PSyclone Documentation

Release 2.5.0

Andrew Coughtrie, Rupert Ford, Joerg Henrichs, Iva Kavcic, Andrew Porter, Sergi Siso and Joseph Wallwork

Nov 27, 2024

TABLE OF CONTENTS

PSyclone is a source-to-source Fortran compiler designed to programmatically optimise, parallelise and instrument HPC applications via user-provided transformation scripts.

By encapsulating the performance-portability aspects (e.g. whether to parallelise with OpenMP or OpenACC), these scripts enable a separation of concerns between the scientific implementation and the optimisation choices. This allows each aspect to be explored and developed largely independently. Additionally, PSyclone supports the development of kernel-based, Fortran-embedded DSLs following the PSyKAl model developed in the GungHo project.

PSyclone is currently used to support the LFRic mixed finite-element PSyKAl DSL for the UK MetOffice's next generation modelling system and the GOcean finite-difference PSyKAl DSL for a prototype 2D ocean modelling system. It is also used to insert GPU offloading directives into existing directly-addressed MPI applications such as the NEMO ocean model.

More detailed implementation information is available in the Developer Guide and the Reference Guide.

CHAPTER

ONE

GETTING GOING

1.1 Installation

The following instructions are intended for a PSyclone user who wants to work with a released version of the code. If you are a developer or wish to test a specific branch of PSyclone from the GitHub repository please see the Installation section in the Developer Guide.

From PyPI:

For a system-wide installation use:

```
pip install psyclone
```

For a user-local installation use:

```
pip install --user psyclone
```

For a specific release (where `X.Y.Z` is the release version) use:

```
pip install psyclone==X.Y.Z
```

For more information about using `pip` or encapsulating the installation in its own `virtual environment` we recommend reading the Python Packaging User Guide.

From Spack:

To install psyclone to your loaded Spack installation use:

```
spack install psyclone
```

For more information about how to use Spack we recommend reading the Spack documentation.

From Source:

To download and install a specific PSyclone release (where `X.Y.Z` is the release version) from source, use:

```
wget https://github.com/stfc/PSyclone/archive/X.Y.Z.tar.gz
tar zxf X.Y.Z.tar.gz
cd PSyclone-X.Y.Z
pip install .
```

1.1.1 Installation location

The PSyclone installation location will vary depending on the specific installation method and options used. The `psyclone` command will typically already be prepended to your `PATH` after following the instructions above, but sometimes you will need to source the virtual environment or load the Spack module again after restarting your terminal.

Once psyclone is in your *PATH* you can execute `which psyclone` to see the installation directory. Some supporting files such as configuration, examples and instrumentation libraries are installed under the `share/psyclone` directory relative to the psyclone installation. You can replace `bin/psyclone` in the string returned by `which psyclone` with `share/psyclone` to find their location.

1.2 Configuration

Various aspects of PSyclone are controlled through a configuration file, `psyclone.cfg`. The default version of this file is located in the `share/psyclone/psyclone.cfg` file in the *Installation location*.

Warning: If PSyclone is installed in 'editable' mode (`-e` flag to `pip`), or in a non-standard location, then PSyclone will not be able to find the default configuration file. There are two solutions to this:

1. copy a configuration file to the location specified above.

2. set the `PSYCLONE_CONFIG` environment variable (or the `--config` flag) to the full-path to the configuration file, e.g.:

```
export PSYCLONE_CONFIG=/some/path/PSyclone/config/psyclone.cfg
```

See *Configuration* for more details about the settings contained within the config file.

1.3 Running PSyclone

You are now ready to run PSyclone. One way of doing this is to use the `psyclone` command. To list the available options run: `psyclone -h`, it should output:

```
usage: psyclone [-h] [--version] [--config CONFIG] [-s SCRIPT] [-I INCLUDE]
                [-l {off,all,output}] [--profile {invokes,routines,kernels}]
                                [--backend {enable-validation,disable-validation}] [-o␣
↪OUTPUT_FILE]
                                [-api DSL] [-oalg OUTPUT_ALGORITHM_FILE] [-opsy OUTPUT_
↪PSY_FILE]
                [-okern OUTPUT_KERNEL_PATH] [-d DIRECTORY] [-dm] [-nodm]
                [--kernel-renaming {multiple,single}]
                filename

Transform a file using the PSyclone source-to-source Fortran compiler

positional arguments:
  filename              input source code

options:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --version, -v         display version information
  --config CONFIG, -c CONFIG
                        config file with PSyclone specific options
  -s SCRIPT, --script SCRIPT
                        filename of a PSyclone optimisation recipe
  -I INCLUDE, --include INCLUDE
                        path to Fortran INCLUDE or module files
  -l {off,all,output}, --limit {off,all,output}
                        limit the Fortran line length to 132 characters (default 'off').
                        Use 'all' to apply limit to both input and output Fortran. Use
                        'output' to apply line-length limit to output Fortran only.
  --profile {invokes,routines,kernels}, -p {invokes,routines,kernels}
                        add profiling hooks for 'kernels', 'invokes' or 'routines'
  --backend {enable-validation,disable-validation}
                        options to control the PSyIR backend used for code generation.
                        Use 'disable-validation' to disable the validation checks that
                        are performed by default.
  -o OUTPUT_FILE        (code-transformation mode) output file
  -api DSL, --psykal-dsl DSL
                        whether to use a PSyKAl DSL (one of ['lfric', 'gocean'])
  -oalg OUTPUT_ALGORITHM_FILE
                        (psykal mode) filename of transformed algorithm code
  -opsy OUTPUT_PSY_FILE
                        (psykal mode) filename of generated PSy-layer code
  -okern OUTPUT_KERNEL_PATH
                        (psykal mode) directory in which to put transformed kernels,␣
↪default
                        is the current working directory
  -d DIRECTORY, --directory DIRECTORY
                        (psykal mode) path to a root directory structure containing␣
↪kernel
                        source code. Multiple roots can be specified by using multiple -d
                        arguments.
  -dm, --dist_mem       (psykal mode) generate distributed memory code
  -nodm, --no_dist_mem  (psykal mode) do not generate distributed memory code
  --kernel-renaming {multiple,single}
                        (psykal mode) naming scheme to use when re-naming transformed␣
↪kernels
```

There is more detailed information about each flag in *The psyclone command* section, but the main parameters are the input source file that we aim to transform, and a transformation recipe that is provided with the `-s` flag. In addition to these, note that psyclone can be used in two distinct modes: the code-transformation mode (when no `-api`/`--psykal-dsl` flags are provided) or the PSyKAl DSL mode (when a `-api`/`--psykal-dsl` flag is provided). The following sections provide a brief introduction to each mode.

1.3.1 PSyclone for Code Transformation

When using PSyclone for transforming existing Fortran files, only an input source file is required:

```
psyclone input_file.f90
```

However, we usually want to redirect the output to a file so that we can later compile it. We can do this using the *-o* flag:

```
psyclone input_file.f90 -o output.f90
```

This should not transform the semantics of the code (only the syntax), and is what we sometimes refer to as a "passthrough" run. This can be useful as an initial correctness test when applying PSyclone to a new code.

However, PSyclone allows users to programatically change the source code of the processed file. This is achieved using transformation recipes which are python scripts with a *trans* function defined. For example:

```
def trans(psyir):
    ''' Add OpenMP Parallel Loop directives.

    :param psyir: the PSyIR of the provided file.
    :type psyir: :py:class:`psyclone.psyir.nodes.FileContainer`

    '''
    omp_trans = TransInfo().get_trans_name('OMPParallelLoopTrans')

    for loop in psyir.walk(Loop):
        try:
            omp_trans.apply(loop)
        except TransformationError as err:
            print(f"Loop not paralellised because: {err.value}")
```

Warning: Before PSyclone 3.0 the transformation scripts took a PSy object as argument:

```
def trans(psy):
    ''' Add OpenMP Parallel Loop directives.

    :param psy: the PSy object that PSyclone has constructed for the
                'invoke'(s) found in the Algorithm file.
    :type psy: :py:class:`psyclone.dynamo0p3.DynamoPSy`

    '''
    for invoke in psy.invokes.invoke_list:
       invoke.schedule
```

This is deprecated and will stop working in PSyclone releases post version 3.0

And can be applied using the *-s* flag:

```
psyclone input_file.f90 -s trans_script.py -o output.f90
```

To see more complete examples of PSyclone for code transformation, see the `examples/nemo` folder in the PSyclone repository.

1.3.2 PSyclone for PSyKAl DSLs

As indicated above, the `psyclone` command can also be used to process PSyKAl DSLs (`--psykal-dsl` flag). In this case the command takes as input the Fortran source file containing the algorithm specification (in terms of calls to `invoke()`). It parses this, finds the necessary kernel source files and produces two Fortran files. The first contains the *middle, PSy-layer* and the second a re-write of the *algorithm code* to use that layer. These files are named according to the user-supplied arguments (options `-opsy` and `-oalg` respectively). If those arguments are not supplied then the script writes the re-written Fortran Algorithm layer to the terminal. For details of the other command-line arguments please see the *The psyclone command* Section.

