Futhark Documentation

Release 2.4

name:

DIKU

Apr 23, 2020

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 main ways to install the Futhark compiler: using a precompiled tarball or compiling from source. Both methods are discussed below. If you are using Linux, see *Installing Futhark on Linux*. If you are using Windows, make sure to read *Setting up Futhark on Windows*. If you are using macOS, read *Using OpenCL or CUDA*.

Futhark is also available via Nix. If you are using Nix, simply install the `futhark` derivation from Nixpkgs.

1.1 Dependencies

The Linux binaries we distribute are statically linked and should not require any special libraries installed system-wide.

When building from source on Linux and macOS, you will need to have the `gmp` and `tinfo` libraries installed. These are pretty common, so you may already have them. On Debian-like systems (e.g. Ubuntu), use:

```
sudo apt install libtinfo-dev libgmp-dev
```

If you install Futhark via a package manager (e.g. Homebrew, Nix, or AUR), you shouldn't need to worry about any of this.

Actually *running* the output of the Futhark compiler may require additional dependencies, for example an OpenCL library and GPU driver. See the documentation for the respective compiler backends.

1.2 Compiling from source

The recommended way to compile Futhark is with the Haskell Tool Stack, which handles dependencies and compilation of the Futhark compiler. You will therefore 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. This step typically requires at least 8GiB of memory. You may be able to build it on a smaller machine by adding the `--fast` option, although the resulting Futhark compiler binary will run slower.

After building, you can copy the binaries to your `$HOME/.local/bin` directory by running:

```
$ stack install
```

Note that this does not install the Futhark manual pages.

1.2.1 Compiling with `cabal`

You can also compile Futhark with `cabal`. If so, you must install an appropriate version of GHC (usually the newest) and `cabal` yourself, for example through your favourite package manager. On Linux, you can always use ghcup. Then clone the repository as listed above and run:

```
$ cabal update
$ cabal build
```

To install the Futhark binaries to a specific location, for example `$HOME/.local/bin`, run:

```
$ cabal install --install-method=copy  --overwrite-policy=always --installdir=$HOME/.
↪local/bin/
```

1.3 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). See the enclosed `README.md` for installation instructions.

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.

1.4 Installing Futhark on Linux

- Linuxbrew is a distribution-agnostic package manager that contains a formula for Futhark. If Linuxbrew is installed (which does not require `root` access), installation is as easy as:

  ```
  $ brew install futhark
  ```

  Note that as of this writing, Linuxbrew is hampered by limited compute resources for building packages, so the Futhark version may be a bit behind.

- Arch Linux users can use a futhark-nightly package or a regular futhark package.
- NixOS users can install the `futhark` derivation.

Otherwise (or if the version in the package system is too old), your best bet is to install from source or use a tarball, as described above.

1.4.1 Using OpenCL or CUDA

If you wish to use `futhark opencl` or `futhark cuda`, you must have the OpenCL or CUDA libraries installed, respectively. Consult your favourite search engine for instructions on how to do this on your distribution. It is usually not terribly difficult if you already have working GPU drivers.

For OpenCL, note that there is a distinction between the general OpenCL host library (`OpenCL.so`) that Futhark links against, and the *Installable Client Driver* (ICD) that OpenCL uses to actually talk to the hardware. You will need both. Working display drivers for the GPU does not imply that an ICD has been installed - they are usually in a separate package. Consult your favourite search engine for details.

1.5 Installing Futhark on macOS

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

```
$ brew install futhark
```

Or you can install the unreleased development version with:

```
$ brew install --HEAD futhark
```

This has to compile from source, so it takes a little while (20-30 minutes is common).

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. On a Mac with an AMD GPU, this is done by passing `-dAMD` to the generated Futhark executable.

1.6 Setting up Futhark on Windows

The Futhark compiler itself is easily installed on Windows via `stack` (see above). If you are using the default Windows console, you may need to run `chcp 65001` to make Unicode characters show up correctly.

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.

1.6.1 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 with `futhark opencl` and run Futhark programs on the GPU.

Congratulations!

1.6.2 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']
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 Game of Life example.

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 subcommand of the `futhark` binary. For example, `futhark c` compiles a Futhark program by translating it to sequential C code, while `futhark pyopencl` generates Python code with calls to 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, accessible as `futhark run` and `futhark repl`. The latter is an interactive prompt, useful for experimenting with Futhark expressions. Be aware that the interpreter runs code very slowly.

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 setup or teardown, including reading the input or writing the result, is not included in the measurement. See the documentation for 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.

Parallel Options

The following options are supported by executables generated with the parallel backends (`opencl`, `pyopencl`, `csopencl`, and `cuda`).

`--tuning=FILE`

Load tuning options from the indicated *tuning file*. The file must contain lines of the form `SIZE=VALUE`, where each *SIZE* must be one of the sizes listed by the `--print-sizes` option (without size class), and the *VALUE* must be a non-negative integer. Extraneous spaces or blank lines are not allowed. A zero means to use the default size, whatever it may be. In case of duplicate assignments to the same size, the last one takes predecence. This is equivalent to passing each size setting on the command like using the `--size` option, but more convenient.

`--print-sizes`

Print a list of tunable sizes followed by their *size class* in parentheses, which indicates what they are used for.

`--size=SIZE=VALUE`

Set one of the tunable sizes to the given value. Using the `--tuning` option is more convenient.

OpenCL-specific Options

The following options are supported by executables generated with the OpenCL backends (`opencl`, `pyopencl`, and `csopencl`):

`-P`

Measure the time taken by various OpenCL operations (such as kernels) and print a summary at the end. Unfortunately, it is currently nontrivial (and manual) to relate these operations back to source Futhark code.

`-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.

`--default-group-size INT`

The default size of OpenCL workgroups that are launched. Capped to the hardware limit if necessary.

`--default-num-groups INT`

The default number of OpenCL workgroups that are launched.

`--dump-opencl FILE`

Don't run the program, but instead 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).

`--dump-opencl-binary FILE`

Don't run the program, but instead dump the compiled version of the embedded OpenCL program to the indicated file. On NVIDIA platforms, this will be PTX code. If this option is set, no entry point will be run.

`--load-opencl-binary FILE`

Load an OpenCL binary from the indicated file.

`--build-option OPT`

Add an additional build option to the string passed to `clBuildProgram()`. Refer to the OpenCL documentation for which options are supported. Be careful - some options can easily result in invalid results.

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.

CUDA-specific Options

The following options are supported by executables generated by the `cuda` backend:

`--dump-cuda FILE`

Don't run the program, but instead dump the embedded CUDA program to the indicated file. Useful if you want to see what is actually being executed.

`--load-cuda FILE`

Instead of using the embedded CUDA 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-cuda` (perhaps lightly modified).

`--dump-ptx FILE`

As `--dump-cuda`, but dumps the compiled PTX code instead.

`--load-ptx FILE`

Instead of using the embedded CUDA program, load compiled PTX code from the indicated file.

`--nvrtc-option=OPT`

Add the given option to the command line used to compile CUDA kernels with NVRTC. The list of supported options varies with the CUDA version but can be found in the NVRTC documentation.

For convenience, CUDA executables also accept the same `--default-num-groups` and `--default-group-size` options that the OpenCL backend uses. These then refer to grid size and thread block size, respectively.

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. Five of the Futhark compilers support this: `futhark c`, `futhark opencl`, `futhark cuda`, `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, for example, are a common issue. 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, if you insert an explicit type annotation (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 opencl --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 an object file like this:

```
$ gcc futlib.c -c
```

This produces a file `futlib.o` that can then be linked with the main application. Details of how to link the generated code with other C code is highly system-dependent, and outside the scope of this manual. On Unix, we can simply add `futlib.o` to the final compiler or linker command line:

```
$ gcc main.c -o main futlib.o
```

Depending on the Futhark backend you are using, you may need to add some linker flags. For example, `futhark opencl` requires `-lOpenCL` (`-framework OpenCL` on macOS). See the manual page for each compiler for details.

