Futhark User's Guide

Release 0.22.3

DIKU

Oct 28, 2022

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 built-in prelude is also available online.

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

CHAPTER

ONE

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, see read *Setting up Futhark on Windows*. If you are using macOS, see *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`, `tinfo`, and `zlib` 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 zlib1g-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

To compile Futhark you must first install an appropriate version of GHC, either with ghcup or a package manager. Any version since GHC 8.10 should work. You also need the `cabal` command line program, which ghcup will install for you as well.

You then 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
```

First you must run the following command to download metadata about Futhark's dependencies:

```
$ make configure
```

To build the Futhark compiler and all of its dependencies, run:

```
$ make build
```

This step typically requires at least 8GiB of memory. This will create files in your `~/.cabal` and `~/.ghc` directories.

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

```
$ make install
```

You can set the `PREFIX` environment variable to indicate a different installation path. Note that this does not install the Futhark manual pages. You can delete `~/.cabal` and `~/.ghc` after this if you wish - the `futhark` binary will still work.

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.

We build such binary snapshots for the following operating systems:

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

You will still likely need to make a C compiler (such as GCC) available on your own.

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

Due to limited maintenance and testing resources, Futhark is not directly supported on Windows. Install WSL and follow the Linux instructions above. The C code generated by the Futhark compiler should work on Windows.

In the future, we may support Windows directly again.

1.7 Futhark with Nix

Futhark mostly works fine with Nix and NixOS, but when using OpenCL you may need to make more packages available in your environment. This is regardless of whether you are using the `futhark` package from Nixpkgs or one you have installed otherwise.

- On NixOS, for OpenCL, you should import `opencl-headers` and `ocl-icd`. You also need some form of OpenCL backend. If you have an AMD GPU and use ROCm, you may also need `rocm-opencl-runtime`.
- On NixOS, for CUDA (and probably also OpenCL on NVIDIA GPUs), you need `cudatoolkit`. However, `cudatoolkit` does not appear to provide `libcuda.so` and similar libraries. These are instead provided in an `nvidia_x11` package that is specific to some kernel version, e.g. `linuxPackages_5_4.nvidia_x11`. You will need this as well.

- On macOS, for OpenCL, you need `darwin.apple_sdk.frameworks.OpenCL`.

These can be easily made available with e.g:

```
nix-shell -p opencl-headers -p ocl-icd
```

CHAPTER

TWO

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 three main ways of compiling a Futhark program: to an ordinary executable (by using `--executable`, which is the default), to a *server executable* (`--server`), and to a library (`--library`). Plain executables can be run immediately, but are useful mostly for testing and benchmarking. Server executables are discussed in *Server Protocol*. 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 `cc` 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.

`-h/--help`

Print help text to standard output and exit.

`-D/--debugging`

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. This implies `--log`.

`-L/--log`

Print low-overhead logging information during initialisation and during execution of entry points. Enabling this option should not affect program performance.

`--cache-file FILE`

Create (if necessary) and use data in the provided cache file to speed up subsequent launches of the same program. The cache file is automatically updated by the running program as necessary. It is safe to delete at any time, and will be recreated as necessary.

`--print-params`

Print a list of tuning parameters followed by their *parameter class* in parentheses, which indicates what they are used for.

`--param SIZE=VALUE`

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

`--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-params` 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 line using the `--params` option, but more convenient.

2.1.2 Non-Server Executable Options

The following options are only supported on non-server executables, because they make no sense in a server context.

`-t/--write-runtime-to 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 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.

`-b/--binary-output`

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

`-n/--no-print-result`

Do not print the result of running the program.

GPU Options

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

`-d/--device DEVICE`

Pick the first 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.

`-P/--profile`

Measure the time taken by various GPU 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.

OpenCL-specific Options

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

`-p/--platform 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. If used in conjunction with `-d`, 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.

`--list-devices`

List all OpenCL devices and platforms available on the system.

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.

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.

Multicore options

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

`--num-threads INT`

The number of threads used to run parallel operations. If set to a value less than 1, then the runtime system will use one thread per detected core.

`-P/--profile`

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

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. If the entry point returns an *m*-element tuple, then the function will return *m* values (although the tuple can be replaced with a single opaque value, see below). 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.

The entry point should have a name that is also a valid C identifier.

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 and using a type abbreviation is the best way around this.

Value Mapping

The rules for how Futhark values are mapped to target language values are as follows:

- Primitive types or arrays of primitive types are mapped transparently (although for the C backends, this still involves a distinct type for arrays).
- All other types are mapped to an opaque type. Use a type ascription with a type abbreviation to give it a specific name, otherwise one will be generated.

Return types follow these rules, with one addition:

- If the return type is an *m*-element tuple, then the function returns *m* values, mapped according to the rules above (but not including this one - nested tuples are not mapped directly). This rule does not apply when the entry point has been given a return type ascription that is not syntactically a tuple type.

Consumption and Aliasing

Futhark's support for *In-place Updates* has implications for the generated API. Unfortunately, The type system of most languages (e.g. C) is not rich enough to express the rules, so they are not statically (or currently even dynamically checked). Since Futhark will never infer a unique/consuming type for an entry point parameter, this section can be ignored unless uniqueness annotations have been manually added to the entry points parameter types. The rules are essentially the same as in the language itself:

1. Each entry point input parameter is either *consuming* or *nonconsuming* (the default). This corresponds to unique and nonunique types in the original Futhark program. A value passed for a consuming parameter is considered *consumed*, now has an unspecified value, and may never be used again. It must still be manually freed, if applicable. Further, any *aliases* of that value are also considered consumed and may not be used.
2. Each entry point output is either *unique* or *nonunique*. A unique output has no aliases. A nonunique output aliases *every* nonconsuming input parameter.

Note that these distinctions are currently usually not visible in the generated API, and so correct usage requires knowledge of the original types in the Futhark function. The safest strategy is to not expose unique types in entry points.

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 three files in the current directory: `futlib.c`, `futlib.h`, and `futlib.json` (see *Manifest* for more on the latter).

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. See *C API Reference* for more information.

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. The API functions are all thread safe.

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.

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.

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.

2.3 Reproducibility

The Futhark compiler is deterministic by design, meaning that repeatedly compiling the *same program* with the *same compilation flags* and using the *same version* of the compiler will produce identical output every time.

Note that this only applies to the code generated by the Futhark compiler itself. When compiling to an executable with one of the C backends (see *Compiling to Executable*), Futhark will invoke a C compiler that may not be perfectly reproducible. In such cases the generated `.c` and `.h` files will be reproducible, but the final executable may not.

CHAPTER

THREE

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 built-in prelude is available elsewhere.

3.1 Comments

Line comments are indicated with `--` and continue until end of line. A contiguous block of line comments beginning with `-- |` is a *documentation comment* and has special meaning to documentation tools. Documentation comments are only allowed immediately before declarations.

3.2 Identifiers and Keywords

```
id           ::=   letter constituent* | "_" constituent*
constituent  ::=   letter | digit | "_" | "'"
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 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.

3.3 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 `f16`, `f32`, `f64`, as well as `bool`.

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

Numeric literals can be suffixed with their intended type. For example `42i8` is of type `i8`, and `1337e2f64` is of type `f64`. If no suffix is given, the type of the literal will be inferred based on its use. If the use is not constrained, integral literals will be assigned type `i32`, and decimal literals type `f64`. Hexadecimal literals are supported by prefixing with `0x`, and binary literals by prefixing with `0b`.

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

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

```
floatnumber       ::=   (pointfloat | exponentfloat | hexadecimalfloat) [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"
```

3.3.1 Compound Types and Values

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

```
                | existential_size
```

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

```
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. An anonymous size is a syntactic shorthand, and is always replaced by an actual size by the type checker (either via inference or by inventing a new name, depending on context).

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 double quotes>
charlit     ::=   "'" char "'"
char        ::=   <any source character except "\" or newline or single 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, but character literals are interpreted as integers of the corresponding Unicode code point.

```
existential_size  ::=   "?" ("[" id "]")+ "." type
```

An existential size quantifier brings an unknown size into scope within a type. This can be used to encode constraints for statically unknowable array sizes.

3.4 Declarations

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

```
dec  ::=   val_bind | type_bind | mod_bind | mod_type_bind
         | "open" mod_exp
         | "import" stringlit
         | "local" dec
         | "#[" attr "]" dec
```

Any names defined by a declaration inside a module are by default visible to users of that module (see *Modules*).

- `open mod_exp` brings names bound in `mod_exp` into the current scope. These names will also be visible to users of the module.
- `local dec` has the meaning of `dec`, but any names bound by `dec` will not be visible outside the module.
- `import "foo"` is a shorthand for `local open import "foo"`, where the `import` is interpreted as a module expression (see *Modules*).
- `#[attr] dec` adds an attribute to a declaration (see *Attributes*).

3.4.1 Declaring Functions and Values

```
val_bind  ::=  ("def" | "entry" | "let") (id | "(" binop ")") type_param* pat* [":" type] "=" exp
             | ("def" | "entry" | "let") pat binop pat [":" type] "=" exp
```

Note: using `let` to define top-level bindings is deprecated.

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

```
def 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*:

```
def 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:

```
def 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:

```
def 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.

3.4.2 User-Defined Operators

Infix operators are defined much like functions:

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

For example:

```
def (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:

```
def (+^) (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:

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

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

3.4.3 Entry Points

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

The name of an entry point must not contain an apostrophe (`'`), even though that is normally permitted in Futhark identifiers.

3.4.4 Value Declarations

A named value/constant can be declared as follows:

```
def 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. If the return type contains any anonymous sizes (see *Size types*), new existential sizes will be constructed for them.

3.4.5 Type Abbreviations

```
type_bind   ::=   ("type" | "type^" | "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 existential 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 existential 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 size, or as a size 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)
```

(continues on next page)

(continued from previous page)

```
def 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
def 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.

3.5 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
                  | charlit
                  | "(" ")"
                  | "(" exp ")" ("." fieldid)*
                  | "(" exp ("," exp)* ")"
                  | "{" "}"
                  | "{" field ("," field)* "}"
                  | qualid "[" index ("," index)* "]"
                  | "(" exp ")" "[" index ("," index)* "]"
                  | quals "." "(" exp ")"
                  | "[" exp ("," exp)* "]"
                  | "[" exp [".." exp] "..." exp "]"
                  | "(" qualbinop ")"
                  | "(" exp qualbinop ")"
                  | "(" qualbinop exp ")"
                  | "(" ( "." field )+ ")"
                  | "(" "." "[" index ("," index)* "]" ")"
                  | "???"
exp           ::=   atom
                  | exp qualbinop exp
                  | exp exp
                  | "!" exp
                  | "-" exp
                  | constructor exp*
                  | exp ":" type
                  | exp ":>" type
```

```
                  | exp [ ".." exp ] "..." exp
                  | exp [ ".." exp ] "..<" exp
                  | exp [ ".." exp ] "..>" exp
                  | "if" exp "then" exp "else" exp
                  | "let" size* 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
                  | "#[" attr "]" 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
size        ::=   "[" id "]"
pat         ::=   id
                  | pat_literal
                  | "_"
                  | "(" ")"
                  | "(" pat ")"
                  | "(" pat ("," pat)+ ")"
                  | "{" "}"
                  | "{" fieldid ["=" pat] ("," fieldid ["=" pat])* "}"
                  | constructor pat*
                  | pat ":" type
                  | "#[" attr "]" pat
pat_literal ::=   [ "-" ] intnumber
                  | [ "-" ] floatnumber
                  | charlit
                  | "true"
                  | "false"
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.

3.5.1 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 whether `x` is a value or module.
- 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 `-`.
- Function and type application, and prefix operators, bind more tightly than any infix operator. Note that the only prefix operators are `!` and `-`, and more cannot be defined.
- `#foo #bar` is interpreted as a constructor with a `#bar` payload, not as applying `#foo` to `#bar` (the latter would be semantically invalid anyway).
- The following table describes the precedence and associativity of infix operators in both expressions and type expressions. 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	`` `op` ``
left	`||`
left	`&&`
left	`<=` `>=` `>` `<` `==` `!=`
left	`&` `^` `|`
left	`<<` `>>`
left	`+` `-`
left	`*` `/` `%` `//` `%%`
left	`|>`
right	`<|`
right	`->`
left	juxtaposition

3.5.2 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.

3.5.3 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 `[]u8` that contains the characters encoded as UTF-8.

`()`

Evaluates to an empty tuple.

`( e )`

Evaluates to the result of `e`.

`???`

A *typed hole*, usable as a placeholder expression. The type checker will infer any necessary type for this expression. This can sometimes result in an ambiguous type, which can be resolved using a type ascription. Evaluating a typed hole results in a run-time error.

`(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 index may be of any unsigned integer type.

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). Slicing can be done only with expressions of type `i64`.

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 a signed integer array whose first element is `x` and which proceeds with a 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 less 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 a signed integer array whose first elements is `x`, and which proceeds upwards with a stride of `y-x` 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 less 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 a signed integer array whose first elements is `x`, and which proceeds downwards with a stride of `y-x` 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. **Shifting is undefined if the right operand is negative, or greater than or equal to the length in bits of the left operand.**

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.

`` `op` ``

Use `op`, which may be any non-operator function name, as an infix operator.

`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 `i64`, 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.

`#[attr] e`

Apply the given attribute to the expression. Attributes are an ad-hoc and optional mechanism for providing extra information, directives, or hints to the compiler. See *Attributes* for more information.

`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.

This construct is deprecated. Use the `#[unsafe]` attribute instead.

`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`.

3.5.4 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 [n] pat = e in body`

As above, but bind sizes (here `n`) used in the pattern (here to the size of the array being bound). All sizes must be used in the pattern. Roughly Equivalent to `let f [n] pat = body in f e`.

`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.

3.5.5 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]`.

3.6 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 consumption interacts with higher-order functions.

3.7 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, consumed parameters (see *In-place updates*) must be explicitly annotated.

Type inference processes top-level declared in top-down order, and the type of a top-level function must be completely inferred at its definition site. Specifically, if a top-level function uses overloaded arithmetic operators, the resolution of those overloads cannot be influenced by later uses of the function.

3.8 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:

```
def 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 type 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. On the right-hand side of a function arrow ("return types"), this results in an *existential size* that is not known until the function is fully applied, e.g:

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

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:

```
def replicate 't (n: i64) (x: t): [n]t = ...
```

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

Whenever we write a type `[n]t`, `n` must already be a variable of type `i64` in scope (possibly by being bound as a size parameter).