Examples are provided in the `examples/lfric` and `examples/gocean` directories of the PSyclone repository. Alternatively, if you have installed PSyclone using `pip` then they may be found in the `share/psyclone` directory under your PSyclone installation (see *which psyclone* for the location of the PSyclone installation). In this case you should copy the whole `examples` directory to some convenient location before attempting to carry out the following instructions.

In this case we are going to use one of the LFRic examples:

```
cd <EGS_HOME>/examples/lfric/eg1
psyclone --psykal-dsl lfric -d ../code -nodm -oalg alg.f90 \
    -opsy psy.f90 ./single_invoke.x90
```

You should see two new files created, called `alg.f90` (containing the re-written algorithm layer) and `psy.f90` (containing the generated PSy- or middle-layer). Since this is an LFRic example the Fortran source code has dependencies on the LFRic system and therefore cannot be compiled stand-alone.

The PSy-layer that PSyclone creates is constructed using the PSyclone Internal Representation (*PSyIR*). Accessing this is demonstrated by the `print_psyir_trans.py` script in the second LFRic example:

```
cd <EGS_HOME>/examples/lfric/eg2
psyclone --psykal-dsl lfric -d ../code -s ./print_psyir_trans.py \
    -opsy psy.f90 -oalg alg.f90 ./multi_invoke_mod.x90
```

Take a look at the `print_psyir_trans.py` script for more information. *Hint*; you can insert a single line in that script in order to break into the Python interpreter during execution: `import pdb; pdb.set_trace()`. This then enables interactive exploration of the PSyIR if you are interested.

CHAPTER

TWO

THE PSYCLONE COMMAND

The `psyclone` command is an executable script designed to be run from the command line, e.g.:

```
> psyclone <args>
```

The optional `-h` argument gives a description of the options provided by the command:

```
> psyclone -h
 usage: psyclone [-h] [--version] [--config CONFIG] [-s SCRIPT] [-I INCLUDE]
                 [-l {off,all,output}] [--profile {invokes,routines,kernels}]
                                 [--backend {enable-validation,disable-validation}] [-o␣
↪OUTPUT_FILE]
                                 [-api DSL] [-oalg OUTPUT_ALGORITHM_FILE] [-opsy OUTPUT_
↪PSY_FILE]
                 [-okern OUTPUT_KERNEL_PATH] [-d DIRECTORY] [-dm] [-nodm]
                 [--kernel-renaming {multiple,single}]
                 filename

 Transform a file using the PSyclone source-to-source Fortran compiler

 positional arguments:
   filename              input source code

 options:
   -h, --help            show this help message and exit
   --version, -v         display version information
   --config CONFIG, -c CONFIG
                         config file with PSyclone specific options
   -s SCRIPT, --script SCRIPT
                         filename of a PSyclone optimisation recipe
   -I INCLUDE, --include INCLUDE
                         path to Fortran INCLUDE or module files
   -l {off,all,output}, --limit {off,all,output}
                         limit the Fortran line length to 132 characters (default 'off').
                         Use 'all' to apply limit to both input and output Fortran. Use
                         'output' to apply line-length limit to output Fortran only.
   --profile {invokes,routines,kernels}, -p {invokes,routines,kernels}
                         add profiling hooks for 'kernels', 'invokes' or 'routines'
   --backend {enable-validation,disable-validation}
                         options to control the PSyIR backend used for code generation.
                         Use 'disable-validation' to disable the validation checks that
                         are performed by default.
```

(continues on next page)

(continued from previous page)

```
  -o OUTPUT_FILE        (code-transformation mode) output file
  -api DSL, --psykal-dsl DSL
                        whether to use a PSyKAl DSL (one of ['lfric', 'gocean'])
  -oalg OUTPUT_ALGORITHM_FILE
                        (psykal mode) filename of transformed algorithm code
  -opsy OUTPUT_PSY_FILE
                        (psykal mode) filename of generated PSy-layer code
  -okern OUTPUT_KERNEL_PATH
                        (psykal mode) directory in which to put transformed kernels,␣
↪default
                        is the current working directory.
  -d DIRECTORY, --directory DIRECTORY
                        (psykal mode) path to a root directory structure containing␣
↪kernel
                        source code. Multiple roots can be specified by using multiple -
↪d
                        arguments.
  -dm, --dist_mem       (psykal mode) generate distributed memory code
  -nodm, --no_dist_mem  (psykal mode) do not generate distributed memory code
  --kernel-renaming {multiple,single}
                        (psykal mode) naming scheme to use when re-naming transformed␣
↪kernels
```

2.1 Basic Use

The simplest way to use `psyclone` is to provide a Fortran input source file:

```
psyclone input.f90
```

If the input file is valid Fortran, PSyclone will print the output Fortran (in this case the same unmodified code but with normalised syntax) to stdout. Otherwise it will print the errors detected while parsing the Fortran file.

Usually we want to redirect the output to a file that we can later compile. We can do this with the *-o* flag:

```
psyclone input.f90 -o output.f90
```

2.2 Transformation script

By default, the `psyclone` command will not apply any transformation (other than canonicalising the code and generating a normalised syntax). To apply transformations to the code, a recipe needs to be specified with the *-s* flag. This option is discussed in more detail in the *Script* section. With a transformation recipe the command looks like:

```
psyclone input.f90 -s transformation_recipe.py
```

2.3 Fortran INCLUDE Files and Modules

If the source code to be processed by PSyclone contains INCLUDE statements then the location of any INCLUDE'd files *must* be supplied to PSyclone via the `-I` or `--include` option. (This is necessary because INCLUDE lines are a part of the Fortran language and must therefore be parsed - they are not handled by any pre-processing step.) Multiple locations may be specified by using multiple `-I` flags, e.g.:

```
psyclone -I /some/path -I /some/other/path input.f90
```

If no include paths are specified then the directory containing the source file currently being parsed is searched by default. If the specified INCLUDE file is not found then PSyclone will abort with an appropriate error. For example:

```
psyclone -I nonexisting test.f90
PSyclone configuration error: Include path 'nonexisting' does not exist
```

Currently, the PSyKAl-based APIs (LFRic and GOcean - see below) will ignore (but preserve) INCLUDE statements in algorithm-layer code. However, INCLUDE statements in kernels will, in general, cause the kernel parsing to fail unless the file(s) referenced in such statements are in the same directory as the kernel file. Once kernel parsing has been re-implemented to use fparser2 (issue #239) and the PSyclone Intermediate Representation then the behaviour will be the same as for generic code-transformations.

Since PSyclone does not attempt to be a full compiler, it does not require that the code be available for any Fortran modules referred to by `use` statements. However, certain transformations *do* require that e.g. type information be determined for all variables in the code being transformed. In this case PSyclone *will* need to be able to find and process any referenced modules. To do this it searches in the directories specified by the `-I`/`--include` flags.

2.3.1 C Pre-processor #include Files

PSyclone currently only supports Fortran input. As such, if a file to be processed contains CPP `#include` statements then it must first be processed by a suitable pre-processor before being passed to PSyclone. PSyclone will abort with an appropriate error if it encounters a `#include` in any code being processed (whether or not a PSykAL DSL is in use).

2.4 Fortran line length

By default the `psyclone` command will generate Fortran code with no consideration of Fortran line-length limits. As the line-length limit for free-format Fortran is 132 characters, the code that is output may be non-conformant.

Line length is not an issue for many compilers as they provide flags to increase or disable Fortran standard line lengths limits. However this is not the case for all compilers.

When either the `-l all` or `-l output` option is specified to the `psyclone` command, the output will be line wrapped so that the output lines are always within the 132 character limit.

The `-l all` additionally checks the input Fortran files for conformance and raises an error if they do not conform.

Line wrapping is not performed by default. There are two reasons for this. This first reason is that most compilers are able to cope with long lines. The second reason is that the line wrapping implementation could fail in certain pathological cases. The implementation and limitations of line wrapping are discussed in the *Limitations* section.

2.5 Backend Options

The final code generated by PSyclone is created by passing the PSyIR tree to one of the 'backends' (see PSyIR Back-ends in the Developer Guide for more details). The `--backend` flag permits a user to tune the behaviour of this code generation. Currently, the only option is `{en,dis}able-validation` which turns on/off the validation checks performed when doing code generation. By default, such validation is enabled as it is only at code-generation time that certain constraints can be checked (since PSyclone does not mandate the order in which code transformations are applied). Occasionally, these validation checks may raise false positives (due to incomplete implementations), at which point it is useful to be able to disable them. The default behaviour may be changed by adding the `BACKEND_CHECKS_ENABLED` entry to the *configuration file*. Any command-line setting always takes precendence though. It is recommended that validation only be disabled as a last resort and for as few input source files as possible.