It is also possible to simply add the generated `.c` file to the C compiler command line used for compiling our whole program (here `main.c`):

```
$ gcc main.c -o main futlib.c
```

The downside of this approach is that the generated `.c` file may contain code that causes the C compiler to warn (for example, unused support code that is not needed by the Futhark program).

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

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);
```

Context creation may fail. Immediately after `futhark_context_new()`, call `futhark_context_get_error()` (see below), which will return a non-NULL error string if context creation failed.

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 caller's 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.

C with OpenCL

When generating C code with `futhark opencl`, you will need to link against the OpenCL library when linking the final binary:

```
$ gcc main.c -o main futlib.o -lOpenCL
```

When using the OpenCL backend, extra API functions are provided for directly accessing or providing the OpenCL objects used by Futhark. Take care when using these functions. In particular, a Futhark context can now be provided with the command queue to use:

```
struct futhark_context *futhark_context_new_with_command_queue(struct futhark_context_
↪config *cfg, cl_command_queue queue);
```

As a `cl_command_queue` specifies an OpenCL device, this is also how manual platform and device selection is possible. A function is also provided for retrieving the command queue used by some Futhark context:

```
cl_command_queue futhark_context_get_command_queue(struct futhark_context *ctx);
```

This can be used to connect two separate Futhark contexts that have been loaded dynamically.

The raw `cl_mem` object underlying a Futhark array can be accessed with the function named `futhark_values_raw_type`, where `type` depends on the array in question. For example:

```
cl_mem futhark_values_raw_i32_1d(struct futhark_context *ctx, struct futhark_i32_1d␣
↪*arr);
```

The array will be stored in row-major form in the returned memory object. The function performs no copying, so the `cl_mem` still belongs to Futhark, and may be reused for other purposes when the corresponding array is freed. A dual function can be used to construct a Futhark array from a `cl_mem`:

```
struct futhark_i32_1d *futhark_new_raw_i32_1d(struct futhark_context *ctx,
                                              cl_mem data,
                                              int offset,
                                              int dim0);
```

This function *does* copy the provided memory into fresh internally allocated memory. The array is assumed to be stored in row-major form `offset` bytes into the memory region.

See also *futhark-opencl*.

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 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
opstartchar  ::=  "+" | "-" | "*" | "/" | "%" | "=" | "!" | ">" | "<" | "|" | "&" | "^"
opchar       ::=  opstartchar | "."
constructor  ::=  "#" 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 constructor names of a sum type (*Compound Types and Values*) are identifiers prefixed with `#`, with no space afterwards. 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  ::=  "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

```
type  ::=  qualid
         | array_type
         | tuple_type
         | record_type
         | sum_type
         | function_type
```

```
              | type_application
```

Compound types can be constructed based on the primitive types. The Futhark type system is entirely structural, and type abbreviations are merely shorthands. The only exception is abstract types whose definition has been hidden via the module system (see *Module System*).

```
tuple_type  ::=   "(" ")" | "(" type ("[" "," type "]")* ")"
```

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. Empty tuples are written `()` and are of type `()`.

```
array_type  ::=   "[" [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 or variable indicating the size. We can often elide `d` and write just `[]` (an *anonymous size*), 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), empty arrays are written `empty([0]t)` for an empty array of type `[0]t`. When using `empty`, all dimensions must be given a size, and at least one must be zero, e.g. `empty([2][0]i32)`.

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.

```
sum_type  ::=   constructor type* ("|" constructor type*)*
```

Sum types are anonymous in Futhark, and are written as the constructors separated by vertical bars. Each constructor consists of a `#`-prefixed *name*, followed by zero or more types, called its *payload*. **Note:** The current implementation of sum types is fairly inefficient, in that all possible constructors of a sum-typed value will be resident in memory. Avoid using sum types where multiple constructors have large payloads.

```
record_type  ::=   "{" "}" | "{" fieldid ":" type ("," fieldid ":" type)* "}"
```

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 starting from zero, and vice versa. It is an error for a record type to name the same field twice.

```
type_application  ::=   type type_arg | "*" type
type_arg          ::=   "[" [dim] "]" | type
```

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.

```
function_type  ::=   param_type "->" type
param_type     ::=   type | "(" id ":" type ")"
```

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 UTF-8 encoded arrays of `u8` 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 is 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
```

Hindley-Milner-style type inference is supported. A parameter may be given a type with the notation `(name: type)`. Functions may not be recursive. You may put *size annotations* in the return type and parameter types; see *Size Types*. 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]
```

Type parameters for a function do not need to cover the types of all parameters. The type checker will add more if necessary. For example, the following is well typed:

```
let pair 'a (x: a) y = (x, y)
```

A new type variable will be invented for the parameter `y`.

Shape and type parameters are not passed explicitly when calling function, but are automatically derived. If an array value *v* is passed for a type parameter *t*, all other arguments passed of type *t* must have the same shape as *v*. For example, consider the following definition:

```
let pair 't (x: t) (y: t) = (x, y)
```

The application `pair [1] [2,3]` will fail at run-time.

To simplify the handling of in-place updates (see *In-place Updates*), the value returned by a function may not alias any global variables.

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)
```

We can also define operators by enclosing the operator name in parentheses and suffixing the parameters, as an ordinary function:

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

This is necessary when defining a polymorphic operator.

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 ordinary functions, except for being infix.

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 top-level 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 top-level 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. A constant value may not have a unique type (see *In-place updates*). If the return type contains any anonymous sizes (see *Size types*), new existential sizes will be constructed for them.

3.3.5 Type Abbreviations

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

Type abbreviations function as shorthands for the 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 distinct types: the types `t1` and `t2` are entirely interchangeable.

If the right-hand side of a type contains anonymous sizes, it must be declared "size-lifted" with `type~`. If it (potentially) contains a function, it must be declared "fully lifted" with `type^`. A lifted type can also contain anonymous sizes. Lifted types cannot be put in arrays. Fully lifted types cannot be returned from conditional or loop expressions.

A type abbreviation can have zero or more parameters. A type parameter enclosed with square brackets is a *size 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)
```

Size parameters work much like shape declarations for arrays. Like shape declarations, they can be elided via square brackets containing nothing. All size parameters must be used in the definition of the type abbreviation.

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 *size-lifted type parameter* is prefixed with `'~`, and a *fully lifted type parameter* with `'^`. These have the same rules and restrictions as lifted type abbreviations.

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
           | constructor exp*
           | exp ":" type
           | exp ":>" type
           | exp [ ".." exp ] "..." exp
           | exp [ ".." exp ] "..<" exp
           | exp [ ".." exp ] "..>" exp
           | "if" exp "then" exp "else" exp
           | "let" pat "=" exp "in" exp
           | "let" id "[" index ("," index)* "]" "=" exp "in" exp
           | "let" id type_param* pat+ [":" type] "=" exp "in" exp
           | "(" "\" pat+ [":" type] "->" exp ")"
           | "loop" pat [("=" exp)] loopform "do" exp
           | "unsafe" exp
           | "assert" atom atom
           | exp "with" "[" index ("," index)* "]" "=" exp
           | exp "with" fieldid ("." fieldid)* "=" exp
           | "match" exp ("case" pat "->" exp)+
field ::=   fieldid "=" exp
          | id
pat       ::=   id
          | literal
          | "_"
          | "(" ")"
          | "(" pat ")"
```