3.8.1 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 existential size:

```
val concat [n] [m] 't : [n]t -> [m]t -> ?[k].[k]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 unknown size. Exceptions exist for *general ranges* and *"upto" ranges*.

Existential size in function return type

Whenever the result of a function application has an existential 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) -> ?[k].[k]a
```

For an application `filter f xs`, the type checker invents a fresh unknown size `k'`, and the actual type for this specific application 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.

3.8.2 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:

```
def 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.

3.8.3 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:

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

The following is not an error:

```
def 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, the argument to a function is always evaluated before the function itself, and the left operand to an operator is evaluated before the right.

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:

```
def 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.

3.8.4 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:

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

The result is a type error.

3.8.5 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)

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

3.8.6 Modules

When matching a module with a module type (see *Modules*), 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 existential 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.

3.8.7 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-existential size. If the return type is a lifted type parameter, then the instantiation may contain existential 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-existential 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 existential 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 existential 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 -> ?[k].[k]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.

3.8.8 Sizes in type abbreviations

When anonymous sizes are passed to type abbreviations, if that anonymous size is eventually instantiated with an existential size, the *same* existential size is going to be used for all instances of the corresponding parameter in the right-hand-side of the type abbreviation. Note that this breaks with the usual conception of type abbreviations as purely a shorthand, as this could not be expressed without the abbreviation. Example:

```
type square [n] = [n][n]i32
```

The following function is be *known* to return a square array:

```
val f : () -> square []
```

But this is not the case for the function that inlines the definition of `square`:

```
val g : () -> [][]i32
```

As this above would be elaborated as follows:

```
val g : () -> ?[n][m].[n][m]i32
```

We can of course explicitly write that the function returns a square array of existential size:

```
val g : () -> ?[n].[n]i32
```

3.9 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 we can mark it as *consuming* by prefixing it with an asterisk. For a return type, we can mark it as *alias-free* by prefixing it with an asterisk. For example:

```
def 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.

3.9.1 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 *alias-free* (with an asterisk `*`) or not. If it is declared alias-free, then it has no aliases. Otherwise, it aliases all arguments passed for *non-consumed* parameters.

3.9.2 In-place Updates and Higher-Order Functions

Consumption generally interacts inflexibly 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 consumption 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.

3.10 Modules

```
mod_bind       ::=   "module" id mod_param* "=" [":" mod_type_exp] "=" mod_exp
mod_param      ::=   "(" id ":" mod_type_exp ")"
mod_type_bind  ::=   "module" "type" id "=" 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.

3.10.1 Module bindings

`module m = mod_exp`

Binds *m* to the module produced by the module expression `mod_exp`. Any name x in the module produced by `mod_exp` can then be accessed with `m.x`.

`module m : mod_type_exp = mod_exp`

Shorthand for `module m = mod_exp : mod_type_exp`.

`module m mod_params... = mod_exp`

Shorthand for `module m = \mod_params... -> mod_exp`. This produces a parametric module.

`module type mt = mod_type_exp`

Binds *mt* to the module type produced by the module type expression `mod_type_exp`.

3.10.2 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.

3.10.3 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* ":" type
          | "val" binop type_param* ":" type
          | ("type" | "type^" | "type~") id type_param* "=" type
          | ("type" | "type^" | "type~") id type_param*
          | "module" id ":" mod_type_exp
          | "include" mod_type_exp
          | "#[" attr "]" spec
```

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.

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*.

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
```

3.11 Referencing 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"
```

To re-export names from another file in the current module, use:

```
open import "file"
```

3.12 Attributes

```
attr  ::=  id
         | decimal
         | id "(" [attr ("," attr)*] ")"
```

An expression, declaration, pattern, or module type spec can be prefixed with an attribute, written as `#[attr]`. This may affect how it is treated by the compiler or other tools. In no case will attributes affect or change the *semantics* of a program, but it may affect how well it compiles and runs (or in some cases, whether it compiles or runs at all). Unknown attributes are silently ignored. Most have no effect in the interpreter. An attribute can be either an *atom*, written as an identifier or number, or *compound*, consisting of an identifier and a comma-separated sequence of attributes. The latter is used for grouping and encoding of more complex information.

3.12.1 Expression attributes

Many expression attributes affect second-order array combinators (*SOACS*). These must be applied to a fully saturated function application or they will have no effect. If two SOACs with contradictory attributes are combined through fusion, it is unspecified which attributes take precedence.

The following expression attributes are supported.

`trace`

Print the value produced by the attributed expression. Used for debugging. Somewhat unreliable outside of the interpreter, and in particular does not work for GPU device code.

`trace(tag)`

Like `trace`, but prefix output with *tag*, which must lexically be an identifier.

`break`

In the interpreter, pause execution *before* evaluating the expression. No effect for compiled code.

`opaque`

The compiler will treat the attributed expression as a black box. This is used to work around optimisation deficiencies (or bugs), although it should hopefully rarely be necessary.

`incremental_flattening(no_outer)`

When using incremental flattening, do not generate the "only outer parallelism" version for the attributed SOACs.

incremental_flattening(no_intra)

When using incremental flattening, do not generate the "intra-group parallelism" version for the attributed SOACs.

incremental_flattening(only_intra)

When using incremental flattening, *only* generate the "intra-group parallelism" version of the attributed SOACs. **Beware**: the resulting program will fail to run if the inner parallelism does not fit on the device.

incremental_flattening(only_inner)

When using incremental flattening, do not generate multiple versions for this SOAC, but do exploit inner parallelism (which may give rise to multiple versions at deeper levels).

noinline

Do not inline the attributed function application. If used within a parallel construct (e.g. `map`), this will likely prevent the GPU backends from generating working code.

sequential

Fully sequentialise the attributed SOAC.

sequential_outer

Turn the outer parallelism in the attributed SOAC sequential, but preserve any inner parallelism.

sequential_inner

Exploit only outer parallelism in the attributed SOAC.

unroll

Fully unroll the attributed `loop`. If the compiler cannot determine the exact number of iterations (possibly after other optimisations and simplifications have taken place), then this attribute has no code generation effect, but instead results in a warning. Be very careful with this attribute: it can massively increase program size (possibly crashing the compiler) if the loop has a huge number of iterations.

unsafe

Do not perform any dynamic safety checks (such as bound checks) during execution of the attributed expression.

`warn(safety_checks)`

Make the compiler issue a warning if the attributed expression (or its subexpressions) requires safety checks (such as bounds checking) at run-time. This is used for performance-critical code where you want to be told when the compiler is unable to statically verify the safety of all operations.

3.12.2 Declaration attributes

The following declaration attributes are supported.

`noinline`

Do not inline any calls to this function. If the function is then used within a parallel construct (e.g. `map`), this will likely prevent the GPU backends from generating working code.

`inline`

Always inline calls to this function.

3.12.3 Pattern attributes

No pattern attributes are currently supported by the compiler itself, although they are syntactically permitted and may be used by other tools.

3.12.4 Spec attributes

No spec attributes are currently supported by the compiler itself, although they are syntactically permitted and may be used by other tools.

CHAPTER

FOUR

C API REFERENCE

A Futhark program `futlib.fut` compiled to a C library with the `--library` command line option produces two files: `futlib.c` and `futlib.h`. The API provided in the `.h` file is documented in the following.

Using the API requires creating a *configuration object*, which is then used to obtain a *context object*, which is then used to perform most other operations, such as calling Futhark functions.

Most functions that can fail return an integer: 0 on success and a non-zero value on error, as documented below. Others return a `NULL` pointer. Use *futhark_context_get_error()* to get a (possibly) more precise error message.

FUTHARK_BACKEND_foo

A preprocessor macro identifying that the backend *foo* was used to generate the code; e.g. `c`, `opencl`, or `cuda`. This can be used for conditional compilation of code that only works with specific backends.

4.1 Error codes

Most errors are result in a not otherwise specified nonzero return code, but a few classes of errors have distinct error codes.

FUTHARK_SUCCESS

Defined as `0`. Returned in case of success.

FUTHARK_PROGRAM_ERROR

Defined as `2`. Returned when the program fails due to out-of-bounds, an invalid size coercion, invalid entry point arguments, or similar misuse.

FUTHARK_OUT_OF_MEMORY

Defined as `3`. Returned when the program fails to allocate memory. This is (somewhat) reliable only for GPU memory - due to overcommit and other VM tricks, you should not expect running out of main memory to be reported gracefully.

4.2 Configuration

Context creation is parameterised by a configuration object. Any changes to the configuration must be made *before* calling *futhark_context_new()*. A configuration object must not be freed before any context objects for which it is used. The same configuration may *not* be used for multiple concurrent contexts.

struct **futhark_context_config**

An opaque struct representing a Futhark configuration.

struct *futhark_context_config* ***futhark_context_config_new**(void)

Produce a new configuration object. You must call *futhark_context_config_free()* when you are done with it.

void **futhark_context_config_free**(struct *futhark_context_config* *cfg)

Free the configuration object.

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

With a nonzero flag, enable various debugging information, with the details specific to the backend. This may involve spewing copious amounts of information to the standard error stream. It is also likely to make the program run much slower.

void **futhark_context_config_set_profiling**(struct *futhark_context_config* *cfg, int flag)

With a nonzero flag, enable the capture of profiling information. This should not significantly impact program performance. Use *futhark_context_report()* to retrieve captured information, the details of which are backend-specific.

void **futhark_context_config_set_logging**(struct *futhark_context_config* *cfg, int flag)

With a nonzero flag, print a running log to standard error of what the program is doing.

int **futhark_context_config_set_tuning_param**(struct *futhark_context_config* *cfg, const char *param_name, size_t new_value)

Set the value of a tuning parameter. Returns zero on success, and non-zero if the parameter cannot be set. This is usually because a parameter of the given name does not exist. See *futhark_get_tuning_param_count()* and *futhark_get_tuning_param_name()* for how to query which parameters are available. Most of the tuning parameters are applied only when the context is created, but some may be changed even after the context is active. At the moment, only parameters of class "threshold" may change after the context has been created. Use *futhark_get_tuning_param_class()* to determine the class of a tuning parameter.

int **futhark_get_tuning_param_count**(void)

Return the number of available tuning parameters. Useful for knowing how to call *futhark_get_tuning_param_name()* and *futhark_get_tuning_param_class()*.

const char ***futhark_get_tuning_param_name**(int i)

Return the name of tuning parameter *i*, counting from zero.

const char ***futhark_get_tuning_param_class**(int i)

Return the class of tuning parameter *i*, counting from zero.

void **futhark_context_config_set_cache_file**(struct *futhark_context_config* *cfg, const char *fname)

Ask the Futhark context to use a file with the designated file as a cross-execution cache. This can result in faster initialisation of the program next time it is run. For example, the GPU backends will store JIT-compiled GPU code in this file.

The cache is managed entirely automatically, and if it is invalid or stale, the program performs initialisation from scratch. There is no machine-readable way to get information about whether the cache was hit succesfully, but you can enable logging to see what hapens.

The lifespan of `fname` must exceed the lifespan of the configuration object. Pass `NULL` to disable caching (this is the default).

4.3 Context

struct **futhark_context**

An opaque struct representing a Futhark context.

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

Create a new context object. You must call *futhark_context_free()* when you are done with it. It is fine for multiple contexts to co-exist within the same process, but you must not pass values between them. They have the same C type, so this is an easy mistake to make.

After you have created a context object, you must immediately call *futhark_context_get_error()*, which will return non-NULL if initialisation failed. If initialisation has failed, then you still need to call *futhark_context_free()* to release resources used for the context object, but you may not use the context object for anything else.

void **futhark_context_free**(struct *futhark_context* *ctx)

Free the context object. It must not be used again. You must call *futhark_context_sync()* before calling this function to ensure there are no outstanding asynchronous operations still running. The configuration must be freed separately with *futhark_context_config_free()*.