2.6 Automatic Profiling Instrumentation

The `--profile` option allows the user to instruct PSyclone to automatically insert profiling calls in addition to the code transformations specified in the recipe. This flag accepts the options: `routines`, `invokes` and `kernels`. PSyclone will insert profiling-start and -stop calls at the beginning and end of each routine, PSy-layer invoke or PSy-layer kernel call, respectively. The generated code must be linked against the PSyclone profiling interface and the profiling tool itself. The application that calls the PSyclone-generated code is responsible for initialising and finalising the profiling library that is being used (if necessary). For more details on the use of this profiling functionality please see the *Profiling* section.

2.7 Using PSyclone for PSyKAL DSLs

In addition to the default code-transformation mode, `psyclone` can also be used to process Fortran files that implement PSyKAL DSLs (see *Introduction to PSyKAl*). To do this you can choose a DSL API with the `-api` or `--psykal-dsl` flag.

The main difference is that, instead of providing a single file to process, for PSyKAl DSLs PSyclone expects an algorithm-layer file that describes the high-level view of an algorithm. PSyclone will use this algorithm file and the metadata of the kernels that it calls to generate a PSy(Parallel System)-layer code that connects the Algorithm layer to the Kernels. In this mode of operation, any supplied transformation recipe is applied to the PSy-layer.

By default, the `psyclone` command for PSyKAl APIs will generate distributed memory (DM) code (unless otherwise specified in the *Configuration* file). Alternatively, whether or not to generate DM code can be specified as an argument to the `psyclone` command using the `-dm`/`--dist_mem` or `-nodm`/`--no_dist_mem` flags, respectively. For exampe the following command will generate GOcean PSyKAl code with DM:

```
psyclone -api gocean -dm algorithm.f90
```

See *psyclone usage for PSyKAl* section for more information about how to use PSyKAl DSLs.

2.7.1 PSyKAl file output

By default the modified algorithm code and the generated PSy code are output to the terminal. These can instead be output to files by using the `-oalg <file>` and `-opsy <file>` options, respectively. For example, the following will output the generated PSy code to the file 'psy.f90' but the algorithm code will be output to the terminal:

```
psyclone -opsy psy.f90 algorithm.f90
```

If PSyclone is being used to transform Kernels then the location to write these to is specified using the `-okern <directory>` option. If this is not supplied then they are written to the current working directory. By default, PSyclone will overwrite any kernel of the same name in that directory. To change this behaviour, the user can use the `--no_kernel_clobber` option. This causes PSyclone to re-name any transformed kernel that would clash with any of those already present in the output directory.

2.7.2 Algorithm files with no invokes

If `psyclone` is provided with a file that contains no `invoke` calls then the command outputs a warning to `stdout` and copies the input file to `stdout`, or to the specified algorithm file (if the `-oalg <file>` option is used). No PSy code will be output. If a file is specified using the `-opsy <file>` option this file will not be created.

```
> psyclone -opsy psy.f90 -oalg alg_new.f90 empty_alg.f90
Warning: 'Algorithm Error: Algorithm file contains no invoke() calls: refusing to
generate empty PSy code'
```

2.7.3 Kernel search directory

When an algorithm file is parsed, the parser looks for the associated kernel files. The way in which this is done requires that any user-defined kernel routine (as opposed to *Built-ins*) called within an invoke must have an explicit use statement. For example, the following code gives an error:

```
> cat no_use.f90
program no_use
  call invoke(testkern_type(a,b,c,d,e))
end program no_use
> psyclone -api gocean no_use.f90
"Parse Error: kernel call 'testkern_type' must either be named in a use statement or be␣
↪a recognised built-in (one of '[]' for this API)"
```

(If the chosen API has any *Built-ins* defined then these will be listed within the `[]` in the above error message.) If the name of the kernel is provided in a use statement then the parser will look for a file with the same name as the module in the use statement. In the example below, the parser will look for a file called "testkern.f90" or "testkern.F90":

```
> cat use.f90
program use
  use testkern, only : testkern_type
  call invoke(testkern_type(a,b,c,d,e))
end program use
```

Therefore, for PSyclone to find kernel files, the module name of a kernel file must be the same as its filename. By default the parser looks for the kernel file in the same directory as the algorithm file. If this file is not found then an error is reported.

```
> psyclone use.f90
Kernel file 'testkern.[fF]90' not found in <location>
```

The `-d` option can be used to tell `psyclone` where to look for kernel files by supplying it with a directory. The execution will recurse from the specified directory path to look for the required file. There must be only one instance of the specified file within (or below) the specified directory:

```
> cd <PSYCLONEHOME>/src/psyclone
> psyclone -d . use.f90
More than one match for kernel file 'testkern.[fF]90' found!
> psyclone -d tests/test_files/dynamo0p3 -api lfric use.f90
[code output]
```

Note: The `-d` option can be repeated to add as many search directories as is required, with the constraint that there must be only one instance of the specified file within (or below) the specified directories.

2.7.4 Transforming PSyKAl Kernels

When transforming kernels there are two use-cases to consider:

1. a given kernel will be transformed only once and that version then used from multiple, different Invokes and Algorithms;
2. a given kernel is used from multiple, different Invokes and Algorithms and is transformed differently, depending on the Invoke.

Whenever PSyclone is used to transform a kernel, the new kernel must be re-named in order to avoid clashing with other possible calls to the original. By default (`--kernel-renaming multiple`), PSyclone generates a new, unique name for each kernel that is transformed. Since PSyclone is run on one Algorithm file at a time, it uses the chosen kernel output directory (`-okern`) to ensure that names created by different invocations do not clash. Therefore, when building a single application, the same kernel output directory must be used for each separate invocation of PSyclone.

Alternatively, in order to support use case 1, a user may specify `--kernel-renaming single`: now, before transforming a kernel, PSyclone will check the kernel output directory and if a transformed version of that kernel is already present then that will be used. Note, if the kernel file on disk does not match with what would be generated then PSyclone will raise an exception.

CHAPTER

THREE

CONFIGURATION

PSyclone reads various run-time configuration options from the `psyclone.cfg` file. As described in *Configuration*, the default `psyclone.cfg` configuration file is installed in `<python-base-prefix>/share/psyclone/` during the installation process. The original version of this file is in the `PSyclone/config` directory of the PSyclone distribution.

At execution-time, the user can specify a custom configuration file to be used. This can either be done with the `--config` command-line option, or by specifying the (full path to the) configuration file to use via the `PSYCLONE_CONFIG` environment variable. If the specified configuration file is not found then PSyclone will fall back to searching in a list of default locations.

The ordering of these locations depends upon whether PSyclone is being run within a Python virtual environment (such as `venv`). If no virtual environment is detected then the locations searched, in order, are:

1. `${PWD}/.psyclone/`
2. `${HOME}/.local/share/psyclone/`
3. `<python-base-dir>/share/psyclone/`

where `<python-base-dir>` is the path stored in Python's `sys.prefix`.

If a virtual environment is detected then it is assumed that the `share` directory will be a part of that environment. In order to maintain isolation of distinct virtual environments this directory is then checked *before* the user's home directory. i.e. the list of locations searched is now:

1. `${PWD}/.psyclone/`
2. `<python-base-dir>/share/psyclone/`
3. `${HOME}/.local/share/psyclone/`

Note that for developers a slightly different configuration handling is implemented, see Module: configuration for details.

3.1 Options

The configuration file is read by the Python ConfigParser class (https://docs.python.org/3/library/configparser.html) and must be formatted accordingly. It currently consists of a `DEFAULT` section e.g.:

```
[DEFAULT]
DISTRIBUTED_MEMORY = true
REPRODUCIBLE_REDUCTIONS = false
REPROD_PAD_SIZE = 8
PSYIR_ROOT_NAME = psyir_tmp
VALID_PSY_DATA_PREFIXES = profile, extract
```

and an optional API specific section, for example for the `lfric` section:

```
[lfric]
access_mapping = gh_read: read, gh_write: write, gh_readwrite: readwrite,
                 gh_inc: inc, gh_readinc: readinc, gh_sum: sum
COMPUTE_ANNEXED_DOFS = false
supported_fortran_datatypes = real, integer, logical
default_kind = real: r_def, integer: i_def, logical: l_def
precision_map = i_def: 4,
                l_def: 1,
                r_def: 8,
                r_double: 8,
                r_ncdf: 8,
                r_quad: 16,
                r_second: 8,
                r_single: 4,
                r_solver: 4,
                r_tran: 8,
                r_bl: 8,
                r_phys: 8,
                r_um: 8
RUN_TIME_CHECKS = false
NUM_ANY_SPACE = 10
NUM_ANY_DISCONTINUOUS_SPACE = 10
```

or for `gocean`:

```
[gocean]
access_mapping = go_read:read, go_write:write, go_readwrite:readwrite
grid-properties = go_grid_xstop: {0}%%grid%%subdomain%%internal%%xstop: scalar,
                  go_grid_ystop: {0}%%grid%%subdomain%%internal%%ystop: scalar,
                  go_grid_data: {0}%%data: array,
                  ...
```

The meaning of the various entries is described in the following sub-sections.

Note that ConfigParser supports various forms of boolean entry including "true/false", "yes/no" and "1/0". See https://docs.python.org/3/library/configparser.html#supported-datatypes for more details.

