `y` until reaching `z` (exclusive). The `..y` part can be elided in which case a stride of -1 is used. An empty array is returned in cases where `z` would never be reached or `x` and `y` are the same value.

`e.f`

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

`m.(e)`

Evaluate the expression `e` with the module `m` locally opened, as if by `open`. This can make some expressions easier to read and write, without polluting the global scope with a declaration-level `open`.

`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

Apply the function `f` to the argument `x`.

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.

unsafe e

Elide safety checks and assertions (such as bounds checking) that occur during execution of `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 there. Make very sure that the code is correct; eliding such checks can lead to memory corruption.

assert cond e

Terminate execution with an error if `cond` evaluates to false, otherwise produce the result of evaluating `e`. Make sure that `e` produces a meaningful value that is used subsequently (it can just be a variable), or dead code elimination can easily remove the assertion.

a with [i] <- e

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

if c then a else b

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

3.4.3 Binding Expressions

`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` 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 x in a do loopbody`

1. Bind `pat` to the initial values given in `initial`.
2. For each element `x` in `a`, evaluate `loopbody` and rebind `pat` to the result of the evaluation.
3. Return the final value of `pat`.

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 for x < n do loopbody`

Equivalent to `loop (pat = initial) for x in [0..1..<n] do loopbody`.

`loop pat = initial = while cond do loopbody`

1. Bind `pat` to the initial values given in `initial`.
2. If `cond` evaluates to true, bind `pat` to the result of evaluating `loopbody`, and repeat the step.
3. Return the final value of `pat`.

See also *Shape Declarations*.

3.4.4 Function Expressions

`\x y z:  t -> e`

Produces an anonymous function taking parameters `x`, `y`, and `z`, returns type `t`, and whose body is `e`.

`(*binop*)`

An *operator section* that is equivalent to `\x y -> x *binop* y`.

`(x *binop*)`

An *operator section* that is equivalent to `\y -> x *binop* y`.

`(*binop* y)`

An *operator section* that is equivalent to `\x -> x *binop* y`.

`(.a.b.c)`

An *operator section* that is equivalent to `\x -> a.b.c`.

`(.[i,j])`

An *operator section* that is equivalent to `\x -> x[i,j]`.

3.5 Higher-order functions

At a high level, Futhark functions are values, and can be used as any other value. However, to ensure that the compiler is able to compile the higher-order functions efficiently via *defunctionalisation*, certain type-driven restrictions exist on how functions can be used. These also apply to any record or tuple containing a function (a *functional type*):.

- Arrays of functions are not permitted.
- A function cannot be returned from an *if* expression.
- A loop parameter cannot be a function.

Further, *type parameters* are divided into *non-lifted* (bound with an apostrophe, e.g. `'t`), and *lifted* (`'^t`). Only lifted type parameters may be instantiated with a functional type. Within a function, a lifted type parameter is treated as a functional type. All abstract types declared in modules (see *Module System*) are considered non-lifted, and may not be functional.

See also *In-place updates* for details on how uniqueness types interact with higher-order functions.

3.6 Type Inference

Futhark supports Hindley-Milner-style type inference, so in many cases explicit type annotations can be left off. Record field projection cannot in isolation be fully inferred, and may need type annotations where their inputs are bound. Further, unique types (see *In-place updates*) must be explicitly annotated.

3.7 In-place updates

In-place updates do not provide observable side effects, but they do provide a way to efficiently update an array in-place, with the guarantee that the cost is proportional to the size of the value(s) being written, not the size of the full array.

The `a with [i] <- v` language construct, and derived forms, performs an in-place update. The compiler verifies that the original array (`a`) is not used on any execution path following the in-place update. This involves also checking that no *alias* of `a` is used. Generally, most language constructs produce new arrays, but some (slicing) create arrays that alias their input arrays.