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

Block until all outstanding operations, including copies, have finished executing. Many API functions are asynchronous on their own.

void **futhark_context_pause_profiling**(struct *futhark_context* *ctx)

Temporarily suspend the collection of profiling information. Has no effect if profiling was not enabled in the configuration.

void **futhark_context_unpause_profiling**(struct *futhark_context* *ctx)

Resume the collection of profiling information. Has no effect if profiling was not enabled in the configuration.

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

A human-readable string describing the last error. Returns NULL if no error has occurred. It is the caller's responsibility to free() the returned string. Any subsequent call to the function returns NULL, until a new error occurs.

void **futhark_context_set_logging_file**(struct *futhark_context* *ctx, FILE *f)

Set the stream used to print diagnostics, debug prints, and logging messages during runtime. This is stderr by default. Even when this is used to re-route logging messages, fatal errors will still only be printed to stderr.

char ***futhark_context_report**(struct *futhark_context* *ctx)

Produce a human-readable C string with debug and profiling information collected during program runtime. It is the caller's responsibility to free the returned string. It is likely to only contain interesting information if *futhark_context_config_set_debugging()* or *futhark_context_config_set_profiling()* has been called previously. Returns NULL on failure.

int **futhark_context_clear_caches**(struct *futhark_context* *ctx)

Release any context-internal caches and buffers that may otherwise use computer resources. This is useful for freeing up those resources when no Futhark entry points are expected to run for some time. Particularly relevant when using a GPU backend, due to the relative scarcity of GPU memory.

4.4 Values

Primitive types (`i32`, `bool`, etc) are mapped directly to their corresponding C type. The `f16` type is mapped to `uint16_t`, because C does not have a standard `half` type. This integer contains the bitwise representation of the `f16` value in the IEEE 754 binary16 format.

For each distinct array type of primitives (ignoring sizes), an opaque C struct is defined. Arrays of `f16` are presented as containing `uint16_t` elements. For types that do not map cleanly to C, including records, sum types, and arrays of tuples, see *Opaque values*.

All array values share a similar API, which is illustrated here for the case of the type `[]i32`. The creation/retrieval functions are all asynchronous, so make sure to call *futhark_context_sync()* when appropriate. Memory management is entirely manual. All values that are created with a `new` function, or returned from an entry point, *must* at some point be freed manually. Values are internally reference counted, so even for entry points that return their input unchanged, you should still free both the input and the output - this will not result in a double free.

struct **futhark_i32_1d**

An opaque struct representing a Futhark value of type `[]i32`.

struct *futhark_i32_1d* ***futhark_new_i32_1d**(struct *futhark_context* *ctx, int32_t *data, int64_t dim0)

Asynchronously create a new array based on the given data. The dimensions express the number of elements. The data is copied into the new value. It is the caller's responsibility to eventually call *futhark_free_i32_1d()*. Multi-dimensional arrays are assumed to be in row-major form. Returns `NULL` on failure.

struct *futhark_i32_1d* ***futhark_new_raw_i32_1d**(struct *futhark_context* *ctx, char *data, int64_t offset, int64_t dim0)

Create an array based on *raw* data, as well as an offset into it. This differs little from *futhark_i32_1d()* when using the `c` backend, but when using e.g. the `opencl` backend, the `data` parameter will be a `cl_mem`. It is the caller's responsibility to eventually call *futhark_free_i32_1d()*. The `data` pointer must remain valid for the lifetime of the array. Unless you are very careful, this basically means for the lifetime of the context. Returns `NULL` on failure.

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

Free the value. In practice, this merely decrements the reference count by one. The value (or at least this reference) may not be used again after this function returns.

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

Asynchronously copy data from the value into `data`, which must be of sufficient size. Multi-dimensional arrays are written in row-major form.

const int64_t ***futhark_shape_i32_1d**(struct *futhark_context* *ctx, struct *futhark_i32_1d* *arr)

Return a pointer to the shape of the array, with one element per dimension. The lifetime of the shape is the same as `arr`, and should *not* be manually freed. Assuming `arr` is a valid object, this function cannot fail.

4.4.1 Opaque values

Each instance of a complex type in an entry point (records, nested tuples, etc) is represented by an opaque C struct named `futhark_opaque_foo`. In the general case, `foo` will be a hash of the internal representation. However, if you insert explicit type annotations in the entry point (and the type name contains only characters valid for C identifiers), the indicated name will be used. Note that arrays contain brackets, which are usually not valid in identifiers. Defining a simple type abbreviation is the best way around this.

The API for opaque values is similar to that of arrays, and the same rules for memory management apply. You cannot construct them from scratch, but must obtain them via entry points (or deserialisation, see *futhark_restore_opaque_foo()*).

struct **futhark_opaque_foo**

An opaque struct representing a Futhark value of type `foo`.

int **futhark_free_opaque_foo**(struct *futhark_context* *ctx, struct *futhark_opaque_foo* *obj)

Free the value. In practice, this merely decrements the reference count by one. The value (or at least this reference) may not be used again after this function returns.

int **futhark_store_opaque_foo**(struct *futhark_context* *ctx, const struct *futhark_opaque_foo* *obj, void **p, size_t *n)

Serialise an opaque value to a byte sequence, which can later be restored with *futhark_restore_opaque_foo()*. The byte representation is not otherwise specified, and is not stable between compiler versions or programs. It is stable under change of compiler backend, but not change of compiler version, or modification to the source program (although in most cases the format will not change).

The variable pointed to by `n` will always be set to the number of bytes needed to represent the value. The `p` parameter is more complex:

- If `p` is `NULL`, the function will write to `*n`, but not actually serialise the opaque value.
- If `*p` is `NULL`, the function will allocate sufficient storage with `malloc()`, serialise the value, and write the address of the byte representation to `*p`.
- Otherwise, the serialised representation of the value will be stored at `*p`, which *must* have room for at least `*n` bytes. This is done asynchronously.

Returns 0 on success.

struct *futhark_opaque_foo* ***futhark_restore_opaque_foo**(struct *futhark_context* *ctx, const void *p)

Asynchronously restore a byte sequence previously written with *futhark_store_opaque_foo()*. Returns `NULL` on failure. The byte sequence does not need to have been generated by the same program *instance*, but it *must* have been generated by the same Futhark program, and compiled with the same version of the Futhark compiler.

4.4.2 Records

A record is an opaque type (see above) that supports additional functions to *project* individual fields (read their values) to construct a value given values for the fields. An opaque type is a record if its definition is a record at the Futhark level.

The projection and construction functions are equivalent in functionality to writing entry points by hand, and so serve only to cut down on boilerplate. Important things to be aware of:

1. The objects constructed though these functions have their own lifetime (like any objects returned from an entry point) and must be manually freed, independently of the records from which they are projected, or the fields they are constructed from.
2. The objects are however in an *aliasing* relationship with the fields or original record. This means you must be careful when passing them to entry points that consume their arguments. As always, you don't have to worry about this if you never write entry points that consume their arguments.

The precise functions generated depend on the fields of the record. The following functions assume a record with Futhark-level type `type t = {foo: t1, bar: t2}` where `t1` and `t2` are also opaque types.

int **futhark_new_opaque_t**(struct *futhark_context* *ctx, struct futhark_opaque_t **out, const struct futhark_opaque_t2 *bar, const struct futhark_opaque_t1 *foo);

Construct a record in `*out` which has the given values for the `bar` and `foo` fields. The parameter ordering constitutes the fields in alphabetic order. Tuple fields are named `vX` where `X` is an integer. The resulting record

aliases the values provided for `bar` and `foo`, but has its own lifetime, and all values must be individually freed when they are no longer needed.

int **futhark_project_opaque_t_bar**(struct *futhark_context* *ctx, struct futhark_opaque_t2 **out, const struct futhark_opaque_t *obj);

Extract the value of the field `bar` from the provided record. The resulting value *aliases* the record, but has its own lifetime, and must eventually be freed.

int **futhark_project_opaque_t_foo**(struct *futhark_context* *ctx, struct futhark_opaque_t1 **out, const struct futhark_opaque_t *obj);

Extract the value of the field `bar` from the provided record. The resulting value *aliases* the record, but has its own lifetime, and must eventually be freed.

4.5 Entry points

Entry points are mapped 1:1 to C functions. Return values are handled with *out*-parameters.

For example, this Futhark entry point:

```
entry sum = i32.sum
```

Results in the following C function:

int **futhark_entry_sum**(struct *futhark_context* *ctx, int32_t *out0, const struct *futhark_i32_1d* *in0)

Asynchronously call the entry point with the given arguments. Make sure to call *futhark_context_sync()* before using the value of `out0`.

Errors are indicated by a nonzero return value. On error, nothing is written to the *out*-parameters.

The precise semantics of the return value depends on the backend. For the sequential C backend, errors will always be available when the entry point returns, and *futhark_context_sync()* will always return zero. When using a GPU backend such as `cuda` or `opencl`, the entry point may still be running asynchronous operations when it returns, in which case the entry point may return zero successfully, even though execution has already (or will) fail. These problems will be reported when *futhark_context_sync()* is called. Therefore, be careful to check the return code of *both* the entry point itself, and *futhark_context_sync()*.

For the rules on entry points that consume their input, see *Consumption and Aliasing*. Note that even if a value has been consumed, you must still manually free it. This is the only operation that is permitted on a consumed value.

4.6 GPU

The following API functions are available when using the `opencl` or `cuda` backends.

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

Use the first 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 *futhark_context_config_set_platform()*, only the devices from matching platforms are considered.

4.6.1 Exotic

The following functions are not interesting to most users.

void **futhark_context_config_set_default_group_size**(struct *futhark_context_config* *cfg, int size)

Set the default number of work-items in a work-group.

void **futhark_context_config_set_default_num_groups**(struct *futhark_context_config* *cfg, int num)

Set the default number of work-groups used for kernels.

void **futhark_context_config_set_default_tile_size**(struct *futhark_context_config* *cfg, int num)

Set the default tile size used when executing kernels that have been block tiled.

void **futhark_context_config_dump_program_to**(struct *futhark_context_config* *cfg, const char *path)

During *futhark_context_new()*, dump the OpenCL or CUDA program source to the given file.

void **futhark_context_config_load_program_from**(struct *futhark_context_config* *cfg, const char *path)

During *futhark_context_new()*, read OpenCL or CUDA program source from the given file instead of using the embedded program.

4.7 OpenCL

The following API functions are available only when using the `opencl` backend.

void **futhark_context_config_set_platform**(struct *futhark_context_config* *cfg, const char *s)

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.

void **futhark_context_config_select_device_interactively**(struct *futhark_context_config* *cfg)

Immediately conduct an interactive dialogue on standard output to select the platform and device from a list.

struct *futhark_context* ***futhark_context_new_with_command_queue**(struct *futhark_context_config* *cfg, cl_command_queue queue)

Construct a context that uses a pre-existing command queue. This allows the caller to directly customise which device and platform is used.

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

Retrieve the command queue used by the Futhark context. Be very careful with it - enqueueing your own work is unlikely to go well.

4.7.1 Exotic

The following functions are used for debugging generated code or advanced usage.

void **futhark_context_config_add_build_option**(struct *futhark_context_config* *cfg, const char *opt)

Add a build option to the OpenCL kernel compiler. See the OpenCL specification for *clBuildProgram* for available options.

void **futhark_context_config_dump_binary_to**(struct *futhark_context_config* *cfg, const char *path)

During *futhark_context_new()*, dump the compiled OpenCL binary to the given file.

void **futhark_context_config_load_binary_from**(struct *futhark_context_config* *cfg, const char *path)

During *futhark_context_new()*, read a compiled OpenCL binary from the given file instead of using the embedded program.

4.8 CUDA

The following API functions are available when using the `cuda` backend.

4.8.1 Exotic

The following functions are used for debugging generated code or advanced usage.

void **futhark_context_config_add_nvrtc_option**(struct *futhark_context_config* *cfg, const char *opt)

Add a build option to the NVRTC compiler. See the CUDA documentation for `nvrtcCompileProgram` for available options.

void **futhark_context_config_dump_ptx_to**(struct *futhark_context_config* *cfg, const char *path)

During *futhark_context_new()*, dump the generated PTX code to the given file.

void **futhark_context_config_load_ptx_from**(struct *futhark_context_config* *cfg, const char *path)

During *futhark_context_new()*, read PTX code from the given file instead of using the embedded program.

4.9 Multicore

The following API functions are available when using the `multicore` backend.

void **futhark_context_config_set_num_threads**(struct *futhark_context_config* *cfg, int n)

The number of threads used to run parallel operations. If set to a value less than `1`, then the runtime system will use one thread per detected core.

4.10 General guarantees

Calling an entry point, or interacting with Futhark values through the functions listed above, has no system-wide side effects, such as writing to the file system, launching processes, or performing network connections. Defects in the program or Futhark compiler itself can with high probability result only in the consumption of CPU or GPU resources, or a process crash.

Using the `#[unsafe]` attribute with in-place updates can result in writes to arbitrary memory locations. A malicious program can likely exploit this to obtain arbitrary code execution, just as with any insecure C program. If you must run untrusted code, consider using the `--safe` command line option to instruct the compiler to disable `#[unsafe]`.