3.1.1 `DEFAULT` Section

This section contains entries that are, in principle, applicable to all APIs supported by PSyclone.

Entry	Description	Type
DISTRIBUTED_MEMORY	Whether or not to generate code for distributed-memory parallelism by default. Note that this is currently only supported for the LFRic (Dynamo 0.3) API.	bool
REPRODUCIBLE_REDUCTIONS	Whether or not to generate code for reproducible OpenMP reductions (see *Reductions*) by default.	bool
REPROD_PAD_SIZE	If generating code for reproducible OpenMP reductions, this setting controls the amount of padding used between elements of the array in which each thread accumulates its local reduction. (This prevents false sharing of cache lines by different threads.)	int
PSYIR_ROOT_NAME	The root for generated PSyIR symbol names if one is not supplied when creating a symbol. Defaults to "psyir_tmp".	str
VALID_PSY_DATA_PREFIXES	Which class prefixes are permitted in any PSyData-related transformations. See *PSyData API* for details.	list of str
BACKEND_CHECKS_ENABLED	Optional (defaults to True). Whether or not the PSyIR backend should validate the tree that it is passed. Can be overridden by the `--backend` command-line flag (see *Backend Options*).	bool

3.1.2 Common Sections

The following entries must be defined for each API in order for PSyclone to work as expected:

Entry	Description
access_mapping	This field defines the strings that are used by a particular API to indicate write, read, … access. Its value is a comma separated list of access-string:access pairs, e.g.: `gh_read: read, gh_write: write, gh_readwrite: readwrite, gh_inc: inc, gh_readinc: gh_sum: sum` At this stage these 6 types are defined for read, write, read+write, increment, read+increment and summation access by PSyclone. Sum is a form of reduction. The GOcean API does not support increment or sum, so it only defines three mappings for read, write, and readwrite.

3.1.3 `lfric` Section

This section contains configuration options that are only applicable when using the LFRic (Dynamo 0.3) API.

Entry	Description
COMPUTE_ANNEXED_DOFS	Whether or not to perform redundant computation over annexed dofs in order to reduce the number of halo exchanges, see *Annexed DoFs*.
supported_fortran_datatypes	Captures the supported Fortran data types of LFRic arguments, see *Supported Data Types and Default Kind*.
default_kind	Captures the default kinds (precisions) for the supported Fortran data types in LFRic, see *Supported Data Types and Default Kind*.
precision_map	Captures the value of the actual precisions in bytes, see *Precision Map*
RUN_TIME_CHECKS	Specifies whether to generate run-time validation checks, see *Run-time Checks*.
NUM_ANY_SPACE	Sets the number of `ANY_SPACE` function spaces in LFRic, see *Number of Generalised ANY_*_SPACE Function Spaces*.
NUM_ANY_DISCONTINUOUS_SPACE	Sets the number of `ANY_DISCONTINUOUS_SPACE` function spaces in LFRic, see *Number of Generalised ANY_*_SPACE Function Spaces*.

3.1.4 gocean Section

This section contains configuration options that are only applicable when using the Gocean 1.0 API.

Entry	Description
iteration-spaces	This contains definitions of additional iteration spaces used by PSyclone. A detailed description can be found in the *Iteration-spaces* section of the GOcean1.0 chapter.
grid-properties	This key contains definitions to access various grid properties. A detailed description can be found in the *Grid Properties* section of the GOcean1.0 chapter.

CHAPTER

FOUR

TUTORIAL AND EXAMPLES

4.1 Tutorial

PSyclone provides a hands-on tutorial. The easiest way to follow it is reading the Readme files in github. The tutorial is divided into two sections, a first section that introduces PSyclone and how to use it to transform generic Fortran code (this is the recommended starting point for everybody). And a second section about the LFRic DSL (this is only recommended for people interested in PSyKAL DSLs and LFRic in particular).

To do the proposed hands-on you will need a linux shell with Python installed and to download the hands-on directory with:

```
git clone --recursive git@github.com:stfc/PSyclone.git
cd PSyclone
# If psyclone isn't already installed you can use 'pip' in this folder to
# install a version that matches the downloaded tutorials
pip install .
cd tutorial/practicals
```

4.2 Examples

Various examples of the use of PSyclone are provided under the `examples` directory in the Git repository. If you have installed PSyclone using `pip` then the examples may be found in `share/psyclone/examples` in psyclone *Installation location*.

Running any of these examples requires that PSyclone be installed on the host system, see Section *Getting Going*. This section is intended to provide an overview of the various examples so that a user can find one that is appropriate to them. For details of what each example does and how to run each example please see the `README.md` files in the associated directories.

For the purposes of correctness checking, the whole suite of examples may be executed using Gnu `make` (this functionality is used by GitHub Actions alongside the test suite). The default target is `transform` which just performs the PSyclone code transformation steps for each example. For those examples that support it, the `compile` target also requests that the generated code be compiled. The `notebook` target checks the various Jupyter notebooks using `nbconvert`.

Note: As outlined in the *Run* section, if working with the examples from a PSyclone installation, it is advisable to copy the whole `examples` directory to some convenient location before running them. If you have copied the `examples` directory but still wish to use `make` then you will also have to set the `PSYCLONE_CONFIG` environment variable to the full path to the PSyclone configuration file, e.g. `PSYCLONE_CONFIG=/some/path/psyclone.cfg make`.

4.2.1 Compilation

Some of the examples support compilation (and some even execution of a compiled binary). Please consult the `README.md` to check which ones can be compiled and executed.

As mentioned above, by default each example will execute the `transform` target, which performs the PSyclone code transformation steps. In order to compile the sources, use the target `compile`:

```
make compile
```

which will first perform the transformation steps before compiling any created Fortan source files. If the example also supports running a compiled and linked binary, use the target:

```
make run
```

This will first trigger compilation using the `compile` target, and then execute the program with any parameters that might be required (check the corresponding `README.md` document for details).

All `Makefile`s support the variables `F90` and `F90FLAGS` to specify the compiler and compilation flags to use. By default, the Gnu Fortran compiler (`gfortran`) is used, and the compilation flags will be set to debugging. If you want to change the compiler or flags, just define these as environment variables:

```
F90=ifort F90FLAGS="-g -check bounds" make compile
```

To clean all compiled files (and potential output files from a run), use:

```
make clean
```

This will clean up in the `examples` directory. If you want to change compilers or compiler flags, you should run `make allclean`, see the section about *Dependencies* for details.

4.2.2 Supported Compilers

All examples have been tested with the following compilers. Please let the developers know if you have problems using a compiler that has been tested or if you are working with a different compiler so it can be recorded in this table.

Compiler	Version
Gnu Fortran	9.3
Intel Fortran	17, 21
NVIDIA Fortran	23.5

4.2.3 Dependencies

Any required library that is included in PSyclone (typically the infrastructure libraries for the APIs, or *PSyData wrapper libraries*) will automatically be compiled with the same compiler and compilation flags as the examples.

Note: Once a dependent library is compiled, changing the compilation flags will not trigger a recompilation of this library. For example, if an example is first compiled with debug options, and later the same or a different example is compiled with optimisations, the dependent library will not automatically be recompiled!

All `Makefile`s support an `allclean` target, which will not only clean the current directory, but also all libraries the current example depends on.

Important: Using `make allclean` is especially important if the compiler is changed. Typically, one compiler cannot read module information from a different compiler, and then compilation will fail.

NetCDF

Some examples require NetCDF for compilation. Installation of NetCDF is described in detail in the hands-on practicals documentation.

4.3 PSyIR Examples

Examples may all be found in the `examples/psyir` directory. Read the `README.md` file in this directory for full details.

4.3.1 Example 1: Constructing PSyIR and Generating Code

`create.py` is a Python script that demonstrates the use of the various `create` methods to build a PSyIR tree from scratch.

4.3.2 Example 2: Creating PSyIR for Structure Types

`create_structure_types.py` demonstrates the representation of structure types (i.e. Fortran derived types or C structs) in the PSyIR.

4.4 GOcean Examples

4.4.1 Example 1: Loop transformations

Examples of applying various transformations (loop fusion, OpenMP, OpenMP Taskloop, OpenACC, OpenCL) to the semi-PSyKAl'd version of the Shallow benchmark. ("semi" because not all kernels are called from within invoke()'s.) Also includes an example of generating a DAG from an InvokeSchedule.

4.4.2 Example 2: OpenACC

This is a simple but complete example of using PSyclone to enable an application to run on a GPU by adding OpenACC directives. A `Makefile` is included which will use PSyclone to generate the PSy code and transformed kernels and then compile the application. This compilation requires that the dl_esm_inf library be installed/available - it is provided as a Git submodule of the PSyclone project (see Installation in the Developers' Guide for details on working with submodules).

The supplied `Makefile` also provides a second, `profile` target which performs the same OpenACC transformations but then encloses the whole of the resulting PSy layer in a profiling region. By linking this with the PSyclone NVTX profiling wrapper (and the NVTX library itself), the resulting application can be profiled using NVIDIA's *nvprof* or *nvvp* tools.