```
                | "(" pat ("," pat)+ ")"
                | "{" "}"
                | "{" fieldid ["=" pat] ["," fieldid ["=" pat]] "}"
                | constructor pat*
                | pat ":" type
loopform  ::=   "for" id "<" exp
                | "for" pat "in" exp
                | "while" exp
index     ::=   exp [":" [exp]] [":" [exp]]
                | [exp] ":" exp [":" [exp]]
                | [exp] [":" exp] ":" [exp]
```

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

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 Patterns

We say that a pattern is *irrefutable* if it can never fail to match a value of the appropriate type. Concretely, this means that it does not require any specific sum type constructor (unless the type in question has only a single constructor), or any specific numeric or boolean literal. Patterns used in function parameters and `let` bindings must be irrefutable. Patterns used in `case` need not be irrefutable.

A pattern _ matches any value. A pattern consisting of a literal value (e.g. a numeric constant) matches exactly that value.

3.4.3 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 code points of the 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 `{0=e1, 1=e2, ..., N-1=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 parenthesised expression. Furthermore, 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`.

In the general case, the size of the array produced by a slice is unknown (see *Size types*). In a few cases, the size is known statically:

- `a[0:n]` has size `n`
- `a[:n]` has size `n`
- `a[0:n:1]` has size `n`
- `a[:n:1]` has size `n`

This holds only if `n` is a variable or constant.

[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. A run-time error occurs if `z` is lesser than `x` or `y`, or if `x` and `y` are the same value.

In the general case, the size of the array produced by a range is unknown (see *Size types*). In a few cases, the size is known statically:

- `1..2...n` has size `n`

This holds only if `n` is a variable or constant.

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. A run-time error occurs if `z` is lesser than `x` or `y`, or if `x` and `y` are the same value.

- `0..1..<n` has size `n`
- `0..<n` has size `n`

This holds only if `n` is a variable or constant.

`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. A run-time error occurs if `z` is greater than `x` or `y`, or if `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`.

#c x y z

Apply the sum type constructor `#x` to the payload `x`, `y`, and `z`. A constructor application is always assumed to be saturated, i.e. its entire payload provided. This means that constructors may not be partially applied.

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. However, as an array index must always be of type `i32`, there is never a reason to put an explicit type ascription there.

e :> t

Coerce the size of `e` to `t`. The type of `t` must match the type of `e`, except that the sizes may be statically different. At run-time, it will be verified that the sizes are the same.

! x

Logical negation if `x` is of type `bool`. Bitwise negation if `x` is of integral type.

- x

Numerical negation of `x`, which must be of numeric 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`. Unless `e` produces a value that is used subsequently (it can just be a variable), dead code elimination may 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`.

`r with f = e`

Return the record `r`, but with field `f` changed to have value `e`. The type of the field must remain unchanged. Type inference is limited: `r` must have a *completely known type* up to `f`. This sometimes requires extra type annotations to make the type of `r` known.

`if c then a else b`

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

3.4.4 Binding Expressions

`let pat = e in body`

Evaluate `e` and bind the result to the irrefutable pattern `pat` (see *Patterns*) while evaluating `body`. The `in` keyword is optional if `body` is a `let` expression.

`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. This notation is 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.

`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) = ...`.

`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`.

`match x case p1 -> e1 case p2 -> e2`

Match the value produced by `x` to each of the patterns in turn, picking the first one that succeeds. The result of the corresponding expression is the value of the entire `match` expression. All the expressions associated with a `case` must have the same type (but not necessarily match the type of `x`). It is a type error if there is not a `case` for every possible value of `x` - inexhaustive pattern matching is not allowed.

3.4.5 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`. Lambdas do not permit type parameters; use a named function if you want a polymorphic function.

`(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 -> 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`), *size-lifted* (`'~t`), and *fully lifted* (`'^t`). Only fully lifted type parameters may be instantiated with a functional type. Within a function, a lifted type parameter is treated as a functional type.

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. The same goes when constructing sum types, as Futhark cannot assume that a given constructor only belongs to a single type. Further, unique types (see *In-place updates*) must be explicitly annotated.

3.7 Size Types

Futhark supports a simple system of size-dependent types that statically verifies that the sizes of arrays passed to a function are compatible. The focus is on simplicity, not completeness.

Whenever a pattern occurs (in `let`, `loop`, and function parameters), as well as in return types, *size annotations* 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 =
  map2 (+) a b
```

We use a *size parameter*, `[n]`, to explicitly quantify sizes. The `[n]` parameter is not explicitly passed when calling `f`. Rather, its value is implicitly deduced from the arguments passed for the value parameters. An array can contain *anonymous dimensions*, e.g. `[]i32`, for which the type checker will invent fresh size parameters, which ensures that all arrays have a (symbolic) size.

A size annotation can also be an integer constant (with no suffix). Size parameters can be used as ordinary variables within the scope of the parameters. The type checker verifies that the program obeys any constraints imposed by size annotations.

Size-dependent types are supported, as the names of parameters can be used in the return type of a function:

```
let replicate 't (n: i32) (x: t): [n]t = ...
```

An application `replicate 10 0` will have type `[10]i32`.

3.7.1 Unknown sizes

Since sizes must be constants or variables, there are many cases where the type checker cannot assign a precise size to the result of some operation. For example, the type of `concat` should conceptually be:

```
val concat [n] [m] 't : [n]t -> [m]t -> [n+m]t
```

But this is not presently allowed. Instead, the return type contains an anonymous size:

```
val concat [n] [m] 't : [n]t -> [m]t -> []t
```

When an application `concat xs ys` is found, the result will be of type `[k]t`, where `k` is a fresh *unknown* size variable that is considered distinct from every other size in the program. It is often necessary to perform a size coercion (see *Size coercion*) to convert an unknown size to a known size.

Generally, unknown sizes are constructed whenever the true size cannot be expressed. The following lists all possible sources of unknown sizes.

Size going out of scope

An unknown size is created when the proper size of an array refers to a name that has gone out of scope:

```
let c = a + b
in replicate c 0
```

The type of `replicate c 0` is `[c]i32`, but since `c` is locally bound, the type of the entire expression is `[k]i32` for some fresh `k`.

Compound expression passed as function argument

Intuitively, the type of `replicate (x+y) 0` should be `[x+y]i32`, but since sizes must be names or constants, this is not expressible. Therefore an unknown size variable is created and the size of the expression becomes `[k]i32`.

Compound expression used as range bound

While a simple range expression such as `0..<n` can be assigned type `[n]i32`, a range expression `0..<(n+1)` will give produce an unknown size.

Complex slicing

Most complex array slicing, such as `xs[a:b]`, will have an unknown size. Exceptions are listed in the *reference for slice expressions*.

Complex ranges

Most complex ranges, such as `a..<b`, will have an known size. Exceptions exist for *general ranges* and *"upto" ranges*.

Anonymous size in function return type

Whenever the result of a function application would have an anonymous size, that size is replaced with a fresh unknown size variable.

For example, `filter` has the following type:

```
val filter [n] 'a : (p: a -> bool) -> (as: [n]a) -> []a
```

Naively, an application `filter f xs` seems like it would have type `[]a`, but a fresh unknown size `k` will be created and the actual type will be `[k]a`.

Branches of `if` return arrays of different sizes

When an `if` (or `match`) expression has branches that returns array of different sizes, the differing sizes will be replaced with fresh unknown sizes. For example:

```
if b then [[1,2], [3,4]]
     else [[5,6]]
```

This expression will have type `[k][2]i32`, for some fresh `k`.

Important: The check whether the sizes differ is done when first encountering the `if` or `match` during type checking. At this point, the type checker may not realise that the two sizes are actually equal, even though constraints later in the function force them to be. This can always be resolved by adding type annotations.

An array produced by a loop does not have a known size

If the size of some loop parameter is not maintained across a loop iteration, the final result of the loop will contain unknown sizes. For example:

```
loop xs = [1] for i < n do xs ++ xs
```

Similar to conditionals, the type checker may sometimes be too cautious in assuming that some size may change during the loop. Adding type annotations to the loop parameter can be used to resolve this.

3.7.2 Size coercion

Size coercion, written with `:>`, can be used to perform a runtime-checked coercion of one size to another. Since size annotations can refer only to variables and constants, this is necessary when writing more complicated size functions:

```
let concat_to 'a (m: i32) (a: []a) (b: []a) : [m]a =
  a ++ b :> [m]a
```

Only expression-level type annotations give rise to run-time checks. Despite their similar syntax, parameter and return type annotations must be valid at compile-time, or type checking will fail.