Initialising a Futhark context likewise has no side effects, except if explicitly configured differently, such as by using *futhark_context_config_dump_program_to()*. In its default configuration, Futhark will not access the file system.

Note that for the GPU backends, the underlying API (such as CUDA or OpenCL) may perform file system operations during startup, and perhaps for caching GPU kernels in some cases. This is beyond Futhark's control.

Violation the restrictions of consumption (see *Consumption and Aliasing*) can result in undefined behaviour. This does not matter for programs whose entry points do not have unique parameter types (*In-place Updates*).

4.11 Manifest

The C backends generate a machine-readable *manifest* in JSON format that describes the API of the compiled Futhark program. Specifically, the manifest contains:

- A mapping from the name of each entry point to:
  - The C function name of the entry point.
  - A list of all *inputs*, including their type (as a name) and *whether they are unique* (consuming).
  - A list of all *outputs*, including their type (as a name) and *whether they are unique*.
- A mapping from the name of each non-scalar type to:
  - The C type used to represent this type (which is in practice always a pointer of some kind).
  - What *kind* of type this is - either an *array* or an *opaque*.
  - For arrays, the element type and rank.
  - A mapping from *operations* to the names of the C functions that implement the operations for the type. The types of the C functions are as documented above. The following operations are listed:
    - For arrays: `free`, `shape`, `values`, `new`.
    - For opaques: `free`, `store`, `restore`.
  - For opaques that are actually records (including tuples):
    - The list of fields, including their type and a projection function. The field ordering here is the one used expected by the *new* function.
    - The name of the C *new* function for creating a record from field values.

Manifests are defined by the following JSON Schema:

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://futhark-lang.org/manifest.schema.json",
    "title": "Futhark C Manifest",
    "description": "The C API presented by a compiled Futhark program",
    "type": "object",
    "properties": {
        "backend": {"type": "string"},
        "version": {"type": "string"},
        "entry_points": {
            "type": "object",
            "additionalProperties": {
                "type": "object",
                "properties": {
                    "cfun": {"type": "string"},
                    "outputs": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "type": {"type": "string"},
                                "unique": {"type": "boolean"},
                                "additionalProperties": false
```

(continues on next page)

(continued from previous page)

```
                        }
                    }
                },
                "inputs": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "name": {"type": "string"},
                            "type": {"type": "string"},
                            "unique": {"type": "boolean"},
                            "additionalProperties": false
                        }
                    }
                }
            }
        }
    },
    "types": {
        "type": "object",
        "additionalProperties": {
            "oneOf": [
                { "type": "object",
                  "properties": {
                      "kind": {"const": "opaque"},
                      "ctype": {"type": "string"},
                      "ops": {
                          "type": "object",
                          "properties": {
                              "free": {"type": "string"},
                              "store": {"type": "string"},
                              "restore": {"type": "string"}
                          },
                          "additionalProperties": false
                      },
                      "record": {
                          "type": "object",
                          "properties": {
                              "new": {"type": "string"},
                              "fields": {
                                  "type": "array",
                                  "items": {
                                      "type": "object",
                                      "properties": {
                                          "name": {"type": "string"},
                                          "type": {"type": "string"},
                                          "project": {"type": "string"}
                                      }
                                  }
                              }
                          },
                          "additionalProperties": false
```

(continues on next page)

(continued from previous page)

```
                        }
                      },
                      "required": [ "kind", "ctype", "ops" ]
                    },
                    { "type": "object",
                      "properties": {
                          "kind": {"const": "array"},
                          "ctype": {"type": "string"},
                          "rank": {"type": "integer"},
                          "elemtype": {
                              "enum":
                              ["i8", "i16", "i32", "i64",
                               "u8", "u16", "u32", "u64",
                               "f16", "f32", "f64",
                               "bool"]
                          },
                          "ops": {
                              "type": "object",
                              "properties": {
                                  "free": {"type": "string"},
                                  "shape": {"type": "string"},
                                  "values": {"type": "string"},
                                  "new": {"type": "string"}
                              },
                              "additionalProperties": false
                          }
                      }
                    }]
            }
        }
    },
    "required": ["backend", "entry_points", "types"],
    "additionalProperties": false
}
```

It is likely that we will add more fields in the future, but it is unlikely that we will remove any.

CHAPTER

FIVE

JAVASCRIPT API REFERENCE

The *futhark-wasm* and *futhark-wasm-multicore* compilers produce JavaScript wrapper code to allow JavaScript programs to invoke the generated WebAssembly code. This chapter describes the API exposed by the wrapper.

First a warning: **the JavaScript API is experimental**. It may change incompatibly even in minor versions of the compiler.

A Futhark program `futlib.fut` compiled with a WASM backend as a library with the `--library` command line option produces four files:

- `futlib.c`, `futlib.h`: Implementation and header C files generated by the compiler, similar to `futhark c`. You can delete these - they are not needed at run-time.
- `futlib.class.js`: An intermediate build artifact. Feel free to delete it.
- **`futlib.wasm`: A compiled WebAssembly module, which must be** present at runtime.
- `futlib.mjs`: An ES6 module that can can be imported by other JavaScript code, and implements the API given in the following.

The module exports a function, `newFutharkContext`, which is a factory function that returns a Promise producing a `FutharkContext` instance (see below). A simple usage example:

```
import { newFutharkContext } from './futlib.mjs';
var fc;
newFutharkContext().then(x => fc = x);
```

5.1 General concerns

Memory management is completely manual, as JavaScript does not support finalizers that could let Futhark hook into the garbage collector. You are responsible for eventually freeing all objects produced by the API, using the appropriate methods.

5.2 FutharkContext

FutharkContext is a class that contains information about the context and configuration from the C API. It has methods for invoking the Futhark entry points and creating FutharkArrays on the WebAssembly heap.

`newFutharkContext()`

Asynchronously create a new `FutharkContext` object.

`class FutharkContext()`

A bookkeeping class representing an instance of a Futhark program. Do *not* directly invoke its constructor - always use the `newFutharkContext()` factory function.

`FutharkContext.free()`

Frees all memory created by the `FutharkContext` object. Should be called when the `FutharkContext` is done being used. It is an error use a `FutharkArray` or `FutharkOpaque` after the `FutharkContext` on which they were defined has been freed.

5.3 Values

Numeric types `u8`, `u16`, `u32`, `i8`, `i16`, `i32`, `f32`, and `f64` are mapped to JavaScript's standard number type. 64-bit integers `u64`, and `i64` are mapped to `BigInt`. `bool` is mapped to JavaScript's `boolean` type. Arrays are represented by the `FutharkArray`. complex types (records, nested tuples, etc) are represented by the `FutharkOpaque` class.

5.4 FutharkArray

`FutharkArray` has the following API

`FutharkArray.toArray()`

Returns a nested JavaScript array

`FutharkArray.toTypedArray()`

Returns a flat typed array of the underlying data.

`FutharkArray.shape()`

Returns the shape of the FutharkArray as an array of BigInts.

`FutharkArray.free()`

Frees the memory used by the FutharkArray class

`FutharkContext` also contains two functions for creating `FutharkArrays` from JavaScript arrays, and typed arrays for each array type that appears in an entry point. All array types share similar API methods on the `FutharkContext`, which is illustrated here for the case of the type `[]i32`.

`FutharkContext.new_i32_1d_from_jsarray(`*jsarray*`)`

Creates and returns a one-dimensional `i32` `FutharkArray` representing the JavaScript array jsarray

`FutharkContext.new_i32_1d(`*array*, *dim1*`)`

Creates and returns a one-dimensional `i32` `FutharkArray` representing the typed array of array, with the size given by dim1.

5.5 FutharkOpaque

Complex types (records, nested tuples, etc) are represented by `FutharkOpaque`. It has no use outside of being accepted and returned by entry point functions. For this reason the method only has one function for freeing the memory when `FutharkOpaque` is no longer used.

`FutharkOpaque.`**`free`**`()`

Frees memory used by FutharkOpaque. Should be called when Futhark Opaque is no longer used.

5.6 Entry Points

Each entry point in the compiled futhark program has an entry point method on the FutharkContext

`FutharkContext.`**`<entry_point_name>`**`(`*in1*, ..., *inN*`)`

The entry point function taking the N arguments of the Futhark entry point function, and returns the result. If the result is a tuple the return value is an array.

CHAPTER

SIX

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.

6.1 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.

6.2 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.

6.2.1 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 (*Referencing 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.

6.2.2 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 {
```

(continues on next page)

(continued from previous page)

```
  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.

6.3 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"
```

6.3.1 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.

6.3.2 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.

6.3.3 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.

6.4 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.

6.5 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.

6.6 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

SEVEN

WRITING FAST FUTHARK PROGRAMS

This document contains tips, tricks, and hints for writing efficient Futhark code. Ideally you'd need to know nothing more than an abstract cost model, but sometimes it is useful to have an idea of how the compiler will transform your program, what values look like in memory, and what kind of code the compiler will generate for you. These details are documented below. Don't be discouraged by the complexities mentioned here - most Futhark programs are written without worrying about any of these details, and they still manage to run with good performance. This document focuses on corner cases and pitfalls, which easily makes for depressing reading.

7.1 Parallelism

The Futhark compiler only generates parallel code for explicitly parallel constructs such as `map` and `reduce`. A plain `loop` will *not* result in parallel code (unless the loop body itself contains parallel operations). The most important parallel constructs are the *second-order array combinators* (SOACs) such as `map` and `reduce`, but functions such as `copy` are also parallel.

When describing the asymptotic cost of a Futhark function, it is not enough to give a traditional big-O measure of the total amount of work. Both `foldl` and `reduce` involve *O(n)* work, where *n* is the size of the input array, but `foldl` is sequential while `reduce` is parallel, and this is an important distinction. To make this distinction, each function is described by *two* costs: the *work*, which is the total amount of operations, and the *span* (sometimes called *depth*) which is intuitively the "longest chain of sequential dependencies". We say that `foldl` has span *O(n)*, while `reduce` has span *O(log(n))*. This explains that `reduce` is more parallel than `foldl`. The documentation for a Futhark function should mention both its work and span. See this for more details on parallel cost models and pointers to literature.

7.1.1 Scans and reductions

The `scan` and `reduce` SOACs are rather inefficient when their operators are on arrays. If possible, use tuples instead (see *Small Arrays*). The one exception is when the operator is a `map2` or equivalent. Example:

```
reduce (map2 (+)) (replicate n 0) xss
```

Such "vectorised" operators are typically handled quite efficiently. Although to be on the safe side, you can rewrite the above by interchanging the `reduce` and `map`:

```
map (reduce (+) 0) (transpose xss)
```

Avoid reductions over tiny arrays, e.g. `reduce (+) 0 [x,y,z]`. In such cases the compiler will generate complex code to exploit a negligible amount of parallelism. Instead, just unroll the loop manually (`x+y+z`) or perhaps use `foldl (+) 0 [x,z,y]`, which produces a sequential loop.

7.1.2 Histograms

The `reduce_by_index` construct ("generalised histogram") has a clever and adaptive implementation that handles multiple updates of the same bin efficiently. Its main weakness is when computing a very large histogram (many millions of bins) where only a tiny fraction of the bins are updated. This is because the main mechanism for optimising conflicts is by duplicating the histogram in memory, but this is not efficient when it is very large. If you know your program involves such a situation, it may be better to implement the histogram operation by sorting and then performing an irregular segmented reduction.

Particularly with the GPU backends, `reduce_by_index` is much faster when the operator involves a single 32-bit or 64-bit value. Even if you really want an 8-bit or 16-bit result, it may be faster to compute it with a 32-bit or 64-bit type and manually mask off the excess bits.

7.1.3 Nested parallelism

Futhark allows nested parallelism, understood as a parallel construct used inside some other parallel construct. The simplest example is nested SOACs. Example:

```
map (\xs -> reduce (+) 0 xs) xss
```

Nested parallelism is allowed and encouraged, but its compilation to efficient code is rather complicated, depending on the compiler backend that is used. The problem is that sometimes exploiting all levels of parallelism is not optimal, yet how much to exploit depends on run-time information that is not available to the compiler.

Sequential backends

The sequential backends are straightforward: all parallel operations are compiled into sequential loops. Due to Futhark's low-overhead data representation (see below), this is often surprisingly efficient.

Multicore backend

Whenever the multicore backend encounters nested parallelism, it generates two code versions: one where the nested parallel constructs are also parallelised (possibly recursively involving further nested parallelism), and one where they are turned into sequential loops. At runtime, based on the amount of work available and self-tuning heuristics, the scheduler picks the version that it believes best balances overhead with exploitation of parallelism.

GPU backends

The GPU backends handle parallelism through an elaborate program transformation called *incremental flattening*. The full details are beyond the scope of this document, but some properties are useful to know of. See this paper for more details.

The main restriction is that the GPU backends can only handle *regular* nested parallelism, meaning that the sizes of inner parallel dimensions are invariant to the outer parallel dimensions. For example, this expression contains *irregular* nested parallelism:

```
map (\i -> reduce (+) 0 (iota i)) is
```

This is because the size of the nested parallel construct is `i`, and `i` has a different value for every iteration of the outer `map`. The Futhark compiler will currently turn the irregular constructs (here, the `reduce`) into a sequential loop. Depending on how complicated the irregularity is, it may even refuse to generate code entirely.