4.4.3 Example 3: OpenCL

Example of the use of PSyclone to generate an OpenCL driver version of the PSy layer and OpenCL kernels. The `Makefile` in this example provides a target (*make compile-ocl*) to compile the generated OpenCL code. This requires an OpenCL implementation installed in the system. Read the README provided in the example folder for more details about how to compile and execute the generated OpenCL code.

4.4.4 Example 4: Kernels containing use statements

Transforming kernels for use with either OpenACC or OpenCL requires that we handle those that access data and/or routines via module `use` statements. This example shows the various forms for which support is being implemented. Although there is support for converting global-data accesses into kernel arguments, PSyclone does not yet support nested `use` of modules (i.e. data accessed via a module that in turn imports that symbol from another module) and kernels that call other kernels (Issue #342).

4.4.5 Example 5: PSyData

This directory contains all examples that use the *PSyData API*. At this stage there are three runnable examples:

Example 5.1: Kernel data extraction

This example shows the use of kernel data extraction in PSyclone. It instruments each of the two invokes in the example program with the PSyData-based kernel extraction code. Detailed compilation instructions are in the `README.md` file, including how to switch from using the stand-alone extraction library to the NetCDF-based one (see *Extraction Libraries* for details).

The `Makefile` in this example will create the binary that extracts the data at run time, as well as two driver programs that can read in the extracted data, call the kernel, and compare the results. These driver programs are independent of the dl_esm_inf infrastructure library. These drivers can only read the corresponding file format, i.e. a NetCDF driver program cannot read in extraction data that is based on Fortran IO and vice versa.

Note: At this stage the driver program still needs the infrastructure library when compiling the kernels, see #1757.

Example 5.2: Profiling

This example shows how to use the profiling support in PSyclone. It instruments two invoke statements and can link in with any of the following profiling wrapper libraries: template, simple_timer, dl_timer, TAU, and DrHook (see *Interface to Third Party Profiling Tools*). The `README.md` file contains detailed instructions on how to build the different executables. By default (i.e. just using `make` without additional parameters) it links in with the template profiling library included in PSyclone. This library just prints out the name of the module and region before and after each invoke is executed. This example can actually be executed to test the behaviour of the various profiling wrappers, and is also useful if you want to develop your own wrapper libraries.

Example 5.3: Read-only-verification

This example shows the use of read-only-verification with PSyclone. It instruments each of the two invokes in the example program with the PSyData-based read-only-verification code. It uses the dl_esm_inf-specific read-only-verification library (`lib/read_only/dl_esm_inf/`).

Note: The `update_field_mod` subroutine contains some very buggy and non-standard code to change the value of some read-only variables and fields, even though the variables are all declared with `intent(in)`. It uses the addresses of variables and then out-of-bound writes to a writeable array to actually overwrite the read-only variables. Using array bounds checking at runtime will be triggered by these out-of-bound writes.

The `Makefile` in this example will link with the compiled read-only-verification library. You can execute the created binary and it will print two warnings about modified read-only variables:

```
---------------------------------------
Double precision field b_fld has been modified in main : update
Original checksum:   4611686018427387904
New checksum:        4638355772470722560
---------------------------------------
---------------------------------------
Double precision variable z has been modified in main : update
Original value:    1.0000000000000000
New value:         123.00000000000000
---------------------------------------
```

Example 5.4: Value Range Check

This example shows the use of valid number verification with PSyclone. It instruments each of the two invokes in the example program with the PSyData-based Value-Range-Check code. It uses the dl_esm_inf-specific value range check library (`lib/value_range_check/dl_esm_inf/`).

Note: The `update_field_mod` subroutine contains code that will trigger a division by 0 to create NaNs. If the compiler happens to add code that handles floating point exceptions , this will take effect before the value testing is done by the PSyData-based verification code.

The `Makefile` in this example will link with the compiled value_range_check library. You can then execute the binary and enable the value range check by setting environments (see *value range check* for details).

```
PSYVERIFY__main__init__b_fld=2:3 ./value_range_check
...
PSyData: Variable b_fld has the value 0.0000000000000000 at index/indices 6 1 in module
↪'main', region 'init', which is not between '2.0000000000000000' and '3.
↪0000000000000000'.
...
PSyData: Variable a_fld has the invalid value 'Inf' at index/indices 1 1 in module 'main
↪', region 'update'.
```

As indicated in *value range check*, you can also check a variable in all kernels of a module, or in any instrumented code region (since the example has only one module, both settings below will create the same warnings):

```
PSYVERIFY__main__b_fld=2:3 ./value_range_check
PSYVERIFY__b_fld=2:3 ./value_range_check
...
PSyData: Variable b_fld has the value 0.0000000000000000 at index/indices 6 1 in module
↪'main', region 'init', which is not between '2.0000000000000000' and '3.
↪0000000000000000'.
...
PSyData: Variable b_fld has the value 0.0000000000000000 at index/indices 6 1 in module
↪'main', region 'update', which is not between '2.0000000000000000' and '3.
↪0000000000000000'.
```

Notice that now a warning is created for both kernels: `init` and `update`.

Support for checking arbitrary Fortran code is tracked as issue #2741.

4.4.6 Example 6: PSy-layer Code Creation using PSyIR

This example informs the development of the code generation of PSy-layer code using the PSyIR language backends.

4.5 LFRic Examples

These examples illustrate the functionality of PSyclone for the LFRic domain.

4.5.1 Example 1: Basic Operation

Basic operation of PSyclone with an `invoke()` containing two kernels, one *user-supplied*, the other a *Built-in*. Code is generated both with and without distributed-memory support. Also demonstrates the use of the `-d` flag to specify where to search for user-supplied kernel code (see *The psyclone command* section for more details).

4.5.2 Example 2: Applying Transformations

A more complex example showing the use of PSyclone *transformations* to change the generated PSy-layer code. Provides examples of kernel-inlining and loop-fusion transformations.

4.5.3 Example 3: Distributed and Shared Memory

Shows the use of colouring and OpenMP for the Dynamo 0.3 API. Includes multi-kernel, named invokes with both user-supplied and built-in kernels. Also shows the use of `Wchi` function space metadata for coordinate fields in LFRic.

4.5.4 Example 4: Multiple Built-ins, Named Invokes and Boundary Conditions

Demonstrates the use of the special `enforce_bc_kernel` which PSyclone recognises as a boundary-condition kernel.

4.5.5 Example 5: Stencils

Example of kernels which require stencil information.

4.5.6 Example 6: Reductions

Example of applying OpenMP to an InvokeSchedule containing kernels that perform reduction operations. Two scripts are provided, one of which demonstrates how to request that PSyclone generate code for a reproducible OpenMP reduction. (The default OpenMP reduction is not guaranteed to be reproducible from one run to the next on the same number of threads.)

4.5.7 Example 7: Column-Matrix Assembly Operators

Example of kernels requiring Column-Matrix Assembly operators.

4.5.8 Example 8: Redundant Computation

Example of the use of the redundant-computation and move transformations to eliminate and re-order halo exchanges.

4.5.9 Example 9: Writing to Discontinuous Fields

Demonstrates the behaviour of PSyclone for kernels that read and write quantities on horizontally-discontinuous function spaces. In addition, this example demonstrates how to write a PSyclone transformation script that only colours loops over continuous spaces.

4.5.10 Example 10: Inter-grid Kernels

Demonstrates the use of "inter-grid" kernels that prolong or restrict fields (map between grids of different resolutions), as well as the use of `ANY_DISCONTINUOUS_SPACE` function space metadata.

4.5.11 Example 11: Asynchronous Halo Exchanges

Example of the use of transformations to introduce redundant computation, split synchronous halo exchanges into asynchronous exchanges (start and stop) and move the starts of those exchanges in order to overlap them with computation.

4.5.12 Example 12: Code Extraction

Example of applying code extraction to Nodes in an Invoke Schedule:

```
> psyclone -nodm -s ./extract_nodes.py \
    gw_mixed_schur_preconditioner_alg_mod.x90
```

or to a Kernel in an Invoke after applying transformations:

```
> psyclone -nodm -s ./extract_kernel_with_transformations.py \
    gw_mixed_schur_preconditioner_alg_mod.x90
```

For now it only inserts comments in appropriate locations while the the full support for code extraction is being developed.

This example also contains a Python helper script `find_kernel.py` which displays the names and Schedules of Invokes containing call(s) to the specified Kernel:

```
> python find_kernel.py
```

4.5.13 Example 13 : Kernel Transformation

Demonstrates how an LFRic kernel can be transformed. The example transformation makes Kernel values constant where appropriate. For example, the number of levels is usually passed into a kernel by argument but the transformation allows a particular value to be specified which the transformation then sets as a parameter in the kernel. Hard-coding values in a kernel helps the compiler to do a better job when optimising the code.