3.7.3 Causality restriction

Conceptually, size parameters are assigned their value by reading the sizes of concrete values passed along as parameters. This means that any size parameter must be used as the size of some parameter. This is an error:

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

The following is not an error:

```
let f [n] (g: [n]i32 -> [n]i32) = ...
```

However, using this function comes with a constraint: whenever an application `f x` occurs, the value of the size parameter must be inferable. Specifically, this value must have been used as the size of an array *before* the `f x` application is encountered. The notion of "before" is subtle, as there is no evaluation ordering of a Futhark expression, *except* that a `let`-binding is always evaluated before its body, and the argument to a function is always evaluated before the function.

The causality restriction only occurs when a function has size parameters whose first use is *not* as a concrete array size. For example, it does not apply to uses of the following function:

```
let f [n] (arr: [n]i32) (g: [n]i32 -> [n]i32) = ...
```

This is because the proper value of `n` can be read directly from the actual size of the array.

3.7.4 Empty array literals

Just as with size-polymorphic functions, when constructing an empty array, we must know the exact size of the (missing) elements. For example, in the following program we are forcing the elements of `a` to be the same as the elements of `b`, but the size of the elements of `b` are not known at the time `a` is constructed:

```
let main (b: bool) (xs: []i32) =
  let a = [] : [][]i32
  let b = [filter (>0) xs]
  in a[0] == b[0]
```

The result is a type error.

3.7.5 Sum types

When constructing a value of a sum type, the compiler must still be able to determine the size of the constructors that are *not* used. This is illegal:

```
type sum = #foo ([]i32) | #bar ([]i32)

let main (xs: *[]i32) =
  let v : sum = #foo xs
  in xs
```

3.7.6 Modules

When matching a module with a module type (see *Module System*), a non-lifted abstract type (i.e. one that is declared with `type` rather than `type^`) may not be implemented by a type abbreviation that contains any anonymous sizes. This is to ensure that if we have the following:

```
module m : { type t } = ...
```

Then we can construct an array of values of type `m.t` without worrying about constructing an irregular array.

3.7.7 Higher-order functions

When a higher-order function takes a functional argument whose return type is a non-lifted type parameter, any instantiation of that type parameter must have a non-anonymous size. If the return type is a lifted type parameter, then the instantiation may contain anonymous sizes. This is why the type of `map` guarantees regular arrays:

```
val map [n] 'a 'b : (a -> b) -> [n]a -> [n]b
```

The type parameter `b` can only be replaced with a type that has non-anonymous sizes, which means they must be the same for every application of the function. In contrast, this is the type of the pipeline operator:

```
val (|>) '^a -> '^b : a -> (a -> b) -> b
```

The provided function can return something with an anonymous size (such as `filter`).

A function whose return type has an unknown size

If a function (named or anonymous) is inferred to have a return type that contains an unknown size variable created *within* the function body, that size variable will be replaced with an anonymous size. In most cases this is not important, but it means that an expression like the following is ill-typed:

```
map (\xs -> iota (length xs)) (xss : [n][m]i32)
```

This is because the `(length xs)` expression gives rise to some fresh size `k`. The lambda is then assigned the type `[n]t -> [k]i32`, which is immediately turned into `[n]t -> []i32` because `k` was generated inside its body. A function of this type cannot be passed to `map`, as explained before. The solution is to bind `length` to a name *before* the lambda.

3.8 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.

3.8.1 Alias Analysis

The rules used by the Futhark compiler to determine aliasing are intuitive in the intra-procedural case. Aliases are associated with entire arrays. Aliases of a record are tuple are tracked for each element, not for the record or tuple itself. Most constructs produce fresh arrays, with no aliases. The main exceptions are `if`, `loop`, function calls, and variable literals.

- After a binding `let a = b`, that simply assigns a new name to an existing variable, the variable `a` aliases `b`. Similarly for record projections and patterns.
- The result of an `if` aliases the union of the aliases of the components.
- The result of a `loop` aliases the initial values, as well as any aliases that the merge parameters may assume at the end of an iteration, computed to a fixed point.
- The aliases of a value returned from a function is the most interesting case, and depends on whether the return value is declared *unique* (with an asterisk `*`) or not. If it is declared unique, then it has no aliases. Otherwise, it aliases all arguments passed for *non-unique* parameters.

3.8.2 In-place Updates and Higher-Order Functions

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.

3.9 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 = ...
```

A named module type is defined as:

```
module type name = ...
```

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 = ...
```

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`.

In a value spec, sizes in types on the left-hand side of a function arrow must not be anonymous. For example, this is forbidden:

```
val sum: []t -> t
```

Instead write:

```
val sum [n]: [n]t -> t
```

But this is allowed, because the empty size is not to the left of a function arrow:

```
val evens [n]: [n]i32 -> []i32
```

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

```
module Times = \(M: Addable) -> {
  let times (x: M.t) (k: i32): M.t =
    loop x' = x for i < k do
      M.add x' x
}
```

We can instantiate `Times` with any module that fulfils 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`.

3.9.1 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*.

3.9.2 Module Type Expressions

```
mod_type_exp  ::=   qualid
                  | "{" spec* "}"
                  | mod_type_exp "with" qualid type_param* "=" 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 type_param* ":" 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 type. They can specify of course that a module contains a *submodule* of a specific module type.

3.10 Referring to Other Files

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

```
import "file"
```

The above will include all non-`local` top-level definitions from `file.fut` is and make them available in the current file (but will not export them). 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.

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

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

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

```
local open import "file"
```

CHAPTER 4

Package Management

This document describes `futhark pkg`, a minimalistic package manager inspired by vgo. A Futhark package is a downloadable collection of `.fut` files and little more. There is a (not necessarily comprehensive) list of known packages.

4.1 Basic Concepts

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

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

Most `futhark pkg` operations involve reading and writing a *package manifest*, which is always stored in a file called `futhark.pkg`. The `futhark.pkg` file is human-editable, but is in day-to-day use mainly modified by `futhark pkg` automatically.

4.2 Using Packages

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

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

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

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

This lists one required package, with its package path, minimum version (see *Version Selection*), and the expected commit hash. The latter is used for verification, to ensure that the contents of a package version cannot be changed silently.

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

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

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

3 directories, 1 file
```

Warning: `futhark pkg sync` will remove any unrecognized files or local modifications to files in `lib/` (except of course the package directory of the package path listed in `futhark.pkg`; see *Creating Packages*).

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

```
$ futhark pkg remove pkgpath
```

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

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

4.2.1 Importing Files from Dependencies

`futhark pkg sync` will populate the `lib/` directory, but does not interact with the compiler in any way. The downloaded files can be imported using the usual `import` mechanism (*Referring to Other Files*); for example, assuming the package contains a file `foo.fut`:

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

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

4.2.2 Upgrading Dependencies

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

As an example:

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

4 directories, 2 files
```

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

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

4.3 Creating Packages

A package is a directory tree (which at the moment must correspond to a Git repository). It *must* contain two things:

- A file `futhark.pkg` at the root defining the package path and any required packages.
- A *package directory* `lib/pkg-path`, where `pkg-path` is the full package path.

The contents of the package directory is what will be made available to users of the package. The repository may contain other things (tests, data files, examples, docs, other programs, etc), but these are ignored by `futhark pkg`. This structure can be created automatically by running for example:

```
$ futhark pkg init github.com/sturluson/edda
```

Note again, no `https://`. The result is this `futhark.pkg`:

```
package github.com/sturluson/edda

require {
}
```

And this file hierarchy:

```
$ tree lib
lib
└── github.com
    └── sturluson
        └── edda

3 directories, 0 files
```

Note that `futhark pkg init` is not necessary simply to *use* packages, only when *creating* packages.

When creating a package, the `.fut` files we are writing will be located inside the `lib/` directory. If the package has its own dependencies, whose files we would like to access, we can use *relative imports*. For example, assume we are creating a package `github.com/sturluson/edda` and we are writing a Futhark file located at `lib/github.com/sturluson/edda/saga.fut`. Further, we have a dependency on the package `github.com/athas/foo-fut`, which is stored in the directory `lib/github.com/athas/foo-fut`. We can import a file `lib/github.com/athas/foo-fut/foo.fut` from `lib/github.com/sturluson/edda/saga.fut` with:

```
import "../foo-fut/foo"
```

4.3.1 Releasing a Package

Currently, a package corresponds exactly to a GitHub repository mirroring the package path. A release is done by tagging an appropriate commit with `git tag vX.Y.Z` and then pushing the tag to GitHub with `git push --tags`. In the future, this will be generalised to other code hosting sites and version control systems (and possibly self-hosted tarballs). Remember to take semantic versioning into account - unless you bump the major version number (or the major version is 0), the new version must be *fully compatible* with the old.

When releasing a new package, consider getting it added to the central package list. See this page for details.

4.3.2 Incrementing the Major Version Number

While backwards-incompatible modifications to a package are sometimes unavoidable, it is wise to avoid them as much as possible, as they significantly inconvenience users. To discourage breaking compatibility, `futhark pkg` tries to ensure that the package developer feels this inconvenience as well. In many cases, an incompatible change can be avoided simply by adding new files to the package rather than incompatibly changing the existing ones.

In the general case, the package path also encodes the major version of the package, separated with a `@`. For example, version 5.2.1 of a package might have the package path `github.com/user/repo@5`. For major versions 0 and 1, this can be elided. This means that multiple (major) versions of a package are completely distinct from the point of view of the package manager - this principle is called Semantic Import Versioning, and is intended to facilitate backwards compatibility of packages when new versions are released.

If you really must increment the major version, then you will need to change the package path in `futhark.pkg` to contain the new major version preceded by `@`. For example, `lib/github.com/sturluson/edda` becomes `lib/github.com/sturluson/edda@2`. As a special case, this is not necessary when moving from major version 0 to 1. Since the package path has changed, you will also need to rename the package directory in `lib/`. This is painful and awkward, but it is less painful and awkward than what users feel when their dependencies break compatibility.

4.3.3 Renaming a Package

It is likely that the hosting location for a very long-lived package will change from time to time. Since the hosting location is embedded into the package path itself, this causes some issues for `futhark pkg`.

In simple cases, there is no problem. Consider a package `github.com/asgard/loki` which is moved to `github.com/utgard/loki`. If no GitHub-level redirect is set up, all users must update the path by which they import the package. This is unavoidable, unfortunately.

However, the old tagged versions, which contain a `futhark.pkg` that uses the old package path, will continue to work. This is because the package path indicated in `package.pkg` merely defines the subdirectory of `lib/` where the package files are to be found, while the package path used by dependents in the `require` section defines where the package files are located after `futhark pkg sync`. Thus, when we import an old version of `github.com/utgard/loki` whose `futhark.pkg` defines the package as `github.com/asgard/loki`, the package files will be retrieved from the `lib/github.com/asgard/loki` directory in the repository, but stored at `lib/github.com/utgard/loki` in the local directory.

The above means that package management remains operational in simple cases of renaming, but it is awkward when a transitive dependency is renamed (or deleted). The Futhark package ecosystem is sufficiently embryonic that we have not yet developed more robust solutions. When such solutions are developed, they will likely involve some form of `replace` directive that allows transparent local renaming of packages, as well as perhaps a central registry of package paths that does not depend on specific source code hosts.

4.4 Version Selection

The package manifest `futhark.pkg` declares which packages the program depends on. Dependencies are specified as the *oldest acceptable version* within the given major version. Upper version bounds are not supported, as strict adherence to semantic versioning is assumed, so any later version with the same major version number should work. When `futhark pkg sync` calculates which version of a given package to download, it will pick the oldest version that still satisfies the minimum version requirements of that package in all transitive dependencies. This means that a version may be used that is newer than the one indicated in `futhark.pkg`, but only if a dependency requires a more recent version.

4.5 Tests and Documentation for Dependencies

Package management has been designed to ensure that the normal development tools work as expected with the contents of the `lib/` directory. For example, to ensure that all dependencies do in fact work well (or at least compile) together, run:

```
futhark test lib
```

Also, you can generate hyperlinked documentation for all dependencies with:

```
futhark doc lib -o docs
```

The file `docs/index.html` can be opened in a web browser to browse the documentation. Prebuilt documentation is also available via the online package list.

4.6 Safety

In contrast to some other package managers, `futhark pkg` does not run any package-supplied code on installation, upgrade, or removal. This means that all `futhark pkg` operations are in principle completely safe (barring exploitable bugs in `futhark pkg` itself, which is unlikely but not impossible). Further, Futhark code itself is also completely pure, so executing it cannot have any unfortunate effects, such as infecting all of your own packages with a worm. The worst it can do is loop infinitely, consume arbitrarily large amounts of memory, or produce wrong results.

The exception is packages that uses `unsafe`. With some cleverness, `unsafe` can be combined with in-place updates to perform arbitrary memory reads and writes, which can trivially lead to exploitable behaviour. You should not use untrusted code that employs `unsafe` (but the `--safe` compiler option may help). However, this is not any worse than calling external code in a conventional impure language, which generally can perform any conceivable harmful action.

CHAPTER 5

C Porting Guide

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

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

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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 6

Futhark Compared to Other Functional Languages

This guide is intended for programmers who are familiar with other functional languages and want to start working 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 family. 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 understand the meaning of a Futhark program, and quickly begin writing their own programs. To speed up this process, we describe here some of the various quirks and unexpected limitations imposed by Futhark. We also recommended reading some of the example programs along with this guide. The guide does *not* cover all Futhark features worth knowing, so do also skim *Language Reference*.

6.1 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*. 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#.

6.2 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. 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 work as usual (although functions are not fully first class; see *Types*). Some Futhark language constructs look like functions, but are not. This means they cannot be partially applied. These include `unsafe` and `assert`.

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, like 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.

6.3 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.

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.

All types can be combined in tuples as usual, as well as in *structurally typed records*, as in Standard ML. Non-recursive sum types are supported, and are also structurally typed. 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 0-indexed numeric label. For example, `(i32, bool)` is the same as `{0: i32, 1: bool}`, and can be indexed as `foo.1`.

Sum types are defined as constructors separated by a vertical bar (`|`). Constructor names always start with a `#`. For example, `#red | #blue i32` is a sum type with the constructors `#red` and `#blue`, where the latter has an `i32` as payload. The terms `#red` and `#blue 2` produce values of this type. Constructor applications must always be fully saturated. Due to the structural typing, type annotations are usually necessary to resolve ambiguities. For example, the term `#blue 2` can produce a value of *any type* that has an appropriate constructor.

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 not restricted from being instantiated with function types. On the other hand, in the function definition they are subject to the same restrictions as functional types.

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

```
let id x = x
```

Type abbreviations 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. Most unusually, this is also the case for sum types. The following two types are entirely interchangeable:

```
type maybe 'a = #just a | #nothing

type option 'a = #nothing | #just a
```

Only for abstract types, where the definition has been hidden via the module system, do type names have any significance.

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 7

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 as 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:

- 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 wish 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.

7.1 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.

7.2 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.

7.3 Using `futhark dev`

For debugging specific compiler passes, the `futhark dev` subcommand 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 8

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 `futhark run`.

You can generate random data in the binary format with `futhark dataset` (*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.

8.1 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>
```

8.1.1 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 9

futhark

9.1 SYNOPSIS

futhark <subcommand> options...

9.2 DESCRIPTION

Futhark is a data-parallel functional array language. Through various subcommands, the `futhark` tool provides facilities for compiling, developing, or analysing Futhark programs. Most subcommands are documented in their own manpage. For example, `futhark opencl` is documented as *futhark-opencl*. The remaining subcommands are documented in this page.

9.3 COMMANDS

9.3.1 futhark check [-w] PROGRAM

Check whether a Futhark program type checks. With `-w`, no warnings are printed.

9.3.2 futhark datacmp FILE_A FILE_B

Check whether the two files contain the same Futhark values. The files must be formatted using the general Futhark data format that is used by all other executable and tools (such as *futhark-dataset*). All discrepancies will be reported. This is in contrast to *futhark-test*, which only reports the first one.