When defining a function parameter or return type, we can mark it as *unique* by prefixing it with an asterisk. For example:

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

For bulk in-place updates with multiple values, use the `scatter` function in the basis library. In the parameter declaration `a: *[i32]`, the asterisk means that the function `modify` has been given "ownership" of the array `a`, meaning that any caller of `modify` will never reference array `a` after the call again. This allows the `with` expression to perform an in-place update.

After a call `modify a i x`, neither `a` or any variable that *aliases* `a` may be used on any following execution path.

Uniqueness typing generally interacts poorly with higher-order functions. The issue is that we cannot control how many times a function argument is applied, or to what, so it is not safe to pass a function that consumes its argument. The following two conservative rules govern the interaction between uniqueness types and higher-order functions:

1. In the expression `let p = e1 in ...`, if *any* in-place update takes place in the expression `e1`, the value bound by `p` must not be or contain a function.
2. A function that consumes one of its arguments may not be passed as a higher-order argument to another function.

3.8 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 and module types. *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. Module names exist in the same name space as values, but module types are their own name space.

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
}
```

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`. As a derived form, we can write `module M p = e` to mean `module M = \p -> e`.

3.8.1 Module Expressions

```
mod_exp  ::=   qualid
             | mod_exp ":" mod_type_exp
             | "\" "(" id ":" mod_type_exp ")" [":" mod_type_exp] "->" mod_exp
             | mod_exp mod_exp
             | "(" mod_exp ")"
             | "{" dec* "}"
             | "import" stringlit
```

A module expression produces a module. Modules are collections of bindings produced by declarations (*dec*). In particular, a module may contain other modules or module types.

qualid

Evaluates to the module of the given name.

(mod_exp)

Evaluates to `mod_exp`.

mod_exp : mod_type_exp

Module ascription evaluates the module expression and the module type expression, verifies that the module implements the module type, then returns a module that exposes only the functionality described by the module type. This is how internal details of a module can be hidden.

\(p: mt1): mt2 -> e

Constructs a *parametric module* (a function at the module level) that accepts a parameter of module type `mt1` and returns a module of type `mt2`. The latter is optional, but the parameter type is not.

e1 e2

Apply the parametric module `m1` to the module `m2`.

{ decs }

Returns a module that contains the given definitions. The resulting module defines any name defined by any declaration that is not `local`, *in particular* including names made available via `open`.

import "foo"

Returns a module that contains the definitions of the file `"foo"` relative to the current file. See *Referring to Other Files*.

3.8.2 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
```

Module types classify modules, with the only (unimportant) difference in expressivity being that modules can contain module types, but module types cannot specify that a module must contain a specific module types. They can specify of course that a module contains a *submodule* of a specific module type.

3.9 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:

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

The above will include the file `path/to/a/file.fut` relative to the including file. When importing a nonlocal file (such as the basis library), 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"
```

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

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

CHAPTER 4

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.

4.1 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.

4.2 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.

4.3 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.

4.4 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): u32 =
  let b = 0x10001u32
  let u = 0i32
```

(continues on next page)