4.5.14 Example 14: OpenACC

Example of adding OpenACC directives in the LFRic API. A single transformation script (`acc_parallel.py`) is provided which demonstrates how to add OpenACC Kernels and Enter Data directives to the PSy-layer. It supports distributed memory being switched on by placing an OpenACC Kernels directive around each (parallelisable) loop, rather than having one for the whole invoke. This approach avoids having halo exchanges within an OpenACC Parallel region. The script also uses *ACCRoutineTrans* to transform the one user-supplied kernel through the addition of an `!$acc routine` directive. This ensures that the compiler builds a version suitable for execution on the accelerator (GPU).

This script is used by the supplied Makefile. The invocation of PSyclone within that Makefile also specifies the `--profile invokes` option so that each `invoke` is enclosed within profiling calipers (by default the 'template' profiling library supplied with PSyclone is used at the link stage). Compilation of the example using the NVIDIA compiler may be performed by e.g.:

```
> F90=nvfortran F90FLAGS="-acc -Minfo=all" make compile
```

Launching the resulting binary with `NV_ACC_NOTIFY` set will show details of the kernel launches and data transfers:

```
> NV_ACC_NOTIFY=3 ./example_openacc
...
  Step              5 : chksm =    2.1098315506694516E-004
  PreStart called for module 'main_psy' region 'invoke_2-setval_c-r2'
 upload CUDA data  file=PSyclone/examples/lfric/eg14/main_psy.f90 function=invoke_2␣
↪line=183 device=0 threadid=1 variable=.attach. bytes=144
 upload CUDA data  file=PSyclone/examples/lfric/eg14/main_psy.f90 function=invoke_2␣
↪line=183 device=0 threadid=1 variable=.attach. bytes=144
```

(continues on next page)

(continued from previous page)

```
 launch CUDA kernel  file=PSyclone/examples/lfric/eg14/main_psy.f90 function=invoke_2␣
↪line=186 device=0 threadid=1 num_gangs=5 num_workers=1 vector_length=128 grid=5␣
↪block=128
  PostEnd called for module 'main_psy' region 'invoke_2-setval_c-r2'
 download CUDA data  file=PSyclone/src/psyclone/tests/test_files/dynamo0p3/
↪infrastructure//field/field_r64_mod.f90 function=log_minmax line=756 device=0␣
↪threadid=1 variable=self%data(:) bytes=4312
 20230807214504.374+0100:INFO : Min/max minmax of field1 =   0.30084014E+00  0.
↪17067212E+01
...
```

However, performance will be very poor as, with the limited optimisations and directives currently applied, the NVIDIA compiler refuses to run the user-supplied kernel in parallel.

4.5.15 Example 15: CPU Optimisation of Matvec

Example of optimising the LFRic matvec kernel for CPUs. This is work in progress with the idea being that PSyclone transformations will be able to reproduce hand-optimised code.

There is one script which, when run:

```
> psyclone ./matvec_opt.py ../code/gw_mixed_schur_preconditioner_alg_mod.x90
```

will print out the modified matvec kernel code. At the moment no transformations are included (as they are work-in-progress) so the code that is output is the same as the original (but looks different as it has been translated to PSyIR and then output by the PSyIR Fortran back-end).

4.5.16 Example 16: Generating LFRic Code Using LFRic-specific PSyIR

This example shows how LFRic-specific PSyIR can be used to create LFRic kernel code. There is one Python script provided which when run:

```
> python create.py
```

will print out generated LFRic kernel code. The script makes use of LFRic-specific data symbols to simplify code generation.

4.5.17 Example 17: Runnable Simplified Examples

This directory contains three simplified LFRic examples that can be compiled and executed - of course, a suitable Fortran compiler is required. The examples are using a subset of the LFRic infrastructure library, which is contained in PSyclone and which has been slightly modified to make it easier to create stand-alone, non-MPI LFRic codes.

Example 17.1: A Simple Runnable Example

The subdirectory `full_example` contains a very simple example code that uses PSyclone to process two invokes. It uses unit-testing code from various classes to create the required data structures like initial grid etc. The code can be compiled with `make compile`, and the binary executed with either `make run` or `./example`.

Example 17.2: A Simple Runnable Example With NetCDF

The subdirectory `full_example_netcdf` contains code very similar to the previous example, but uses NetCDF to read the initial grid from the NetCDF file `mesh_BiP128x16-400x100.nc`. Installation of NetCDF is described in the hands-on practicals documentation. The code can be compiled with `make compile`, and the binary executed with either `make run` or `./example`.

Example 17.3: Kernel Data Extraction

The example in the subdirectory `full_example_extract` shows the use of *kernel extraction*. The code can be compiled with `make compile`, and the binary executed with either `make run` or `./extract.standalone`. By default, it will be using a stand-alone extraction library (see *Extraction Libraries*). If you want to use the NetCDF version, set the environment variable `TYPE` to be `netcdf`:

```
TYPE=netcdf make compile
```

This requires the installation of a NetCDF development environment (see here for installing NetCDF). The binary will be called `extract.netcdf`, and the output files will have the `.nc` extension.

Running the compiled binary will create two Fortran binary files or two NetCDF files if the NetCDF library was used. They contain the input and output parameters for the two invokes in this example:

```
cd full_example_extraction
TYPE=netcdf make compile
./extract.netcdf
ncdump ./main-update.nc | less
```

4.5.18 Example 18: Special Accesses of Continuous Fields - Incrementing After Reading and Writing Before (Potentially) Reading

Example containing one kernel with a `GH_READINC` access and one with a `GH_WRITE` access, both for continuous fields. A kernel with `GH_READINC` access first reads the field data and then increments the field data. This contrasts with a `GH_INC` access which simply increments the field data. As an increment is effectively a read followed by a write, it may not be clear why we need to distinguish between these cases. The reason for distinguishing is that the `GH_INC` access is able to remove a halo exchange (or at least reduce its depth by one) in certain circumstances, whereas a `GH_READINC` is not able to take advantage of this optimisation.

A kernel with a `GH_WRITE` access for a continuous field must guarantee to write the same value to a given shared DoF, independent of which cell is being updated. As described in the Developer Guide, this means that annexed DoFs are computed correctly without the need to iterate into the L1 halo and thus can remove the need for halo exchanges on those fields that are read.

4.5.19 Example 19: Mixed Precision

This example shows the use of the LFRic *mixed-precision support* to call a kernel with *scalars*, *fields* and *operators* of different precision.

4.5.20 Example 20: Algorithm Generation

Illustration of the use of the `psyclone-kern` tool to create an algorithm layer for a kernel. A makefile is provide that also runs `psyclone` to create an executable program from the generated algorithm layer and original kernel code. To see the generated algorithm layer run:

```
cd eg20/
psyclone-kern -gen alg ../code/testkern_mod.F90
```

4.6 NEMO Examples

These examples may all be found in the `examples/nemo` directory.

4.6.1 Example 1: OpenMP parallelisation of tra_adv

Demonstrates the use of PSyclone to parallelise loops in a NEMO tracer-advection benchmark using OpenMP for CPUs and for GPUs.

4.6.2 Example 2: OpenMP parallelisation of traldf_iso

Demonstrates the use of PSyclone to parallelise in some NEMO tracer-diffusion code using OpenMP for CPUs and for GPUs.

4.6.3 Example 3: OpenACC parallelisation of tra_adv

Demonstrates the introduction of simple OpenACC parallelisation (using the `data` and `kernels` directives) for a NEMO tracer-advection benchmark.

4.6.4 Example 4: Transforming Fortran code to the SIR

Demonstrates that simple Fortran code can be transformed to the Stencil Intermediate Representation (SIR). The SIR is the front-end language to DAWN (https://github.com/MeteoSwiss-APN/dawn), a tool which generates optimised cuda, or gridtools code. Thus various simple Fortran examples and the computational part of the tracer-advection benchmark can be transformed to optimised cuda and/or gridtools code by using PSyclone and then DAWN.

4.6.5 Example 5: Kernel Data Extraction

This example shows the use of kernel data extraction in PSyclone for generic Fortran code. It instruments each kernel in the NEMO tracer-advection benchmark with the PSyData-based kernel extraction code. Detailed compilation instructions are in the `README.md` file, including how to switch from using the stand-alone extraction library to the NetCDF-based one (see *Extraction Libraries* for details).

4.6.6 Scripts

This contains examples of two different scripts that aid the use of PSyclone with the full NEMO model. The first, *process_nemo.py* is a simple wrapper script that allows a user to control which source files are transformed, which only have profiling instrumentation added and which are ignored altogether. The second, *kernels_trans.py* is a PSyclone transformation script which adds the largest possible OpenACC Kernels regions to the code being processed.

For more details see the `examples/nemo/README.md` file.

Note that these scripts are here to support the ongoing development of PSyclone to transform the NEMO source. They are *not* intended as 'turn-key' solutions but as a starting point.

CHAPTER

FIVE

LIBRARIES

PSyclone provides *PSyData-API-based* wrappers to various external libraries. These wrapper libraries provide PSyclone transformations that insert callbacks to an external library at runtime. The callbacks then allow third-party libraries to access data structures at specified locations in the code for different purposes, such as profiling and extraction of argument values.