Incremental flattening is based on generating multiple code versions to cater to different classes of datasets. At run-time, one of these versions will be picked for execution by comparing properties of the input (its size) with a *threshold parameter*. These threshold parameters have sensible defaults, but for optimal performance, they can be tuned with *futhark-autotune*.

7.2 Value Representation

The compiler discards all type abstraction when compiling. Using the module system to make a type abstract causes no run-time overhead.

7.2.1 Scalars

Scalar values (`i32`, `f64`, `bool`, etc) are represented as themselves. The internal representation does not distinguish signs, so `i32` and `u32` have the same representation, and converting between types that differ only in sign is free.

7.2.2 Tuples

Tuples are flattened and then represented directly by their individual components - there are no *tuple objects* at runtime. A function that takes an argument of type `(f64,f64)` corresponds to a C function that takes two arguments of type `double`. This has one performance implication: whenever you pass a tuple to a function, the *entire* tuple is copied (except any embedded arrays, which are always passed by reference, see below). Due to the compiler's heavy use of inlining, this is rarely a problem in practice, but it can be a concern when using the `loop` construct with a large tuple as the loop variant parameter.

7.2.3 Records

Records are turned into tuples by simply sorting their fields and discarding the labels. This means there is no overhead to using a record compared to using a tuple.

7.2.4 Sum Types

A sum type value is represented as a tuple containing all the payload components in order, prefixed with an *i8* tag to identify the constructor. For example,

```
#foo i32 bool | #bar i32
```

would be represented as a tuple of type

```
(i8, i32, bool, i32)
```

where the value

```
#foo 42 false
```

is represented as

```
(1, 42, false, 0)
```

where `#foo` is assigned the tag 1 because it is alphabetically after `#bar`.

To shrink the tuples, if multiple constructors have payload elements of the *same* type, the compiler assigns them to the same elements in the result tuple. The representation of the above sum type is actually the following:

```
(i8, i32, bool)
```

The types must be the *same* for deduplication to take place - despite *i32* and *f32* being of the same size, they cannot be assigned the same tuple element. This means that the type

```
#foo [n]i32 | #bar [n]i32
```

is efficiently represented as

```
(u8, [n]i32)
```

```
#foo [n]i32 | #bar [n]f32
```

becomes

```
(u8, [n]i32, [n]f32)
```

which is not great. Take caution when you use sum types with large arrays in their payloads.

7.2.5 Functions

Higher-order functions are implemented via defunctionalisation. At run-time, they are represented by a record containing their lexical closure. Since the type system forbids putting functions in arrays, this is essentially a constant cost, and not worth worrying about.

7.2.6 Arrays

Arrays are the only Futhark values that are boxed - that is, are stored on the heap.

The elements of an array are unboxed, stored adjacent to each other in memory. There is zero memory overhead except for the minuscule amount needed to track the shape of the array.

Multidimensional arrays

At the surface language level, Futhark may appear to support "arrays of arrays", and this is indeed a convenient aspect of its programming model, but at runtime multi-dimensional arrays are stored in flattened form. A value of type `[x][y]i32` is laid out in memory simply as one array containing *x*y* integers. This means that constructing an array `[x,y,x]` can be (relatively) expensive if `x`, `y`, `z` are themselves large arrays, as they must be copied in their entirety.

Since arrays cannot contain other arrays, memory management only has to be concerned with one level of indirection. In practice, it means that Futhark can use straightforward reference counting to keep track of when to free the memory backing an array, as circular references are not possible. Further, since arrays tend to be large and relatively few in number, the usual performance impact of naive reference counting is not present.

Arrays of tuples

For arrays of tuples, Futhark uses the so-called structure of arrays representation. In Futhark terms, an array `[n](a,b,c)` is at runtime represented as the tuple `([n]a,[n]b,[n]c)`. This means that the final memory representation always consists of arrays of scalars.

This has some significant implications. For example, `zip` and `unzip` are very cheap, as the actual runtime representation is in always "unzipped", so these functions don't actually have to do anything.

Since records and sum types are represented as tuples, this also explains how arrays of these are represented.

Element order

The exact in-memory element ordering is up to the compiler, and depends on how the array is constructed and how it is used. Absent any other information, Futhark represents multidimensional arrays in row-major order. However, depending on how the array is traversed, the compiler may insert code to represent it in some other order. For particularly tricky programs, an array may even be duplicated in memory, represented in different ways, to ensure efficient traversal. This means you should normally *not* worry about how to represent your arrays to ensure coalesced access on GPUs or similar. That is the compiler's job.

7.3 Crucial Optimisations

Some of the optimisations done by the Futhark compiler are important, complex, or subtle enough that it may be useful to know how they work, and how to write code that caters to their quirks.

7.3.1 Fusion

Futhark performs fusion of SOACs. For an expression `map f (map g A)`, then the compiler will optimise this into a single `map` with the composition of `f` and `g`, which prevents us from storing an intermediate array in memory. This is called *vertical fusion* or *producer-consumer fusion*. In this case the *producer* is `map g` and the *consumer* is `map f`.

Fusion does not depend on the expressions being adjacent as in this example, as the optimisation is performed on a data dependency graph representing the program. This means that you can decompose your programs into many small parallel operations without worrying about the overhead, as the compiler will fuse them together automatically.

Not all producer-consumer relationships between SOACs can be fused. Generally, `map` can always be fused as a producer, but `scan`, `reduce`, and similar SOACs can only act as consumers.

Horizontal fusion occurs when two SOACs take as input the same array, but are not themselves in a producer-consumer relationship. Example:

```
(map f xs, map g xs)
```

Such cases are fused into a single operation that traverses `xs` just once. More than two SOACs can be involved in horizontal fusion, and they need not be of the same kind (e.g. one could be a `map` and the other a `reduce`).

7.4 Free Operations

Some operations such as array slicing, `take`, `drop`, `transpose` and `reverse` are "free" in the sense that they merely return a different view of some underlying array. In most cases they have constant cost, no matter the size of the array they operate on. This is because they are index space transformations that simply result in different code being generated when the arrays are eventually used.

However, there are some cases where the compiler is forced to manifest such a "view" as an actual array in memory, which involves a full copy. An incomplete list follows:

- Any array returned by an entry point is converted to row-major order.
- An array returned by an `if` branch may be copied if its representation is substantially different from that of the other branch.
- An array returned by a `loop` body may be copied if its representation is substantially different from that of the initial loop values.
- An array is copied whenever it becomes the element of another multidimensional array. This is most obviously the case for array literals (`[x,y,z]`), but also for `map` expressions where the mapped function returns an array.

Note that this notion of "views" is not part of the Futhark type system - it is merely an implementation detail. Strictly speaking, all functions that return an array (e.g. `reverse`) should be considered to have a cost proportional to the size of the array, even if that cost will almost never actually be paid at run-time. If you want to be sure no copy takes place, it may be better to explicitly maintain tuples of indexes into some other array.

7.5 Small Arrays

The compiler assumes arrays are "large", which for example means that operations across them are worth parallelising. It also means they are boxed and heap-allocated, even when the size is a small constant. This can cause unexpectedly bad performance when using small constant-size arrays (say, five elements or less). Consider using tuples or records instead. This post contains more information on how and why. If in doubt, try both and measure which is faster.

7.6 Inlining

The compiler currently inlines all functions at their call site, unless they have been marked with the `noinline` attribute (see *Attributes*). This can lead to code explosion, which mostly results in slow compile times, but can also affect run-time performance. In many cases this is currently unavoidable, but sometimes the program can be rewritten such that instead of calling the same function in multiple places, it is called in a single place, in a loop. E.g. we might rewrite `f x (f y (f z v))` as:

```
loop acc = v for a in [z,y,x] do
  f a acc
```

CHAPTER

EIGHT

COMPILER ERROR INDEX

Elaboration on type errors produced by the compiler. Many error messages contain links to the sections below.

8.1 Uniqueness errors

8.1.1 "Using *x*, but this was consumed at *y*."

A core principle of uniqueness typing (see *In-place Updates*) is that after a variable is "consumed", it must not be used again. For example, this is invalid, and will result in the error above:

```
let y = x with [0] = 0
in x
```

Several operations can *consume* a variable: array update expressions, calling a function with unique-typed parameters, or passing it as the initial value of a unique-typed loop parameter. When a variable is consumed, its *aliases* are also considered consumed. Aliasing is the possibility of two variables occupying the same memory at run-time. For example, this will fail as above, because `y` and `x` are aliased:

```
let y = x
let z = y with [0] = 0
in x
```

We can always break aliasing by using a `copy` expression:

```
let y = copy x
let z = y with [0] = 0
in x
```

8.1.2 "Would consume *x*, which is not consumable"

This error message occurs for programs that try to perform a consumption (such as an in-place update) on variables that are not consumable. For example, it would occur for the following program:

```
def f (a: []i32) =
  let a[0] = a[0]+1
  in a
```

Only arrays with a a *unique array type* can be consumed. Such a type is written by prefixing the array type with an asterisk. The program could be fixed by writing it like this:

```
def f (a: *[]i32) =
  let a[0] = a[0]+1
  in a
```

Note that this places extra obligations on the caller of the `f` function, since it now *consumes* its argument. See *In-place Updates* for the full details.

You can always obtain a unique copy of an array by using `copy`:

```
def f (a: []i32) =
  let a = copy a
  let a[0] = a[0]+1
  in a
```

But note that in most cases (although not all), this subverts the purpose of using in-place updates in the first place.

8.1.3 "Unique-typed return value of *x* is aliased to *y*, which is not consumable"

This can be caused by a function like this:

```
def f (xs: []i32) : *[]i32 = xs
```

We are saying that `f` returns a *unique* array - meaning it has no aliases - but at the same time, it aliases the parameter *xs*, which is not marked as being unique (see *In-place Updates*). This violates one of the core guarantees provided by uniqueness types, namely that a unique return value does not alias any value that might be used in the future. Imagine if this was permitted, and we had a program that used `f`:

```
let b = f a
let b[0] = x
...
```

The update of `b` is fine, but if `b` was allowed to alias `a` (hence occupying the same memory), then we would be modifying `a` as well, which is a violation of referential transparency.

As with most uniqueness errors, it can be fixed by using `copy xs` to break the aliasing. We can also change the type of `f` to take a unique array as input:

```
def f (xs: *[]i32) : *[]i32 = xs
```

This makes `xs` "consumable", in the sense used by the error message.

8.1.4 "A unique-typed component of the return value of *x* is aliased to some other component"

Caused by programs like the following:

```
def main (xs: *[]i32) : (*[]i32, *[]i32) = (xs, xs)
```

While we are allowed to "consume" `xs`, as it is a unique parameter, this function is trying to return two unique values that alias each other. This violates one of the core guarantees provided by uniqueness types, namely that a unique return value does not alias any value that might be used in the future (see *In-place Updates*) - and in this case, the two values alias each other. We can fix this by inserting copies to break the aliasing:

```
def main (xs: *[]i32) : (*[]i32, *[]i32) = (xs, copy xs)
```

8.1.5 "Consuming parameter passed non-unique argument"

Caused by programs like the following:

```
def update (xs: *[]i32) = xs with [0] = 0

def f (ys: []i32) = update ys
```

The update `function` *consumes* its `xs` argument to perform an *in-place update*, as denoted by the asterisk before the type. However, the `f` function tries to pass an array that it is not allowed to consume (no asterisk before the type).

One solution is to change the type of `f` so that it also consumes its input, which allows it to pass it on to `update`:

```
def f (ys: *[]i32) = update ys
```

Another solution to `copy` the array that we pass to `update`:

```
def f (ys: []i32) = update (copy ys)
```

8.1.6 "Non-consuming higher-order parameter passed consuming argument."

This error occurs when we have a higher-order function that expects a function that does *not* consume its arguments, and we pass it one that does:

```
def apply 'a 'b (f: a -> b) (x: a) = f x

def consume (xs: *[]i32) = xs with [0] = 0

def f (arr: *[]i32) = apply consume arr
```

We can fix this by changing `consume` so that it does not have to consume its argument, by adding a `copy`:

```
def consume (xs: []i32) = copy xs with [0] = 0
```

Or we can create a variant of `apply` that accepts a consuming function:

```
def apply 'a 'b (f: *a -> b) (x: *a) = f x
```

8.1.7 "Function result aliases the free variable *x*"

Caused by definitions such as the following:

```
def x = [1,2,3]

def f () = x
```

To simplify the tracking of aliases, the Futhark type system requires that the result of a function may only alias the function parameters, not any free variables. Use `copy` to fix this:

```
def f () = copy x
```

8.1.8 "Parameter *x* refers to size *y* which will not be accessible to the caller

This happens when the size of an array parameter depends on a name that cannot be expressed in the function type:

```
def f (x: i64, y: i64) (A: [x]bool) = true
```

Intuitively, this function might have the following type:

```
val f : (x: i64, y: i64) -> [x]bool -> bool
```

But this is not currently a valid Futhark type. In a function type, each parameter can be named *as a whole*, but it cannot be taken apart in a pattern. In this case, we could fix it by splitting the tuple parameter into two separate parameters:

```
def f (x: i64) (y: i64) (A: [x]bool) = true
```

This gives the following type:

```
val f : (x: i64) -> (y: i64) -> [x]bool -> bool
```

Another workaround is to loosen the static safety, and use a size coercion to give A its expected size:

```
def f (x: i64, y: i64) (A_unsized: []bool) =
  let A = A_unsized :> [x]bool
  in true
```

This will produce a function with the following type:

```
val f [d] : (i64, i64) -> [d]bool -> bool
```

This does however lose the constraint that the size of the array must match one of the elements of the tuple, which means the program may fail at run-time.