(continued from previous page)

```
  let v = 1i32 in
  let (_,_,u,_) = loop ((a,b,u,v)) while a > 0u16 do
    let q = b / u32.u16(a)
    let r = b % u32.u16(a)

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

    let t = v
    let v = u - i32.u32 (q) * v
    let u = t in
    (a,b,u,v)

  in u32.i32(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.

4.5 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 5

Futhark Compared to Other Functional Languages

This guide is intended to quickly get programmers who are familiar with other functional languages acquainted with Futhark.

Futhark is a simple language with a complex compiler. Functional programming is fundamentally well suited to data-parallelism, so Futhark's syntax and underlying concepts are taken directly from established functional languages; mostly from Haskell and the members of the ML familiy. While Futhark does add a few small conveniences (built-in array types) and one complicated and unusual feature (in-place updates via uniqueness types, see *In-place updates*), a programmer familiar with a common functional language should be able to easily understand the meaning of a Futhark program, and quickly start writing their own programs. To speed up this process, we describe, in the following, some of the various quirks and unexpected limitations imposed by Futhark. It is recommended to read some of the example programs along with this guide. This guide does *not* cover all Futhark features worth knowing, so do also skim *Language Reference*.

5.1 Basic Syntax

Futhark uses a keyword-based structure, with optional indentation *solely* for human readability. This aspect differs from Haskell and F#.

Names are lexically divided into *identifiers* and *symbols*:

- *Identifiers* begin with a letter or underscore and contain letters, numbers, underscores, and apostrophes.
- *Symbols* contain the characters found in the default operators (`+-*/%=!><|&^`)

All function and variable names must be identifiers, and all infix operators are symbols. An identifier can be used as an infix operator by enclosing it in backticks, as in Haskell.

Identifiers are case-sensitive, and there is no restriction on the case of the first letter (unlike Haskell and OCaml, but like Standard ML).

User-defined operators are possible, but the fixity of the operator depends on its name. Specifically, the fixity of a user-defined operator *op* is equal to the fixity of the built-in operator that is the longest prefix of *op*. So, for example, `<<=` would have the same fixity as `<<`, and `=<<` the same as `=`. This rule is the same as the rule found in OCaml and F#.

Top-level functions and values are defined with `let`, as in OCaml and F#.

5.2 Evaluation

Futhark is a completely pure language, with no cheating through monads or anything of the sort.

Evaluation is *eager* or *call-by-value*, like most non-Haskell languages. However, there is no defined evaluation order. Furthermore, the Futhark compiler is permitted to turn non-terminating programs into terminating programs, for example by removing dead code that might cause an error. Moreover, there is no way to handle errors within a Futhark program (no exceptions or similar); although errors are gracefully reported to whatever invokes the Futhark program.

The evaluation semantics are entirely sequential, with parallelism being solely an operational detail. Hence, race conditions are impossible. However, the Futhark compiler does not automatically go looking for parallelism. Only certain special constructs and built-in library functions (in particular `map`, `reduce`, `scan`, and `filter`) may be executed in parallel.

Currying and partial application works as usual (although functions are not fully first class; see *Types*). Some Futhark language constructs look like functions, but are not (yet). This means they cannot be partially applied, and may not interact well with type inference. These include `unzip` and `zip`. Usually there are more well-behaved wrappers to be found in the basis library.

Lambda terms are written as `\x -> x + 2`, as in Haskell.

A Futhark program is read top-down, and all functions must be declared in the order they are used, similarly to Standard ML. Unlike just about all functional languages, recursive functions are *not* supported. Most of the time, you will use bulk array operations instead, but there is also a dedicated `loop` language construct, which is essentially syntactic sugar for tail recursive functions.

5.3 Types

Futhark supports a range of integer types, floating point types, and booleans (see *Primitive Types and Values*). A numeric literal can be suffixed with its desired type, such as `1i8` for an eight-bit signed integer. Un-adorned numerals have their type inferred based on use. This only works for built-in numeric types.

All types can be combined in tuples as usual, as well as in *structurally typed records*, as in Standard ML. There are not yet any sum types. Abstract types are possible via the module system; see *Module System*.

If a variable `foo` is a record of type `{a: i32, b: bool}`, then we access field `a` with dot notation: `foo.a`. Tuples are a special case of records, where all the fields have a 1-indexed numeric label. For example, `(i32, bool)` is the same as `{1: i32, 2: bool}`, and can be indexed as `foo.1`.

Arrays are a built-in type. The type of an array containing elements of type `t` is written `[]t` (not `[t]` as in Haskell), and we may optionally annotate it with a size as `[n]t` (see *Shape Declarations*). Array values are written as `[1, 2,3]`. Array indexing is written `a[i]` with *no* space allowed between the array name and the brace. Indexing of multi-dimensional arrays is written `a[i,j]`. Arrays are 0-indexed.

Function types are supported with the usual `a -> b`, and functions can be passed as arguments to other functions. However, there are some restrictions:

- A function cannot be put in an array (but a record or tuple is fine).
- A function cannot be returned from a branch.
- A function cannot be used as a `loop` parameter.

Function types interact with type parameters in a subtle way:

```
let id 't (x: t) = x
```

This declaration defines a function `id` that has a type parameter `t`. Here, `t` is an *unlifted* type parameter, which is guaranteed never to be a function type, and so in the body of the function we could choose to put parameter values of type `t` in an array. However, it means that this identity function cannot be called on a functional value. Instead, we probably want a *lifted* type parameter:

```
let id '^t (x: t) = x
```

Such *lifted* type parameters are restricted from being instantiated with function types.

Futhark supports Hindley-Milner type inference (with some restrictions), so we could also just write it as:

```
let id x = x
```

Type appreviations are possible:

```
type foo = (i32, i32)
```

Type parameters are supported as well:

```
type pair 'a 'b = (a, b)
```

As with everything else, they are structurally typed, so the types `pair i32 bool` and `(i32, bool)` are entirely interchangeable.

Size parameters can also be passed:

```
type vector [n] t = [n]t
type i32matrix [n][m] = [n] (vector [m] i32)
```

Note that for an actual array type, the dimensions come *before* the element type, but with a type abbreviation, a size is just another parameter. This easily becomes hard to read if you are not careful.

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.
- You may with to set the environment variable `FUTHARK_COMPILER_DEBUGGING=1`. Currently this only has the effect of making the frontend print internal names, but it may control more things in the future.

6.1 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.

6.2 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 `-D` to have the program print information on allocation and deallocation of memory.
- (*futhark-opencl* only) Use the `-D` 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.

6.3 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 programs generated by `futhark-c`/`futhark-opencl`, and `futhark-py`/`futhark-pyopencl`. It is *not* supported for `futharki`.

You can generate random data in the binary format with *futhark-dataset*. This tool can also be used to convert between binary and textual data.

Futhark-generated executables can be asked to generate binary output with the `-b` option.

7.1 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 2), `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 treating it as a 0-dimensional array:

```
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>
```

7.1.1 Type Values

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

```
"  i8"
" i16"
" i32"
" i64"
"  u8"
" u16"
" u32"
" u64"
" f32"
" f64"
"bool"
```

Note that unsigned and signed integers have the same byte-level representation.

CHAPTER 8

futhark

8.1 SYNOPSIS

futhark [options...] infile

8.2 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
```

8.3 SEE ALSO

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

CHAPTER 9

futhark-c

9.1 SYNOPSIS

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

9.2 DESCRIPTION

`futhark-c` translates a Futhark program to sequential C code, and either compiles that C code with gcc(1) to an executable binary program, or produces a `.h` and `.c` file that can be linked with other code.. 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 entry point (`main` by default) from standard input and print its return value on standard output. The arguments are read and printed in Futhark syntax, just like futharki(1).

9.3 OPTIONS

-o outfile Where to write the result. If the source program is named 'foo.fut', this defaults to 'foo'.

--library Generate a library instead of an executable. Appends `.c`/`.h` to the name indicated by the `-o` option to determine output file names.

--Werror Treat warnings as errors.

--safe Ignore `unsafe` in program and perform safety checks unconditionally.

-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.

9.4 SEE ALSO

futharki(1), futhark-test(1)

CHAPTER 10

futhark-opencl

10.1 SYNOPSIS

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

10.2 DESCRIPTION

`futhark-opencl` translates a Futhark program to C code invoking OpenCL kernels, and either compiles that C code with gcc(1) to an executable binary program, or produces a `.h` and `.c` file that can be linked with other code. The standard Futhark optimisation pipeline is used, and GCC is invoked with `-O3`, `-lm`, and `-std=c99`. The resulting program will otherwise behave exactly as one compiled with `futhark-c`.

10.3 OPTIONS

-o outfile	Where to write the result. If the source program is named 'foo.fut', this defaults to 'foo'.
--library	Generate a library instead of an executable. Appends `.c`/`.h` to the name indicated by the `-o` option to determine output file names.
--Werror	Treat warnings as errors.
--safe	Ignore `unsafe` in program and perform safety checks unconditionally.
-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.