These wrapper libraries can be found under the `lib` directory in the Git repository. If you have installed PSyclone using `pip` then the libraries may be found in `share/psyclone/lib` in PSyclone *Installation location*.

Note: If working with wrapper libraries from a PSyclone installation, it is advisable to copy the entire `lib` directory to some convenient location before building and using them. The provided `Makefile`s support the options to specify paths to the libraries and their dependencies, see *compilation* for more information.

5.1 Available libraries

An overview of the currently available functionality is below. For details of what each library does and how to build and use it please see the related sections in the User Guide and the specific `README.md` files in the associated directories.

5.1.1 Profiling

PSyclone provides wrapper libraries for some common performance profiling tools, such as dl_timer, TAU, and Dr Hook. More information can be found in the *Profiling* section.

Profiling libraries are located in the `lib/profiling` directory. For detailed instructions on how to build and use them please refer to their specific `README.md` documentation.

5.1.2 Kernel Data Extraction

These libraries enable PSyclone to add callbacks that provide access to all input variables before, and output variables after a kernel invocation. More information can be found in the *PSy Kernel Extractor (PSyKE)* section.

Example libraries that extract input and output data into a NetCDF file for *LFRic* and *GOcean* APIs are included with PSyclone in the `lib/extract/netcdf` directory. For detailed instructions on how to build and use these libraries please refer to their specific `README.md` documentation.

5.1.3 Access Verification

Read-only libraries check that a field declared as read-only is not modified during a kernel call. More information can be found in the *Read-Only Verification* section.

The libraries for *LFRic* and *GOcean* APIs are included with PSyclone in the `lib/read_only` directory. For detailed instructions on how to build and use these libraries please refer to their specific `README.md` documentation.

5.1.4 Value Range Check

These libraries can test if user-defined variables are within a specified range. Additionally, they also verify that they are not `NaN` or infinite. More information can be found in the *Value Range Check* section.

The libraries for *LFRic* and *GOcean* APIs are included with PSyclone in the `lib/value_range_check` directory. For detailed instructions on how to build and use these libraries please refer to their specific `README.md` documentation.

5.2 Dependencies

Building and using the wrapper libraries requires that PSyclone be installed on the host system, see section *Getting Going*. A Fortran compiler (e.g. Gnu Fortran compiler, `gfortran`, is free and easily installed) and Gnu Make are also required.

The majority of wrapper libraries use Jinja templates to create PSyData-derived classes (please refer to psy_data and Jinja Support in the Base Class for full details about the PSyData API).

Compilation of `extract`, `value_range_check`, `read_only` and some of the profiling wrapper libraries depends on infrastructure libraries relevant to the API they are used for. The *LFRic API* uses the LFRic infrastructure and *GOcean* uses the dl_esm_inf library. The LFRic infrastructure can be obtained from the LFRic code repository, however this requires access to the Met Office Science Repository Service (MOSRS). A useful contact for LFRic-related questions (including access to MOSRS) is the "lfric" mailing list which gathers the Met Office and external LFRic developers and users. The dl_esm_inf library is freely available and can be downloaded from https://github.com/stfc/dl_esm_inf.

Some libraries require NetCDF for compilation. Installation of NetCDF is described in details in the hands-on practicals documentation.

Profiling wrapper libraries that depend on external tools (e.g. dl_timer) require these tools be installed and configured beforehand.

5.3 Compilation

Each library is compiled with `make` using the provided `Makefile` that has configurable options for compiler flags and locations of dependencies.

As in case of *examples*, `F90` and `F90FLAGS` specify the compiler and compilation flags to use. The default value for `F90` is `gfortran`.

Locations of the top-level `lib` directory and the required Jinja templates are specified with the `PSYDATA_LIB_DIR` and `LIB_TMPLT_DIR` variables. For testing purposes their default values are set to relative paths to the respective directories in the PSyclone repository.

The locations of the infrastructure libraries for LFRic and GOcean applications can be configured with the variables `LFRIC_INF_DIR` and `GOCEAN_INF_DIR`, respectively. Their default values are set to relative paths to the locations of these libraries in the PSyclone repository. The dl_esm_inf library is provided as a Git submodule of the PSyclone project (see Installation in the Developers' Guide for details on working with submodules) and a pared-down version of LFRic

infrastructure is also available in the PSyclone repository (please refer to the `README.md` documentation of relevant wrapper libraries). However, the infrastructure libraries are not available in a PSyclone installation and they need to be downloaded separately, see *Dependencies* for more information. In this case `LFRIC_INF_DIR` and `GOCEAN_INF_DIR` **must be set** to the exact paths to where the respective infrastructure source can be found. For instance,

```
GOCEAN_INF_DIR=$HOME/dl_esm_inf/finite_difference make
```

Profiling wrapper libraries that depend on external tools have specific variables that configure paths to where these libraries are located in a user environment.

For more information on how to build and configure a specific library please refer to its `README.md` documentation.

Similar to compilation of the *examples*, the compiled library can be removed by running `make clean`. There is also the `allclean` target that removes the compiled wrapper library as well as the compiled infrastructure library that the wrapper may depend on.

The compilation of wrapper libraries was tested with the Gnu and Intel Fortran compilers, see *here* for the full list. Please let the PSyclone developers know if you have problems using a compiler that has been tested or if you are working with a different compiler.

CHAPTER

SIX

PSYIR: THE PSYCLONE INTERMEDIATE REPRESENTATION

The PSyIR is at the heart of PSyclone, representing code for existing code and PSyKAl DSLs (at both the PSy- and kernel-layer levels). A PSyIR tree may be constructed from scratch (in Python) or by processing existing source code using a frontend. Transformations act on the PSyIR and ultimately the generated code is produced by one of the PSyIR's backends.

6.1 PSyIR Nodes

The PSyIR consists of classes whose instances can be connected together to form a tree which represent computation in a syntax-independent way. These classes all inherit from the `Node` baseclass and, as a result, PSyIR instances are often referred to collectively as 'PSyIR nodes'.

At the present time PSyIR classes can be essentially split into two types: language-level nodes, which are nodes that the PSyIR backends support, and therefore they can be directly translated to code; and higher-level nodes, which are additional nodes that each domain can insert. These nodes must implement a *lower_to_language_level* method in order to be converted to their equivalent representation using only language-level nodes. This then permits code to be generated for them.

The rest of this document describes only the language-level nodes, but as all nodes inherit from the same base classes, the methods described here are applicable to all PSyIR nodes.

6.2 Available language-level nodes

- ArrayMember
- ArrayReference
- ArrayOfStructuresMember
- ArrayOfStructuresReference
- Assignment
- BinaryOperation
- Call
- CodeBlock
- Container
- FileContainer
- IfBlock

- IntrinsicCall
- Literal
- Loop
- Member
- Node
- Range
- Reference
- Return
- Routine
- Schedule
- Statement
- StructureMember
- StructureReference
- UnaryOperation
- WhileLoop

6.3 Text Representation

When developing a transformation script it is often necessary to examine the structure of the PSyIR. All nodes in the PSyIR have the `view` method that provides a text-representation of that node and all of its descendants. If the `termcolor` package is installed (see *Getting Going*) then colour highlighting is used as part of the output string. For instance, part of the Schedule constructed for the second NEMO example is rendered as:

```
14: If[]
    BinaryOperation[operator:'OR']
        BinaryOperation[operator:'AND']
            BinaryOperation[operator:'EQ']
                Reference[name:'kpass']
                Literal[value:'1', Scalar<INTEGER, UNDEFINED>]
            Reference[name:'ln_traldf_lap']
        BinaryOperation[operator:'AND']
            BinaryOperation[operator:'EQ']
                Reference[name:'kpass']
                Literal[value:'2', Scalar<INTEGER, UNDEFINED>]
            Reference[name:'ln_traldf_blp']
    Schedule[]
        0: If[annotations='was_single_stmt']
            Reference[name:'l_ptr']
            Schedule[]
                0: Call[name='dia_ptr_hst']
                    Reference[name:'dia_ptr_hst']
                    Reference[name:'jn']
                    Literal[value:'ldf', Scalar<CHARACTER, UNDEFINED>]
                    UnaryOperation[operator:'MINUS']
                        ArrayReference[name:'zftv']
                            Range[]
                                IntrinsicCall[name='LBOUND']
                                    Reference[name:'LBOUND']
                                    Reference[name:'zftv']
                                    Literal[value:'1', Scalar<INTEGER, UNDEFINED>]
                                IntrinsicCall[name='UBOUND']
                                    Reference[name:'UBOUND']
                                    Reference[name:'zftv']
                                    Literal[value:'1', Scalar<INTEGER, UNDEFINED>]
                                Literal[value:'1', Scalar<INTEGER, UNDEFINED>]
```

Note that in this view, only those nodes which are children of Schedules have their indices shown. This means that nodes representing e.g. loop bounds or the conditional part of `if` statements are not indexed. For the example shown, the PSyIR node representing the `if(l_hst)` code would be reached by `schedule.children[14].if_body.children[1]` or, using the shorthand notation (see below), `schedule[14].if_body[1]` where `schedule` is the overall parent Schedule node (omitted from the above image).