The error is not always due to an explicit type annotation. It might also be due to size inference:

```
def f (x: i64, y: i64) (A: []bool) = zip A (iota x)
```

Here the type rules force `A` to have size `x`, leading to a problematic type. It can be fixed using the techniques above.

8.2 Size errors

8.2.1 "Size *x* unused in pattern."

Caused by expressions like this:

```
def [n] (y: i32) = x
```

And functions like this:

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

Since `n` is not the size of anything, it cannot be assigned a value at runtime. Hence this program is rejected.

8.2.2 "Causality check"

Causality check errors occur when the program is written in such a way that a size is needed before it is actually computed. See *Causality restriction* for the full rules. Contrived example:

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

Here the inner size of the array `a` must be the same as the inner size of `b`, but the inner size of `b` depends on a `filter` operation that is executed after `a` is constructed.

There are various ways to fix causality errors. In the above case, we could merely change the order of statements, such that `b` is bound first, meaning that the size is available by the time `a` is bound. In many other cases, we can lift out the "size-producing" expressions into a separate `let`-binding preceding the problematic expressions.

8.2.3 "Unknowable size *x* in parameter of *y*"

This error occurs when you define a function that can never be applied, as it requires an input of a specific size, and that size is not known. Somewhat contrived example:

```
def f (x: bool) =
  let n = if x then 10 else 20
  in \(y: [n]bool) -> ...
```

The above constructs a function that accepts an array of size 10 or 20, based on the value of `x` argument. But the type of `f true` by itself would be `?[n].[n]bool -> bool`, where the `n` is unknown. There is no way to construct an array of the right size, so the type checker rejects this program. (In a fully dependently typed language, the type would have been `[10]bool -> bool`, but Futhark does not do any type-level computation.)

In most cases, this error means you have done something you didn't actually mean to. However, in the case that that the above really is what you intend, the workaround is to make the function fully polymorphic, and then perform a size coercion to the desired size inside the function body itself:

```
def f (x: bool) =
  let n = if x then 10 else 20
  in \(y_any: []bool) ->
       let y = y_any :> [n]bool
       in true
```

This requires a check at run-time, but it is the only way to accomplish this in Futhark.

8.2.4 "Existential size would appear in function parameter of return type"

This occurs most commonly when we use function composition with one or more functions that return an *existential size*. Example:

```
filter (>0) >-> length
```

The `filter` function has this type:

```
val filter [n] 't : (t -> bool) -> [n]t -> ?[m].[m]t
```

That is, `filter` returns an array whose size is not known until the function actually returns. The `length` function has this type:

```
val length [n] 't : [n]t -> i64
```

Whenever `length` occurs (as in the composition above), the type checker must *instantiate* the `[n]` with the concrete symbolic size of its input array. But in the composition, that size does not actually exist until `filter` has been run. For that matter, the type checker does not know what >-> does, and for all it knows it may actually apply `filter` many times to different arrays, yielding different sizes. This makes it impossible to uniquely instantiate the type of `length`, and therefore the program is rejected.

The common workaround is to use *pipelining* instead of composition whenever we use functions with existential return types:

```
xs |> filter (>0) |> length
```

This works because `|>` is left-associative, and hence the `xs |> filter (>0)` part will be fully evaluated to a concrete array before `length` is reached.

We can of course also write it as `length (filter (>0) xs)`, with no use of either pipelining or composition.

8.2.5 "Existential size *n* not used as array size"

This error occurs for type expressions that use explicit existential quantification in an incorrect way, such as the following examples:

```
?[n].bool

?[n].bool -> [n]bool
```

When we use existential quantification, we are required to use the size within its scope, *and* it must not exclusively be used to the right of function arrow.

To understand the motivation behind this rule, consider that when we use an existential quantifier we are saying that there is *some size*, it just cannot be known statically, but must be read from some value (i.e. array) at runtime. In the first example above, the existential size `n` is not used at all, so the actual value cannot be determined at runtime. In the second example, while an array `[n]bool` does exist, it is part of a function type, and at runtime functions are black boxes and don't "carry" the size of their parameter or result types.

The workaround is to actually use the existential size. This can be as simple as adding a *witness array* of type `[n]()`:

```
?[n].([n](),bool)

?[n].([n](), bool -> [n]bool)
```

Such an array will take up no space at runtime.

8.2.6 "Parameter *x* used as size would go out of scope."

This error tends to happen when higher-order functions are used in a way that causes a size requirement to become impossible to express. Real programs that encounter this issue tend to be complicated, but to illustrate the problem, consider the following contrived function:

```
def f (n: i64) (m: i64) (b: [n][m]bool) = b[0,0]
```

We have the following type:

```
val f : (n: i64) -> (m: i64) -> (b: [n][m]bool) -> bool
```

Now suppose we say:

```
def g = uncurry f
```

What should be the type of `g`? Intuitively, something like this:

```
val g : (n: i64, m: i64) -> (b: [n][m]bool) -> bool
```

But this is *not* expressible in the Futhark type system - and even if it were, it would not be easy to infer this in general, as it depends on exactly what `uncurry` does, which the type checker does not know.

As a workaround, we can use explicit type annotation and size coercions to give `g` an acceptable type:

```
def g [a][b] (n,m) (b: [a][b]bool) = f n m (b :> [n][m]bool)
```

Another workaround, which is often the right one in cases not as contrived as above, is to modify `f` itself to produce a *witness* of the constraint, in the form of an array of shape `[n][m]`:

```
def f (n: i64) (m: i64) : ([n][m](), [n][m]bool -> bool) =
  (replicate n (replicate m ()), \b -> b[0,0])
```

Then `uncurry f` works just fine and has the following type:

```
(i64, i64) -> ?[n][m].([n][m](), [n][m]bool -> bool)
```

Programming with such *explicit size witnesses* is a fairly advanced technique, but often necessary when writing advanced size-dependent code.

8.2.7 "Ambiguous size *x*"

There are various sources for this error, but they all have the same ultimate cause: the type checker cannot figure out how some symbolic size name should be resolved to a concrete size. The simplest example, although contrived, is probably this:

```
let [n][m] (xss: [n][m]i64) = []
```

The type checker can infer that `n` should be zero, but how can it possibly figure out the shape of the (non-existent) rows of the two-dimensional array? This can be fixed in many ways, but adding a type ascription to the array is one of them: `[] : [0][2]i64`.

Another common case arises when using holes. For an expression `length ???`, how would the type checker figure out the intended size of the array that the hole represents? Again, this can be solved with a type ascription: `length (??? : [10]bool)`.

Finally, ambiguous sizes can also occur for functions that use size parameters only in "non-witnessing" position, meaning sizes that are not actually uses as sizes of real arrays. An example:

```
def f [n] (g: [n]i64 -> i64) : i64 = n

def main = f (\xs -> xs[0])
```

Note that `f` is a higher order function, and that the size parameter `n` is only used in the type of the `g` function. Futhark's value model is such that given a value of type `[n]i64 -> i64`, we cannot extract an `n` from it. Using a function such as `f` is only valid when `n` can be inferred from the usage, which is not the case here. Again, we can fix it by adding a type ascription to disambiguate:

```
def main = f (\(xs:[1]i64) -> xs[0])
```

8.3 Module errors

8.3.1 "Entry points may not be declared inside modules."

This occurs when the program uses the `entry` keyword inside a module:

```
module m = {
  entry f x = x + 1
}
```

Entry points can only be declared at the top level of a file. When we wish to make a function from inside a module available as an entry point, we must define a wrapper function:

```
module m = {
  def f x = x + 1
}

entry f = m.f
```

8.3.2 "Module *x* is a parametric module

A parametric module is a module-level function:

```
module PM (P: {val x : i64}) = {
  def y = x + 2
}
```

If we directly try to access the component of `PM`, as `PM.y`, we will get an error. To use `PM` we must first apply it to a module of the expected type:

```
module M = PM { val x = 2 : i64 }
```

Now we can say `M.y`. See *Modules* for more.

8.4 Other errors

8.4.1 "Literal out of bounds"

This occurs for overloaded constants such as `1234` that are inferred by context to have a type that is too narrow for their value. Example:

```
257 : u8
```

It is not an error to have a *non-overloaded* numeric constant whose value is too large for its type. The following is perfectly cromulent:

```
257u8
```

In such cases, the behaviour is overflow (so this is equivalent to `1u8`).

8.4.2 "Type is ambiguous"

There are various cases where the type checker is unable to infer the full type of something. For example:

```
def f r = r.x
```

We know that `r` must be a record with a field called `x`, but maybe the record could also have other fields as well. Instead of assuming a perhaps too narrow type, the type checker signals an error. The solution is always to add a type annotation in one or more places to disambiguate the type:

```
def f (r: {x:bool, y:i32}) = r.x
```

Usually the best spot to add such an annotation is on a function parameter, as above. But for ambiguous sum types, we often have to put it on the return type. Consider:

```
def f (x: bool) = #some x
```

The type of this function is ambiguous, because the type checker must know what other possible contructors (apart from `#some`) are possible. We fix it with a type annotation on the return type:

```
def f (x: bool) : (#some bool | #none) = #just x
```

See *Type Abbreviations* for how to avoid typing long types in several places.

8.4.3 "The *x* operator may not be redefined"

The `&&` and `||` operators have magical short-circuiting behaviour, and therefore may not be redefined. There is no way to define your own short-circuiting operators.

8.4.4 "Unmatched cases in match expression"

Futhark requires `match` expressions to be *exhaustive* - that is, cover all possible forms of the value being matched. Example:

```
def f (x: i32) =
  match x case 0 -> false
          case 1 -> true
```

Usually this is an actual bug, and you fix it by adding the missing cases. But sometimes you *know* that the missing cases will never actually occur at run-time. To satisfy the type checker, you can turn the final case into a wildcard that matches anything:

```
def f (x: i32) =
  match x case 0 -> false
          case _ -> true
```

Alternatively, you can add a wildcard case that explicitly asserts that it should never happen:

```
def f (x: i32) =
  match x case 0 -> false
          case 1 -> true
          case _ -> assert false false
```

See here for details on how to use `assert`.

8.4.5 "Full type of *x* is not known at this point"

When performing a *record update*, the type of the field we are updating must be known. This restriction is based on a limitation in the type type checker, so the notion of "known" is a bit subtle:

```
def f r : {x:i32} = r with x = 0
```

Even though the return type annotation disambiguates the type, this program still fails to type check. This is because the return type is not consulted until *after* the body of the function has been checked. The solution is to put a type annotation on the parameter instead:

```
def f (r : {x:i32}) = r with x = 0
```

CHAPTER

NINE

SERVER PROTOCOL

A Futhark program can be compiled to a *server executable*. Such a server maintains a Futhark context and presents a line-oriented interface (over stdin/stdout) for loading and dumping values, as well as calling the entry points in the program. The main advantage over the plain executable interface is that program initialisation is done only *once*, and we can work with opaque values.

The server interface is not intended for human consumption, but is useful for writing tools on top of Futhark programs, without having to use the C API. Futhark's built-in benchmarking and testing tools use server executables.

A server executable is started like any other executable, and supports most of the same command line options (*Executable Options*).

9.1 Basics

Each command is sent as a *single line* on standard input. A command consists of space-separated *words*. A word is either a sequence of non-space characters (`foo`), *or* double quotes surrounding a sequence of non-newline and non-quote characters (`"foo bar"`).

The response is sent on standard output. The server will print `%%% OK` on a line by itself to indicate that a command has finished. It will also print `%%% OK` at startup once initialisation has finished. If initialisation fails, the process will terminate. If a command fails, the server will print `%%% FAILURE` followed by the error message, and then `%%% OK` when it is ready for more input. Some output may also precede `%%% FAILURE`, e.g. logging statements that occured before failure was detected. Fatal errors that lead to server shutdown may be printed to stderr.

9.2 Variables

Some commands produce or read variables. A variable is a mapping from a name to a Futhark value. Values can be both transparent (arrays and primitives), but they can also be *opaque* values. These can be produced by entry points and passed to other entry points, but cannot be directly inspected.

9.3 Types

All variables have types, and all entry points accept inputs and produce outputs of defined types. The notion of transparent and opaque types are the same as in the C API: primitives and array of primitives are directly supported, and everything else is treated as opaque. See also *Value Mapping*. When printed, types follow basic Futhark type syntax *without* sizes (e.g. `[][]i32`). Uniqueness is not part of the types, but is indicated with an asterisk in the `inputs` and `outputs` commands (see below).