9.3.3 futhark dataget PROGRAM DATASET

Find the test dataset whose description contains `DATASET` (e.g. `#1`) and print it in binary representation to standard output.

9.3.4 futhark dev options... PROGRAM

A Futhark compiler development command, intentionally undocumented and intended for use in developing the Futhark compiler, not for programmers writing in Futhark.

9.3.5 futhark imports PROGRAM

Print all non-builtin imported Futhark files to stdout, one per line.

9.3.6 futhark query PROGRAM LINE COL

Print information about the variable at the given position in the program.

9.4 SEE ALSO

futhark-opencl, *futhark-c*, *futhark-python*, *futhark-pyopencl*, *futhark-dataset*, *futhark-doc*, *futhark-test*, *futhark-bench*, *futhark-run*, *futhark-repl*

CHAPTER 10

futhark-autotune

10.1 SYNOPSIS

futhark autotune [options...] program

10.2 DESCRIPTION

`futhark-autotune` attemps to find optimal values for threshold parameters given representative datasets. This is done by repeatedly running running the program through *futhark-bench* with different values for the threshold parameters. When `futhark-autotune` finishes tuning a program `foo.fut`, the results are written to `foo.fut.tuning`, which will then automatically be picked up by subsequent uses of *futhark-bench* and *futhark-test*.

Currently, only the entry point named `main` is tuned.

10.3 OPTIONS

--backend=name The backend used when compiling Futhark programs (without leading `futhark`, e.g. just `opencl`).

--futhark=program The program used to perform operations (eg. compilation). Defaults to the binary running `futhark autotune` itself.

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

```
futhark autotune prog.fut --backend=opencl --pass-option=-
↪dHawaii
```

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

-v, --verbose Print verbose information about what the tuner is doing. Pass multiple times to increase the amount of information printed.

--tuning=EXTENSION Change the extension used for tuning files (`.tuning` by default).

--timeout=seconds Initial tuning timeout for each dataset in seconds. After running the intitial tuning run on each dataset, the timeout is based on the run time of that initial tuning. Defaults to 60.

A negative timeout means to wait indefinitely.

10.4 SEE ALSO

futhark-bench

CHAPTER 11

futhark-bench

11.1 SYNOPSIS

futhark bench [options...] programs...

11.2 DESCRIPTION

This tool is the recommended way to benchmark Futhark programs. Programs are compiled using the specified backend (`futhark c` by default), then run a number of times for each test case, and the average runtime printed on standard output. Refer to *futhark-test* for information on how to format test data. A program will be ignored if it contains no data sets - it will not even be compiled.

If compilation of a program fails, then `futhark bench` will abort immediately. If execution of a test set 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.

11.3 OPTIONS

--backend=name The backend used when compiling Futhark programs (without leading `futhark`, e.g. just `opencl`).

--concurrency=NUM The number of benchmark programs to prepare concurrently. Defaults to the number of cores available. *Prepare* means to compile the benchmark, as well as generate any needed datasets. In some cases, this generation can take too much memory, in which case lowering `--concurrency` may help.

--entry-point=name Only run entry points with this name.

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

--futhark=program The program used to perform operations (eg. compilation). Defaults to the binary running `futhark bench` itself.

--ignore-files=REGEX Ignore files whose path match the given regular expression.

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

--no-tuning Do not look for tuning files.

--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 --backend=opencl --pass-option=-
↪dHawaii
```

--pass-compiler-option=opt Pass an extra option to the compiler when compiling the programs.

--runner=program If set to a non-empty string, compiled programs are not run directly, but instead the indicated *program* is run with its first argument being the path to the compiled Futhark program. This is useful for compilation targets that cannot be executed directly (as with *futhark-csharp*), or when you wish to run the program on a remote machine.

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

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

--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.

-v, --verbose Print verbose information about what the benchmark is doing. Pass multiple times to increase the amount of information printed.

--tuning=EXTENSION For each program being run, look for a tuning file with this extension, which is suffixed to the name of the program. For example, given `--tuning=tuning` (the default), the program `foo.fut` will be passed the tuning file `foo.fut.tuning` if it exists.

11.4 WHAT FUTHARK BENCH MEASURES

`futhark bench` measures the time it takes to run the given Futhark program by passing the `-t FILE` option to the generated program. See the man page for the specific compiler to see exactly what is measured.

11.5 EXAMPLES

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

```
-- How quickly can we reduce arrays?
--
-- ==
-- nobench input { 0 }
```

(continues on next page)

(continued from previous page)

```
-- 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)
```

11.6 SEE ALSO

futhark-c, *futhark-test*

CHAPTER 12

futhark-c

12.1 SYNOPSIS

futhark c [options...] infile

12.2 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.

12.3 OPTIONS

-h Print help text to standard output and exit.

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

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

--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.

-V Print version information on standard output and exit.

-W Do not print any warnings.

--Werror Treat warnings as errors.

12.4 EXECUTABLE OPTIONS

The following options are accepted by executables generated by `futhark c`.

-b, --binary-output Print the program result in the binary output format. The default is human-readable text, which is very slow.

-D, --debugging Perform possibly expensive internal correctness checks and verbose logging. Implies `-L`.

-e, --entry-point=FUN The entry point to run. Defaults to `main`.

-L, --log Print various low-overhead logging information to stderr while running.

-r, --runs=NUM Perform NUM runs of the program. With `-t`, the runtime for each individual run will be printed. Additionally, a single leading warmup run will be performed (not counted). Only the final run will have its result written to stdout.

-t, --write-runtime-to=FILE Print the time taken to execute the program to the indicated file, an integral number of microseconds.

12.5 SEE ALSO

futhark-opencl, *futhark-cuda*, *futhark-test*

CHAPTER 13

futhark-csharp

13.1 SYNOPSIS

futhark csharp [options...] infile

13.2 DESCRIPTION

`futhark csharp` translates a Futhark program to sequential C# code, and either compiles that C# code with the Roslyn C# Compiler `csc` to an executable binary program, or produces a `.dll` file that can be linked with other code.. The standard Futhark optimisation pipeline is used, and `csc` is invoked with `-lib:$MONO_PATH`, `-r:Mono.Options.dll`, and `/unsafe`.

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.

13.3 OPTIONS

-h Print help text to standard output and exit.

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

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

--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.

-W Do not print any warnings.

-V Print version information on standard output and exit.

13.4 REQUIREMENTS

`futhark csharp` uses the Mono implementation of the .NET framework. To compile and execute the compiled binaries/libraries, you must have the `MONO_PATH` environment variable defined. `MONO_PATH` must be set to a directory containing the `Mono.Options` dll.

Mono.Options is available on https://www.nuget.org/packages/Mono.Options/5.3.0.1

13.5 SEE ALSO

futhark-csopencl

CHAPTER 14

futhark-csopencl

14.1 SYNOPSIS

futhark csopencl [options...] infile

14.2 DESCRIPTION

`futhark csopencl` translates a Futhark program to C# code invoking OpenCL kernels, and either compiles that C# code with the Roslyn C# Compiler `csc` to an executable binary program, or produces a `.dll` file that can be linked with other code.. The standard Futhark optimisation pipeline is used, and `csc` is invoked with `-lib:$MONO_PATH`, `-r:Cloo.clSharp.dll`, `-r:Mono.Options.dll`, and `/unsafe`.

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

14.3 OPTIONS

-h Print help text to standard output and exit.

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

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

--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.

-V Print version information on standard output and exit.

-W Do not print any warnings.

--Werror Treat warnings as errors.

14.4 REQUIREMENTS

`futhark csopencl` uses the Mono implementation of the .NET framework. To compile and execute the compiled binaries/libraries, you must have the `MONO_PATH` environment variable defined. `MONO_PATH` must be set to a directory containing the `Mono.Options` and `Cloo.clSharp` dll's.

Mono.Options is available on https://www.nuget.org/packages/Mono.Options/5.3.0.1

Cloo.clSharp is available on https://www.nuget.org/packages/Cloo.clSharp/

14.5 SEE ALSO

futhark-test, *futhark-csharp*

CHAPTER 15

futhark-cuda

15.1 SYNOPSIS

futhark cuda [options...] infile

15.2 DESCRIPTION

`futhark cuda` translates a Futhark program to C code invoking CUDA 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 `-O`, `-lm`, and `-std=c99`. The resulting program will otherwise behave exactly as one compiled with `futhark c`.

`futhark cuda` uses `-lcuda -lnvrtc` to link. If using `--library`, you will need to do the same when linking the final binary.

15.3 OPTIONS

-h Print help text to standard output and exit.

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

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

--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.

-V Print version information on standard output and exit.

-W Do not print any warnings.

--Werror Treat warnings as errors.

15.4 EXECUTABLE OPTIONS

Generated executables accept the same options as those generated by *futhark-c*. The `-t` option behaves as with *futhark-opencl*. For commonality, the options use OpenCL nomenclature ("group" instead of "thread block").

The following additional options are accepted.

--default-group-size=INT The default size of thread blocks that are launched. Capped to the hardware limit if necessary.

--default-num-groups=INT The default number of thread blocks that are launched.

--default-threshold=INT The default parallelism threshold used for comparisons when selecting between code versions generated by incremental flattening. Intuitively, the amount of parallelism needed to saturate the GPU.

--default-tile-size=INT The default tile size used when performing two-dimensional tiling (the work-group size will be the square of the tile size).

--dump-cuda=FILE Don't run the program, but instead dump the embedded CUDA kernels to the indicated file. Useful if you want to see what is actually being executed.

--dump-ptx=FILE Don't run the program, but instead dump the PTX-compiled version of the embedded kernels to the indicated file.

--load-cuda=FILE Instead of using the embedded CUDA kernels, load them from the indicated file.

--load-ptx=FILE Load PTX code from the indicated file.

--nvrtc-option=OPT Add an additional build option to the string passed to NVRTC. Refer to the CUDA documentation for which options are supported. Be careful - some options can easily result in invalid results.

--print-sizes Print all sizes that can be set with `-size` or `--tuning`.

–size=NAME=INT

Set a configurable run-time parameter to the given value. Use `--print-sizes` to see which are available.

--tuning=FILE Read size=value assignments from the given file.

15.5 ENVIRONMENT

If run without `--library`, `futhark cuda` will invoke `gcc(1)` to compile the generated C program into a binary. This only works if `gcc` can find the necessary CUDA libraries. On most systems, CUDA is installed in `/usr/local/cuda`, which is not part of the default `gcc` search path. You may need to set the following environment variables before running `futhark cuda`:

```
LIBRARY_PATH=/usr/local/cuda/lib64
LD_LIBRARY_PATH=/usr/local/cuda/lib64/
CPATH=/usr/local/cuda/include
```

15.6 SEE ALSO

futhark-opencl

CHAPTER 16

futhark-dataset

16.1 SYNOPSIS

futhark dataset [options...]

16.2 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.

16.3 OPTIONS

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

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

The type may also be a value, in which case that literal value is generated.

-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.

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

16.4 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
```

16.5 SEE ALSO

futhark-test, *futhark-bench*

CHAPTER 17

futhark-doc

17.1 SYNOPSIS

futhark doc [options...] dir

17.2 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*.

`futhark doc` will ignore any file whose documentation comment consists solely of the word "ignore". This is useful for files that contain tests, or are otherwise not relevant to the reader of the documentation.

17.3 OPTIONS

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

17.4 EXAMPLES

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

17.5 SEE ALSO

futhark-test, *futhark-bench*

CHAPTER 18

futhark-opencl

18.1 SYNOPSIS

futhark opencl [options...] infile

18.2 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 `-O`, `-lm`, and `-std=c99`. The resulting program will otherwise behave exactly as one compiled with `futhark c`.

`futhark opencl` uses `-lOpenCL` to link (`-framework OpenCL` on macOS). If using `--library`, you will need to do the same when linking the final binary.

18.3 OPTIONS

-h Print help text to standard output and exit.

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

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

--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.

-V Print version information on standard output and exit.

-W Do not print any warnings.

--Werror Treat warnings as errors.

18.4 EXECUTABLE OPTIONS

Generated executables accept the same options as those generated by *futhark-c*. For the `-t` option, The time taken to perform device setup or teardown, including writing the input or reading the result, is not included in the measurement. In particular, this means that timing starts after all kernels have been compiled and data has been copied to the device buffers but before setting any kernel arguments. Timing stops after the kernels are done running, but before data has been read from the buffers or the buffers have been released.

The following additional options are accepted.

--build-option=OPT Add an additional build option to the string passed to `clBuildProgram()`. Refer to the OpenCL documentation for which options are supported. Be careful - some options can easily result in invalid results.

--default-group-size=INT The default size of OpenCL workgroups that are launched. Capped to the hardware limit if necessary.

--default-num-groups=INT The default number of OpenCL workgroups that are launched.

--default-threshold=INT The default parallelism threshold used for comparisons when selecting between code versions generated by incremental flattening. Intuitively, the amount of parallelism needed to saturate the GPU.

--default-tile-size=INT The default tile size used when performing two-dimensional tiling (the workgroup size will be the square of the tile size).

-d, --device=NAME Use 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 Don't run the program, but instead dump the embedded OpenCL program to the indicated file. Useful if you want to see what is actually being executed.

--dump-opencl-binary=FILE Don't run the program, but instead dump the compiled version of the embedded OpenCL program to the indicated file. On NVIDIA platforms, this will be PTX code.

--load-opencl=FILE Instead of using the embedded OpenCL program, load it from the indicated file.

--load-opencl-binary=FILE Load an OpenCL binary from the indicated file.

-p, --platform=NAME Use 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.

--print-sizes Print all sizes that can be set with `-size` or `--tuning`.

-P, --profile Gather profiling data while executing and print out a summary at the end. When `-r` is used, only the last run will be profiled. Implied by `-D`.

–size=NAME=INT

Set a configurable run-time parameter to the given value. Use `--print-sizes` to see which are available.

--tuning=FILE Read size=value assignments from the given file.

18.5 SEE ALSO

futhark-test, *futhark-cuda*, *futhark-c*

CHAPTER 19

futhark-pkg

19.1 SYNOPSIS

futhark pkg add PKGPATH [X.Y.Z]

futhark pkg check

futhark pkg init PKGPATH

futhark pkg fmt

futhark pkg remove PKGPATH

futhark pkg sync

futhark pkg upgrade

futhark pkg versions

19.2 DESCRIPTION

This tool is used to modify the package manifest (`futhark.pkg`) and download the required packages it describes. `futhark pkg` is not a build system; you will still need to compile your Futhark code with the usual compilers. The only purpose of `futhark pkg` is to download code (and perform other package management utility tasks). This manpage is not a general introduction to package management in Futhark; see the User's Guide for that.

The `futhark pkg` subcommands will modify only two locations in the file system (relative to the current working directory): the `futhark.pkg` file, and the contents of `lib/`. When modifying `lib/`, `futhark pkg` constructs the new version in `lib~new/` and backs up the old version in `lib~old`. If `futhark pkg` should fail for any reason, you can recover the old state by moving `lib~old` back. These temporary directories are erased if `futhark pkg` finishes without errors.

The `futhark pkg sync` and `futhark pkg init` subcommands are the only ones that actually modifies `lib/` ; the others modify only `futhark.pkg` and require you to manually run `futhark pkg sync` afterwards.

Most commands take a `-v`/`--verbose` option that makes `futhark pkg` write running diagnostics to stderr.

Network requests (exclusively HTTP GETs) are done via `curl`, which must be available on the `PATH`.

19.3 COMMANDS

19.3.1 futhark pkg add PKGPATH [X.Y.Z]

Add the specified package of the given minimum version as a requirement to `futhark.pkg`. If no version is provided, the newest one is used. If the package is already required in `futhark.pkg`, the new version requirement will replace the old one.

Note that adding a package does not automatically download it. Run `futhark pkg sync` to do that.