10.4 SEE ALSO

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

CHAPTER 11

futhark-py

11.1 SYNOPSIS

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

11.2 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, likely too much to be useful. It is more interesting to use this command's big brother, `futhark-pyopencl`.

11.3 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.

--Werror Treat warnings as errors.

--safe Ignore `unsafe` in program and perform safety checks unconditionally.

-h	Print help text to standard output and exit.
-V	Print version information on standard output and exit.

11.4 SEE ALSO

futhark-pyopencl(1)

CHAPTER 12

futhark-pyopencl

12.1 SYNOPSIS

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

12.2 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.

The generated code requires at least PyOpenCL version 2015.2.

12.3 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.
--Werror	Treat warnings as errors.
--safe	Ignore `unsafe` in program and perform safety checks unconditionally.
-h	Print help text to standard output and exit.
-V	Print version information on standard output and exit.

12.4 SEE ALSO

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

CHAPTER 13

futharki

13.1 SYNOPSIS

futharki [infile]

13.2 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.

13.3 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.

13.4 BUGS

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

13.5 SEE ALSO

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

CHAPTER 14

futhark-test

14.1 SYNOPSIS

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

14.2 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.

14.3 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
```

14.4 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)
```

14.5 SEE ALSO

futhark-c(1), futharki(1)

CHAPTER 15

futhark-bench

15.1 SYNOPSIS

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

15.2 DESCRIPTION

This tool is the recommended way to benchmark Futhark programs. Programs are compiled using the specified compiler (`futhark-c` by default), then run a number of times for each test case, 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.

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

15.3 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.

--skip-compilation Do not run the compiler, and instead assume that each benchmark program has already been compiled. Use with caution.

--exclude-case=TAG Do not run test cases that contain the given tag. Cases marked with "nobench" or "disable" are ignored by default.

15.4 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)
```

15.5 SEE ALSO

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

CHAPTER 16

futhark-doc

16.1 SYNOPSIS

futhark-doc [-o outdir] dir

16.2 DESCRIPTION

`futhark-doc` generates HTML-formatted documentation from Futhark code. One HTML file will be created for each `.fut` file in the given directory, as well as any file reachable through `import` expressions. The given Futhark code will be considered as one cohesive whole, and must be type-correct.

Futhark definitions may be documented by prefixing them with a block of line comments starting with `-- |` (see example below). Simple Markdown syntax is supported within these comments. A link to another identifier is possible with the notation `` `name`@namespace ``, where `namespace` must be either `term`, `type`, or `mtype` (module names are in the `term` namespace). A file may contain a leading documentation comment, which will be considered the file *abstract*.

16.3 OPTIONS

-o outdir	The name of the directory that will contain the generated documentation. This option is mandatory.
-v, --verbose	Print status messages to stderr while running.
-h	Print help text to standard output and exit.
-V	Print version information on standard output and exit.

16.4 EXAMPLES

```
-- | Gratuitous re-implementation of `map`@term.
--
-- Does exactly the same.
let mymap = ...
```

16.5 SEE ALSO

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

CHAPTER 17

futhark-dataset

17.1 SYNOPSIS

futhark-dataset options...

17.2 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.

If no `-g`/`--generate` options are passed, values are read from standard input, and printed to standard output in the indicated format. The input format (whether textual or binary) is automatically detected.

17.3 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).

-t, --type Output the types of values (textually) instead of the values themselves. Mostly useful when reading values on stdin.

17.4 EXAMPLES

Generate a 4 by 2 integer matrix:

```
futhark-dataset -g [4][2]i32
```

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
```

Create a binary data file from a data file:

```
futhark-dataset --binary < any_data > binary_data
```

Determine the types of values contained in a data file:

```
futhark-dataset -t < any_data
```

17.5 SEE ALSO

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