9.4 Consumption and aliasing

Since the server protocol closely models the C API, the same rules apply to entry points that consume their arguments (see *Consumption and Aliasing*). In particular, consumed variables must still be freed with the `free` command - but this is the only operation that may be used on consumed variables.

9.5 Commands

The following commands are supported.

9.5.1 General Commands

`types`

Print the names of available types, one per line.

`entry_points`

Print the names of available entry points.

`call` *entry o1 ... oN i1 ... iM*

Call the given entry point with input from the variables *i1* to *iM*. The results are stored in *o1* to *oN*, which must not already exist.

`restore` *file v1 t1 ... vN tN*

Load *N* values from *file* and store them in the variables *v1* to *vN* of types *t1* to *tN*, which must not already exist.

`store` *file v1 ... vN*

Store the *N* values in variables *v1* to *vN* in *file*.

`free` *v1 ... vN*

Delete the given variables.

`rename` *oldname newname*

Rename the variable *oldname* to *newname*, which must not already exist.

`inputs` *entry*

Print the types of inputs accepted by the given entry point, one per line. If the given input is consumed, the type is prefixed by *.

`outputs` *entry*

Print the types of outputs produced by the given entry point, one per line. If the given output is guaranteed to be unique (does not alias any inputs), the type is prefixed by *.

`clear`

Clear all internal caches and counters maintained by the Futhark context. Corresponds to *futhark_context_clear_caches()*.

`pause_profiling`

Corresponds to *futhark_context_pause_profiling()*.

`unpause_profiling`

Corresponds to *futhark_context_unpause_profiling()*.

`report`

Corresponds to *futhark_context_report()*.

`set_tuning_param`

Corresponds to *`futhark_context_config_set_tuning_param()`*.

9.5.2 Record Commands

`fields` *type*

If the given type is a record, print a line for each field of the record. The line will contain the name of the field, followed by a space, followed by the type of the field. Note that the type name can contain spaces. The order of fields is significant, as it is the one expected by the `new_record` command.

`new` *v0 type v1 ... vN*

Create a new variable *v0* of type *type*, which must be a record type with *N* fields, where *v1* to *vN* are variables with the corresponding field types (the expected order is given by the `fields` command).

`project` *to from field*

Create a new variable *to* whose value is the field *field* of the record-typed variable *from*.

9.6 Environment Variables

9.6.1 `FUTHARK_COMPILER_DEBUGGING`

Turns on debugging output for the server when set to 1.

CHAPTER

TEN

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.

10.1 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.

10.2 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.

10.3 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.

10.4 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
  let v = 1i32 in
```

(continues on next page)

(continued from previous page)

```
  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.

10.5 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

ELEVEN

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 such as Haskell and 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*.

11.1 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 `def`. Local variables are bound with `let`.

11.2 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 `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.

11.3 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 *Modules*.

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 type system, type annotations are sometimes 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 written with the usual `a -> b` notation, 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:

```
def 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:

```
def 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:

```
def 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

TWELVE

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 compiled programs. 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.

12.1 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> <values...>
```

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 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 row-major order in their binary little endian representation:

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

12.1.1 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"
" f16"
" f32"
" f64"
"bool"
```

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

Values of type `bool` are encoded with a byte each. The results are undefined if this byte is not either 0 or 1.

CHAPTER

THIRTEEN

FUTHARK

13.1 SYNOPSIS

futhark <subcommand> options...

13.2 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 below.

13.3 COMMANDS

13.3.1 futhark check [-w] PROGRAM

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

13.3.2 futhark check-syntax PROGRAM

Check whether a Futhark program is syntactically correct.

13.3.3 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.

13.3.4 futhark dataget PROGRAM DATASET

Find the test dataset whose description contains `DATASET` (e.g. `#1`) and print it in binary representation to standard output. This does not work for `script` datasets.

13.3.5 futhark defs PROGRAM

Print names and locations of every top-level definition in the program (including top levels of modules), one per line. The program need not be type-correct, but it must not contain syntax errors.

13.3.6 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.

13.3.7 futhark hash PROGRAM

Print a hexadecimal hash of the program AST, including all non-builtin imports. Supposed to be invariant to whitespace changes.

13.3.8 futhark imports PROGRAM

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

13.3.9 futhark lsp

Run an LSP (Language Server Protocol) server for Futhark that communicates on standard input. There is no reason to run this by hand. It is used by LSP clients to provide editor features.

13.3.10 futhark query PROGRAM LINE COL

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

13.3.11 futhark thanks

Expresses gratitude.

13.3.12 futhark tokens FILE

Print the tokens the given Futhark source file; one per line.

13.4 SEE ALSO

futhark-opencl, *futhark-c*, *futhark-python*, *futhark-pyopencl*, *futhark-wasm*, *futhark-wasm-multicore*, *futhark-ispc*, *futhark-dataset*, *futhark-doc*, *futhark-test*, *futhark-bench*, *futhark-run*, *futhark-repl*, *futhark-literate*

CHAPTER

FOURTEEN

FUTHARK-AUTOTUNE

14.1 SYNOPSIS

futhark autotune [options...] <program.fut>

14.2 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*.

14.3 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.

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

--spec-file=FILE Ignore the test specification in the program file(s), and instead load them from this other file. These external test specifications use the same syntax as normal, but *without* line comment prefixes A == is still expected.

--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.

14.4 SEE ALSO

futhark-bench

CHAPTER

FIFTEEN

FUTHARK-BENCH

15.1 SYNOPSIS

futhark bench [options…] programs…

15.2 DESCRIPTION

This tool is the recommended way to benchmark Futhark programs. Programs are compiled using the specified backend (`c` by default), then run a number of times for each test case, and the arithmetic mean runtime and 95% confidence interval 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.

15.3 METHODOLOGY

For each program and dataset, `futhark bench` first does a single "warmup" run that is discarded. After that it uses a two-phase technique.

1. The *initial phase* performs ten runs (change with `-r`), or perform runs for at least half a second, whichever takes longer. If the resulting measurements are sufficiently statistically robust (determined using standard deviation and autocorrelation metrics), the results are produced and the second phase is not entered. Otherwise, the results are discarded and the second phase entered.
2. The *convergence phase* keeps performing runs until a measurement of sufficient statistical quality is reached.

The notion of "sufficient statistical quality" is based on heuristics. The intent is that `futhark bench` will in most cases do *the right thing* by default, both when benchmarking both long-running programs and short-running programs. If you want complete control, disable the convergence phase with `--no-convergence-phase` and set the number of runs you want with `-r`.

15.4 OPTIONS

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

--cache-extension=EXTENSION For a program `foo.fut`, pass `--cache-file foo.fut.EXTENSION`. By default, `--cache-file` is not passed.

--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.

--convergence-max-seconds=NUM Don't run the convergence phase for longer than this. This does not mean that the measurements have converged. Defaults to 300 seconds (five minutes).

--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", "disable", or "no_foo" (where *foo* is the backend used) 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.

--no-convergence-phase Do not run the convergence phase.

--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-pyopencl* on some platforms), 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.

--spec-file=FILE Ignore the test specification in the program file(s), and instead load them from this other file. These external test specifications use the same syntax as normal, but *without* line comment prefixes. A == is still expected.

--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.

15.5 EXAMPLES

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

```
-- How quickly can we reduce arrays?
--
-- ==
-- nobench input { 0i64 }
-- output { 0i64 }
-- input { 100i64 }
-- output { 4950i64 }
-- compiled input { 10000i64 }
-- output { 49995000i64 }
-- compiled input { 1000000i64 }
-- output { 499999500000i64 }

let main(n: i64): i64 =
  reduce (+) 0 (iota n)
```

15.6 SEE ALSO

futhark-c, *futhark-test*

CHAPTER

SIXTEEN

FUTHARK-C

16.1 SYNOPSIS

futhark c [options...] <program.fut>

16.2 DESCRIPTION

`futhark c` translates a Futhark program to sequential C code, and either compiles that C code with a C compiler (see below) 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

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.

16.3 OPTIONS

-h Print help text to standard output and exit.

--entry-point NAME Treat this top-level function as an entry point.

--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.

--server Generate a server-mode executable that reads commands from stdin.

-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.

16.4 ENVIRONMENT VARIABLES

`CC`

The C compiler used to compile the program. Defaults to `cc` if unset.

`CFLAGS`

Space-separated list of options passed to the C compiler. Defaults to `-O3 -std=c99` if unset.

16.5 EXECUTABLE OPTIONS

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

-h, --help Print help text to standard output and exit.

-b, --binary-output Print the program result in the binary output format. The default is human-readable text, which is very slow. Not accepted by server-mode executables.

--cache-file=FILE Store any reusable initialisation data in this file, possibly speeding up subsequent launches.

-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`. Not accepted by server-mode executables.

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

-n, --no-print-result Do not print the program result. Not accepted by server-mode executables.

-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. Not accepted by server-mode executables.

-t, --write-runtime-to=FILE Print the time taken to execute the program to the indicated file, an integral number of microseconds. Not accepted by server-mode executables.

16.6 SEE ALSO

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

CHAPTER

SEVENTEEN

FUTHARK-CUDA

17.1 SYNOPSIS

futhark cuda [options...] <program.fut>

17.2 DESCRIPTION

`futhark cuda` translates a Futhark program to C code invoking CUDA kernels, and either compiles that C code with a C compiler 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.

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

The generated CUDA code can be called from multiple CPU threads, as it brackets every API operation with `cuCtxPushCurrent()` and `cuCtxPopCurrent()`.

17.3 OPTIONS

Accepts the same options as *futhark-c*.

17.4 ENVIRONMENT VARIABLES

`CC`

The C compiler used to compile the program. Defaults to `cc` if unset.

`CFLAGS`

Space-separated list of options passed to the C compiler. Defaults to `-O -std=c99` if unset.

17.5 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.

-h, --help Print help text to standard output and exit.

--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.

-n, --no-print-result Do not print the program result.

--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.

--param=ASSIGNMENT Set a tuning parameter to the given value. `ASSIGNMENT` must be of the form `NAME=INT` Use `--print-params` to see which names are available.

--print-params Print all tuning parameters that can be set with `--param` or `--tuning`.

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

17.6 ENVIRONMENT

If run without `--library`, `futhark cuda` will invoke a C compiler to compile the generated C program into a binary. This only works if the C compiler can find the necessary CUDA libraries. On most systems, CUDA is installed in `/usr/local/cuda`, which is usually not part of the default compiler 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
```

At runtime the generated program must be able to find the CUDA installation directory, which is normally located at `/usr/local/cuda`. If you have CUDA installed elsewhere, set any of the `CUDA_HOME`, `CUDA_ROOT`, or `CUDA_PATH` environment variables to the proper directory.

17.7 SEE ALSO

futhark-opencl

CHAPTER

EIGHTEEN

FUTHARK-DATASET

18.1 SYNOPSIS

futhark dataset [options...]

18.2 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.

Returns a nonzero exit code if it fails to write the full output.

18.3 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.

18.4 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 --i64-bounds=0:9 -g [100]i64
```

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
```

18.5 SEE ALSO

futhark-test, *futhark-bench*

CHAPTER

NINETEEN

FUTHARK-DOC

19.1 SYNOPSIS

futhark doc [options...] dir

19.2 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.

19.3 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.

19.4 EXAMPLES

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

19.5 SEE ALSO

futhark-test, *futhark-bench*

CHAPTER

TWENTY

FUTHARK-LITERATE

20.1 SYNOPSIS

futhark literate [options...] program

20.2 DESCRIPTION

The command `futhark literate foo.fut` will compile the given program and then generate a Markdown file `foo.md` that contains a prettyprinted form of the program. This is useful for demonstrating programming techniques.

- Top-level comments that start with a line comment marker (`--`) and a space in the next column will be turned into ordinary text in the Markdown file.
- Ordinary top-level definitions will be enclosed in Markdown code blocks.
- Any *directives* will be executed and replaced with their output. See below.

Warning: Do not run untrusted programs. See SAFETY below.

Image directives and builtin functions shell out to `convert` (from ImageMagick). Video generation uses `ffmpeg`.

For an input file `foo.fut`, all generated files will be in a directory named `foo-img`. A `file` parameter passed to a directive may not contain a directory component or spaces.

20.3 OPTIONS

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

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

--output=FILE Override the default output file. The image directory will be set to the provided `FILE` with its extension stripped and `-img/` appended.

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

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

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

--stop-on-error Terminate immediately without producing an output file if a directive fails. Otherwise a file will still be produced, and failing directives will be followed by an error message.

-v, --verbose Print verbose information on stderr about directives as they are executing. This is also needed to see `#[trace]` output.

20.4 DIRECTIVES

A directive is a way to show the result of running a function. Depending on the directive, this can be as simple as printing the textual representation of the result, or as complex as running an external plotting program and referencing a generated image.

Any directives that produce images for a program `foo.fut` will place them in the directory `foo-img/`. If this directory already exists, it will be deleted.

A directive is a line starting with `-- >`, which must follow an empty line. Arguments to the directive follow on the remainder of the line. Any expression arguments are given in a very restricted subset of Futhark called *FutharkScript* (see below).