19.3.2 futhark pkg check

Verify that the `futhark.pkg` is valid, that all required packages are available in the indicated versions. This command does not check that these versions contain well-formed code. If a package path is defined in `futhark.pkg`, also checks that `.fut` files are located at the expected location in the file system.

19.3.3 futhark pkg init PKGPATH

Create a new `futhark.pkg` defining a package with the given package path, and initially no requirements.

19.3.4 futhark pkg fmt

Reformat the `futhark.pkg` file, while retaining any comments.

19.3.5 futhark pkg remove PKGPATH

Remove a package from `futhark.pkg`. Does *not* remove it from the `lib/` directory.

19.3.6 futhark pkg sync

Populate the `lib/` directory with the packages listed in `futhark.pkg`. **Warning**: this will delete everything in `lib/` that does not relate to a file listed in `futhark.pkg`, as well as any local modifications.

19.3.7 futhark pkg upgrade

Upgrade all package requirements in `futhark.pkg` to the newest available versions.

19.3.8 futhark pkg versions PKGPATH

Print all available versions for the given package path.

19.4 COMMIT VERSIONS

It is possible to use `futhark pkg` with packages that have not yet made proper releases. This is done via pseudoversions of the form `0.0.0-yyyymmddhhmmss+commitid`. The timestamp is not verified against the actual commit. The timestamp ensures that newer commits take precedence if multiple packages depend on a commit version for the same package. If `futhark pkg add` is given a package with no releases, the most recent commit will be used. In this case, the timestamp is merely set to the current time.

Commit versions are awkward and fragile, and should not be relied upon. Issue proper releases (even experimental 0.x version) as soon as feasible. Released versions also always take precedence over commit versions, since any version number will be greater than 0.0.0.

19.5 EXAMPLES

Create a new package that will be hosted at `https://github.com/sturluson/edda`:

```
futhark pkg init github.com/sturluson/edda
```

Add a package dependency:

```
futhark pkg add github.com/sturluson/hattatal
```

Download the dependencies:

```
futhark pkg sync
```

And then you're ready to start hacking! (Except that these packages do not actually exist.)

19.6 BUGS

Since the `lib/` directory is populated with transitive dependencies as well, it is possible for a package to depend unwittingly on one of the dependencies of its dependencies, without the `futhark.pkg` file reflecting this.

There is no caching of zipballs and version lists between invocations, so the network traffic can be rather heavy.

Only GitHub and GitLab are supported as code hosting sites.

19.7 SEE ALSO

futhark-test, *futhark-doc*

CHAPTER 20

futhark-pyopencl

20.1 SYNOPSIS

futhark pyopencl [options...] infile

20.2 DESCRIPTION

`futhark pyopencl` translates a Futhark program to Python code invoking OpenCL kernels, which depends on Numpy and PyOpenCL. 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 opencl`. 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.

20.3 OPTIONS

-h Print help text to standard output and exit.

--library Instead of compiling to an executable program, generate a Python module that can be imported 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.

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

--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.
-V	Print version information on standard output and exit.
-W	Do not print any warnings.
--Werror	Treat warnings as errors.

20.4 SEE ALSO

futhark-python, *futhark-opencl*

CHAPTER 21

futhark-python

21.1 SYNOPSIS

futhark python [options...] infile

21.2 DESCRIPTION

`futhark python` translates a Futhark program to sequential Python code, which depends on Numpy.

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.

The generated code is very slow, likely too slow to be useful. It is more interesting to use this command's big brother, *futhark-pyopencl*.

21.3 OPTIONS

-h Print help text to standard output and exit.

--library Instead of compiling to an executable program, generate a Python module that can be imported 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.

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

--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.

-V	Print version information on standard output and exit.
-W	Do not print any warnings.
--Werror	Treat warnings as errors.

21.4 SEE ALSO

futhark-pyopencl

CHAPTER 22

futhark-repl

22.1 SYNOPSIS

futhark repl

22.2 DESCRIPTION

Start an interactive Futhark session. This will let you interactively enter expressions and declarations which are then immediately interpreted. If the entered line can be either a declaration or an expression, it is assumed to be a declaration.

Futhark source files can be loaded using the `:load` command. This will erase any interactively entered definitions. Use the `:help` command to see a list of commands. All commands are prefixed with a colon.

`futhark-repl` uses the Futhark interpreter, which grants access to certain special functions. See *futhark-run* for a description.

22.3 OPTIONS

-h Print help text to standard output and exit.

-V Print version information on standard output and exit.

22.4 SEE ALSO

futhark-run, *futhark-test*

CHAPTER 23

futhark-run

23.1 SYNOPSIS

futhark run [program]

23.2 DESCRIPTION

Execute the given program by evaluating the `main` function with arguments read from standard input, and write the results on standard output.

`futhark-run` is very slow, and in practice only useful for testing, teaching, and experimenting with the language. Certain special debugging functions are available in `futhark-run`:

`trace 'a : a -> a` Semantically identity, but prints the value on standard output.

`break 'a : a -> a` Semantically identity, but interrupts execution at the calling point, such that the environment can be inspected. Continue execution by entering an empty input line. Breakpoints are only respected when starting a program from the prompt, not when passing a program on the command line.

23.3 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.
-w, --no-warnings	Disable interpreter warnings.

23.4 SEE ALSO

futhark-repl, *futhark-test*

CHAPTER 24

futhark-test

24.1 SYNOPSIS

futhark test [options...] infiles...

24.2 DESCRIPTION

This tool is used to test Futhark programs based on input/output datasets. 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: names...]
[compiled|nobench|random] input ({ values... } | @ filename)
output { values... } | auto output | 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*.

If `input` is preceded by `random`, the text between the curly braces must consist of a sequence of Futhark types, including sizes in the case of arrays. When `futhark test` is run, a file located in a `data/` subdirectory, containing

values of the indicated types and shapes is, automatically constructed with `futhark-dataset`. Apart from sizes, integer constants (with or without type suffix) are also permitted.

If `input` is followed by an `@` and a file name (which must not contain any whitespace) instead of curly braces, values will be read from the indicated file. This is recommended for large data sets. This notation cannot be used with `random` input.

After the `input` block, the expected result of the test case is written as either `output` followed by 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.

If `output` is preceded by `auto` (as in `auto output`), the expected values are automatically generated by compiling the program with `futhark-c` and recording its result for the given input (which must not fail). This is usually only useful for testing or benchmarking alternative compilers, and not for testing the correctness of Futhark programs.

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, dashes, and underscores, 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. Multiple entry points can be specified on the same line by separating them with space, and they will all be tested with the same input/output pairs. 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, `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.

24.3 OPTIONS

--backend=program The backend used when compiling Futhark programs (without leading `futhark`, e.g. just `opencl`).

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

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

--concurrency=NUM The number of tests to run concurrently. Defaults to the number of (hyper-)cores available.

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

-i Only interpret - do not run any compilers.

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

--futhark=program The program used to perform operations (eg. compilation). Defaults to the binary running `futhark test` itself.

--no-terminal Print each result on a line by itself, without line buffering.

--no-tuning Do not look for tuning files.

--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 test prog.fut --backend=opencl --pass-option=-
↪dHawaii
```

--pass-compiler-option=opt Pass an extra option to the compiler when compiling the programs.

--runner=program If set to a non-empty string, compiled programs are not run directly, but instead the indicated *program* is run with its first argument being the path to the compiled Futhark program. This is useful for compilation targets that cannot be executed directly (as with *futhark-csharp*), or when you wish to run the program on a remote machine.

--tuning=EXTENSION For each program being run, look for a tuning file with this extension, which is suffixed to the name of the program. For example, given `--tuning=tuning` (the default), the program `foo.fut` will be passed the tuning file `foo.fut.tuning` if it exists.

24.4 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
```

(continues on next page)

(continued from previous page)

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

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

The following program contains an entry point that is tested with randomly generated data:

```
-- ==
-- random input { [100]i32 [100]i32 } auto output
-- random input { [1000]i32 [1000]i32 } auto output

let main xs ys = i32.product (map2 (*) xs ys)
```

24.5 SEE ALSO

futhark-bench, *futhark-repl*