Some directives take mandatory or optional parameters. These are entered after a semicolon *and a linebreak*.

The following directives are supported:

- `> e`

  Shows the result of executing the FutharkScript expression `e`, which can have any (transparent) type.

- `> :video e[; parameters...]`

  Creates a video from `e`. The optional parameters are lines of the form *key: value*:

  - `repeat: <true|false>`
  - `fps: <int>`
  - `format: <webm|gif>`
  - `file: <name>`. Make sure to provide a proper extension.

  `e` must be one of the following:

  - A 3D array where the 2D elements is of a type acceptable to `:img`, and the outermost dimension is the number of frames.
  - A triple `(s -> (img,s), s, i64)`, for some types `s` and `img`, where `img` is an array acceptable to `:img`. This means not all frames have to be held in memory at once.

- `> :brief <directive>`

  The same as the given *directive* (which must not start with another >), but suppress parameters when printing it.

- `> :covert <directive>`

  The same as the given *directive* (which must not start with another >), but do not show the directive itself in the output, only its result.

- `> :img e[; parameters...]`

  Visualises `e`. The optional parameters are lines of the form *key: value*:

  - `file:  NAME`. Make sure to use a proper extension.

  The expression `e` must have one of the following types:

  - `[][]i32` and `[][]u32`

    Interpreted as ARGB pixel values.

  - `[][]f32` and `[][]f64`

    Interpreted as greyscale. Values should be between 0 and 1, with 0 being black and 1 being white.

  - `[][]u8`

    Interpreted as greyscale. 0 is black and 255 is white.

  - `[][]bool`

    Interpreted as black and white. `false` is black and `true` is white.

- `> :plot2d e[; size=(height,width)]`

  Shows a plot generated with `gnuplot` of `e`, which must be an expression of type `([]t, []t)`, where `t` is some numeric type. The two arrays must have the same length and are interpreted as `x` and `y` values, respectively.

  The expression may also be a record expression (*not* merely the name of a Futhark variable of record type), where each field will be plotted separately and must have the type mentioned above.

- `> :gnuplot e; script...`

  Similar to `plot2d`, except that it uses the provided Gnuplot script. The `e` argument must be a record whose fields are tuples of one-dimensional arrays, and the data will be available in temporary files whose names are in variables named after the record fields. Each file will contain a column of data for each array in the corresponding tuple.

  Use `set term png size width,height` to change the size to `width` by `height` pixels.

20.5 FUTHARKSCRIPT

Only an extremely limited subset of Futhark is supported:

```
script_exp    ::=   fun script_exp*
                  | "(" script_exp ")"
                  | "(" script_exp ( "," script_exp )+ ")"
                  | "[" script_exp ( "," script_exp )+ "]"
                  | "empty" "(" ("[" decimal "]" )+ script_type ")"
                  | "{" "}"
                  | "{" (id = script_exp) ("," id = script_exp)* "}"
                  | "let" script_pat "=" script_exp "in" script_exp
                  | literal
script_pat    ::=   id | "(" id ("," id) ")"
script_fun    ::=   id | "$" id
script_type   ::=   int_type | float_type | "bool"
```

Note that empty arrays must be written using the `empty(t)` notation, e.g. `empty([0]i32)`.

Function applications are either of Futhark functions or *builtin functions*. The latter are prefixed with `$` and are magical (usually impure) functions that could not possibly be implemented in Futhark. The following builtins are supported:

- `$loadimg "file"` reads an image from the given file and returns it as a row-major `[][]u32` array with each

pixel encoded as ARGB.

- `$loaddata "file"` reads a dataset from the given file. When the file contains a singular value, it is returned as value. Otherwise, a tuple of values is returned, which should be destructured before use. For example: `let (a, b) = $loaddata "foo.in" in bar a b`.

20.6 SAFETY

Some directives (e.g. `:gnuplot`) can run arbitrary shell commands. Other directives or builtin functions can read or write arbitrary files. Running an untrusted literate Futhark program is as dangerous as running a shell script you downloaded off the Internet. Before running a program from an unknown source, you should always give it a quick read to see if anything looks fishy.

20.7 BUGS

FutharkScript expressions can only refer to names defined in the file passed to `futhark literate`, not any names in imported files.

20.8 SEE ALSO

futhark-test, *futhark-bench*

CHAPTER

TWENTYONE

FUTHARK-MULTICORE

21.1 SYNOPSIS

futhark multicore [options…] <program.fut>

21.2 DESCRIPTION

`futhark multicore` translates a Futhark program to multithreaded C code, and either compiles that C code with a C compiler 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.

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.

21.3 OPTIONS

Accepts the same options as *futhark-c*.

21.4 ENVIRONMENT VARIABLES

`CC`

: The C compiler used to compile the program. Defaults to `cc` if unset.

`CFLAGS`

: Space-separated list of options passed to the C compiler. Defaults to `-O3 -std=c99 -pthread` if unset.

21.5 EXECUTABLE OPTIONS

Generated executables accept the same options as those generated by *futhark-c*. The following additional options are accepted.

-P, --profile Collect and report various bits of profiling information.

--num-threads=INT Use this many physical threads.

21.6 BUGS

Currently works only on Unix-like systems because of a dependency on pthreads. Adding support for Windows would likely not be difficult.

21.7 SEE ALSO

futhark-c, *futhark-test*

CHAPTER

TWENTYTWO

FUTHARK-ISPC

22.1 SYNOPSIS

futhark ispc [options…] <program.fut>

22.2 DESCRIPTION

`futhark ispc` translates a Futhark program to a combination of C and ISPC code, with ISPC used for parallel loops. It otherwise operates similarly to *futhark-multicore*. You need to have `ispc` on your `PATH`.

22.3 OPTIONS

Accepts the same options as *futhark-multicore*.

22.4 ENVIRONMENT VARIABLES

`CC`

The C compiler used to compile the program. Defaults to `cc` if unset.

`CFLAGS`

Space-separated list of options passed to the C compiler. Defaults to `-O3 -std=c99 -pthread` if unset.

`ISPCFLAGS`

Space-separated list of options passed to `ispc`. Defaults to `-O3 --woff` if unset.

22.5 EXECUTABLE OPTIONS

Generated executables accept the same options as those generated by *futhark-multicore*.

22.6 BUGS

Currently works only on Unix-like systems because of a dependency on pthreads. Adding support for Windows would likely not be difficult.

22.7 SEE ALSO

futhark-multicore, *futhark-test*

CHAPTER

TWENTYTHREE

FUTHARK-OPENCL

23.1 SYNOPSIS

futhark opencl [options…] <program.fut>

23.2 DESCRIPTION

`futhark opencl` translates a Futhark program to C code invoking OpenCL kernels, and either compiles that C code with a C compiler 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.

`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.

23.3 OPTIONS

Accepts the same options as *futhark-c*.

23.4 ENVIRONMENT VARIABLES

`CC`

The C compiler used to compile the program. Defaults to `cc` if unset.

`CFLAGS`

Space-separated list of options passed to the C compiler. Defaults to `-O -std=c99` if unset.

23.5 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.

-h, --help Print help text to standard output and exit.

--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.

-n, --no-print-result Do not print the program result.

-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.

-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`.

--param=ASSIGNMENT Set a tuning parameter to the given value. `ASSIGNMENT` must be of the form `NAME=INT` Use `--print-params` to see which names are available.

--print-params Print all tuning parameters that can be set with `--param` or `--tuning`.

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

--list-devices List all OpenCL devices and platforms available on the system.

23.6 SEE ALSO

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

CHAPTER

TWENTYFOUR

FUTHARK-PKG

24.1 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

24.2 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`.

24.3 COMMANDS

24.3.1 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.

24.3.2 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.

24.3.3 futhark pkg init PKGPATH

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

24.3.4 futhark pkg fmt

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

24.3.5 futhark pkg remove PKGPATH

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

24.3.6 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.

24.3.7 futhark pkg upgrade

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

24.3.8 futhark pkg versions PKGPATH

Print all available versions for the given package path.

24.4 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.

24.5 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.)

24.6 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.

24.7 SEE ALSO

futhark-test, *futhark-doc*

CHAPTER

TWENTYFIVE

FUTHARK-PYOPENCL

25.1 SYNOPSIS

futhark pyopencl [options...] infile

25.2 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.

25.3 OPTIONS

Accepts the same options as *futhark-opencl*.

25.4 SEE ALSO

futhark-python, *futhark-opencl*

CHAPTER

TWENTYSIX

FUTHARK-PYTHON

26.1 SYNOPSIS

futhark python [options…] infile

26.2 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*.

26.3 OPTIONS

Accepts the same options as *futhark-c*.

26.4 SEE ALSO

futhark-pyopencl

CHAPTER

TWENTYSEVEN

FUTHARK-REPL

27.1 SYNOPSIS

futhark repl [program.fut]

27.2 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 the `#[trace]` and `#[break]` attributes. See *futhark-run* for a description.

27.3 OPTIONS

-h Print help text to standard output and exit.

-V Print version information on standard output and exit.

27.4 SEE ALSO

futhark-run, *futhark-test*

CHAPTER

TWENTYEIGHT

FUTHARK-RUN

28.1 SYNOPSIS

futhark run [options…] <program.fut>

28.2 DESCRIPTION

Execute the given program by evaluating an entry point (`main` by default) 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. The `#[trace]` and `#[break]` attributes are fully supported in the interpreter. Tracing prints values to stdout in contrast to compiled code, which prints to stderr.

28.3 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.

28.4 SEE ALSO

futhark-repl, *futhark-test*

CHAPTER

TWENTYNINE

FUTHARK-TEST

29.1 SYNOPSIS

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

29.2 DESCRIPTION

Test Futhark programs based on input/output datasets. All contained `.fut` files within a given directory are considered. By default, tests are carried out with compiled code. This can be changed with the `-i` option.

A Futhark test program is an ordinary Futhark program, with at least one test block describing input/output test cases and possibly other options. The last line must end in a newline. 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|script] 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), and floating-point constants (always with type suffix) are also permitted.

If `input` is preceded by `script`, the text between the curly braces is interpreted as a FutharkScript expression (see *futhark-literate*), which is executed to generate the input. It must use only functions explicitly declared as entry points. If the expression produces an n-element tuple, it will be unpacked and its components passed as n distinct arguments to the test function.

If `input` is followed by an `@` and a file name (which must not contain any whitespace) instead of curly braces, values or FutharkScript expression 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. This currently does not work for `script` inputs.

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.

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.

29.3 OPTIONS

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

--cache-extension=EXTENSION For a program `foo.fut`, pass `--cache-file foo.fut.EXTENSION`. By default, `--cache-file` is not passed.

-c Only run compiled code - do not run the interpreter. This is the default.

-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, as are cases marked "no_foo", where *foo* is the backend used.

-i Test with the interpreter.

-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-pyopencl* on some platforms), 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.

29.4 ENVIRONMENT VARIABLES

`TMPDIR`

Directory used for temporary files such as gunzipped datasets and log files.

29.5 EXAMPLES

The following program tests simple indexing and bounds checking:

```
-- Test simple indexing of an array.
-- ==
-- tags { firsttag secondtag }
-- input { [4,3,2,1] 1i64 }
-- output { 3 }
-- input { [4,3,2,1] 5i64 }
-- error: Error*

let main (a: []i32) (i: i64): i32 =
  a[i]
```

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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)
```

29.6 SEE ALSO

futhark-bench, *futhark-repl*

CHAPTER

THIRTY

FUTHARK-WASM

30.1 SYNOPSIS

futhark wasm [options…] <program.fut>

30.2 DESCRIPTION

`futhark wasm` translates a Futhark program to sequential WebAssembly code by first generating C as `futhark c`, and then using Emscripten (`emcc`). This produces a `.js` file that allows the compiled code to be invoked from JavaScript. Executables implement the Futhark server protocol and can be run with Node.js.

30.3 OPTIONS

Accepts the same options as *futhark-c*.

30.4 ENVIRONMENT VARIABLES

`CFLAGS`

Space-separated list of options passed to `emcc`. Defaults to `-O3 -std=c99` if unset.

`EMCFLAGS`

Space-separated list of options passed to `emcc`.

30.5 EXECUTABLE OPTIONS

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

-h, --help Print help text to standard output and exit.

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

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

30.6 SEE ALSO

futhark-c, *futhark-wasm-multicore*

CHAPTER

THIRTYONE

FUTHARK-WASM-MULTICORE

31.1 SYNOPSIS

futhark wasm-multicore [options…] <program.fut>

31.2 DESCRIPTION

`futhark wasm-multicore` translates a Futhark program to multi-threaded WebAssembly code by first generating C as `futhark c`, and then using Emscripten (`emcc`). This produces a `.js` file that allows the compiled code to be invoked from JavaScript. Executables implement the Futhark server protocol and can be run with Node.js.

31.3 OPTIONS

Accepts the same options as *futhark-c*.

31.4 ENVIRONMENT VARIABLES

Respects the same environment variables as *futhark-wasm*.

31.5 EXECUTABLE OPTIONS

Generated executables accept the same options as those generated by *futhark-wasm*.

31.6 SEE ALSO

futhark-c, *futhark-wasm*

INDEX

F

N