One problem with the `view` method is that the output can become very large for big ASTs and is not readable for users unfamiliar with the PSyIR. An alternative to it is the `debug_string` method that generates a text representation with Fortran-like syntax but on which the high abstraction constructs have not yet been lowered to Fortran level and instead they will be embedded as < *node* > expressions.

6.4 Tree Navigation

Each PSyIR node provides several ways to navigate the AST. These can be categorised as homogeneous naviation methods (available in all nodes), and heterogenous or semantic navigation methods (different methods available depending on the node type). The homogeneous methods must be used for generic code navigation that should work regardless of its context. However, when the context is known, we recommend using the semantic methods to increase the code readability.

The homogeneous navigation methods are:

`Node.`**`children`**`()`

Returns the immediate children of this Node.

Return type List[psyclone.psyir.nodes.Node]

Node.**siblings**()

Returns list of sibling nodes, including self.

Return type List[psyclone.psyir.nodes.Node]

Node.**parent**()

Returns the parent node.

Return type psyclone.psyir.nodes.Node or NoneType

Node.**root**()

Returns the root node of the PSyIR tree.

Return type psyclone.psyir.nodes.Node

Node.**walk**()

Recurse through the PSyIR tree and return all objects that are an instance of 'my_type', which is either a single class or a tuple of classes. In the latter case all nodes are returned that are instances of any classes in the tuple. The recursion into the tree is stopped if an instance of 'stop_type' (which is either a single class or a tuple of classes) is found. This can be used to avoid analysing e.g. inlined kernels, or as performance optimisation to reduce the number of recursive calls. The recursion into the tree is also stopped if the (optional) 'depth' level is reached.

Parameters

- **my_type** (*type | Tuple[type, ...]*) – the class(es) for which the instances are collected.
- **stop_type** (*Optional[type | Tuple[type, ...]]*) – class(es) at which recursion is halted (optional).
- **depth** (*Optional[int]*) – the depth value the instances must have (optional).

Returns list with all nodes that are instances of my_type starting at and including this node.

Return type List[psyclone.psyir.nodes.Node]

Node.**get_sibling_lists**()

Recurse through the PSyIR tree and return lists of Nodes that are instances of 'my_type' and are immediate siblings. Here 'my_type' is either a single class or a tuple of classes. In the latter case all nodes are returned that are instances of any classes in the tuple. The recursion into the tree is stopped if an instance of 'stop_type' (which is either a single class or a tuple of classes) is found.

Parameters

- **my_type** (*type | Tuple[type, ...]*) – the class(es) for which the instances are collected.
- **stop_type** (*Optional[type | Tuple[type, ...]]*) – class(es) at which recursion is halted (optional).

Returns list of lists, each of which containing nodes that are instances of my_type and are immediate siblings, starting at and including this node.

Return type List[List[psyclone.psyir.nodes.Node]]

Node.**ancestor**()

Search back up the tree and check whether this node has an ancestor that is an instance of the supplied type. If it does then we return it otherwise we return None. An individual (or tuple of) (sub-) class(es) to ignore may be provided via the *excluding* argument. If *include_self* is True then the current node is included in the search. If *limit* is provided then the search ceases if/when the supplied node is encountered. If *shared_with* is provided, then the ancestor search will find an ancestor of both this node and the node provided as *shared_with* if such an ancestor exists.

Parameters

- **my_type** (*type | Tuple[type, ...]*) – class(es) to search for.
- **excluding** (*Optional[type | Tuple[type, ...]]*) – (sub-)class(es) to ignore or None.
- **include_self** (*bool*) – whether or not to include this node in the search.
- **limit** (Optional[psyclone.psyir.nodes.Node]) – an optional node at which to stop the search.
- **shared_with** (Optional[psyclone.psyir.nodes.Node]) – an optional node which must also have the found node as an ancestor.

Returns First ancestor Node that is an instance of any of the requested classes or None if not found.

Return type Optional[psyclone.psyir.nodes.Node]

Raises

- **TypeError** – if *excluding* is provided but is not a type or tuple of types.
- **TypeError** – if *limit* is provided but is not an instance of Node.

Node.**scope**()

Some nodes (e.g. Schedule and Container) allow symbols to be scoped via an attached symbol table. This property returns the closest ScopingNode node including self.

Returns the closest ancestor ScopingNode node.

Return type psyclone.psyir.node.ScopingNode

Raises **SymbolError** – if there is no ScopingNode ancestor.

Node.**path_from**()

Find the path in the psyir tree between ancestor and node and returns a list containing the path.

The result of this method can be used to find the node from its ancestor for example by:

```
>>> index_list = node.path_from(ancestor)
>>> cursor = ancestor
>>> for index in index_list:
>>>     cursor = cursor.children[index]
>>> assert cursor is node
```

Parameters **ancestor** (psyclone.psyir.nodes.Node) – an ancestor node of self to find the path from.

Raises **ValueError** – if ancestor is not an ancestor of self.

Returns a list of child indices representing the path between ancestor and self.

Return type List[int]

In addition to the navigation methods, nodes also have homogeneous methods to interrogate their location and surrounding nodes.

Node.**immediately_precedes**()

Returns True if this node immediately precedes *node_2*, False otherwise

Return type bool

Node.**immediately_follows**()

Returns True if this node immediately follows *node_1*, False otherwise

Return type bool

Node.**position**()
Find a Node's position relative to its parent Node (starting with 0 if it does not have a parent).

Returns relative position of a Node to its parent

Return type int

Node.**abs_position**()
Find a Node's absolute position in the tree (starting with 0 if it is the root). Needs to be computed dynamically from the starting position (0) as its position may change.

Returns absolute position of a Node in the tree.

Return type int

Raises **InternalError** – if the absolute position cannot be found.

Node.**sameParent**()

Returns True if *node_2* has the same parent as this node, False otherwise.

Return type bool

The semantic navigation methods are:

- **Schedule:** subscript operator for indexing the statements (children) inside the Schedule, e.g. `sched[3]` or `sched[2:4]`.
- **Assignment:**

  Assignment.**lhs**()

  Returns the child node representing the Left-Hand Side of the assignment.
  Return type psyclone.psyir.nodes.Node
  Raises **InternalError** – Node has fewer children than expected.

  Assignment.**rhs**()

  Returns the child node representing the Right-Hand Side of the assignment.
  Return type psyclone.psyir.nodes.Node
  Raises **InternalError** – Node has fewer children than expected.

- **IfBlock:**

  IfBlock.**condition**()
  Return the PSyIR Node representing the conditional expression of this IfBlock.
  Returns IfBlock conditional expression.

Return type psyclone.psyir.nodes.Node
Raises **InternalError** – If the IfBlock node does not have the correct number of children.

IfBlock.**if_body**()
Return the Schedule executed when the IfBlock evaluates to True.
Returns Schedule to be executed when IfBlock evaluates to True.
Return type psyclone.psyir.nodes.Schedule
Raises **InternalError** – If the IfBlock node does not have the correct number of children.

IfBlock.**else_body**()
If available return the Schedule executed when the IfBlock evaluates to False, otherwise return None.
Returns Schedule to be executed when IfBlock evaluates to False, if it doesn't exist returns None.
Return type psyclone.psyir.nodes.Schedule or NoneType

- **Loop:**

Loop.**loop_body**()

Returns the PSyIR Schedule with the loop body statements.
Return type psyclone.psyir.nodes.Schedule

- **WhileLoop:**

WhileLoop.**condition**()
Return the PSyIR Node representing the conditional expression of this WhileLoop.
Returns WhileLoop conditional expression.
Return type psyclone.psyir.nodes.Node
Raises **InternalError** – If the WhileLoop node does not have the correct number of children.

WhileLoop.**loop_body**()
Return the Schedule executed when the WhileLoop condition is True.
Returns Schedule to be executed when WhileLoop condition is True.
Return type psyclone.psyir.nodes.Schedule
Raises **InternalError** – If the WhileLoop node does not have the correct number of children.

- **Array nodes (e.g. ArrayReference, ArrayOfStructuresReference):**

ArrayReference.**indices**()
Supports semantic-navigation by returning the list of nodes representing the index expressions for this array reference.
Returns the PSyIR nodes representing the array-index expressions.
Return type list of psyclone.psyir.nodes.Node
Raises **InternalError** – if this node has no children or if they are not valid array-index expressions.

- **RegionDirective:**

RegionDirective.**dir_body**()

Returns the Schedule associated with this directive.
Return type psyclone.psyir.nodes.Schedule
Raises **InternalError** – if this node does not have a Schedule as its first child.

RegionDirective.**clauses**()

Returns the Clauses associated with this directive.
Return type List of psyclone.psyir.nodes.Clause

- **Nodes representing accesses of data within a structure (e.g. `StructureReference`, `StructureMember`):**

  `StructureReference.member()`

  Returns the PSyIR child representing the accessor component.
  Return type `psyclone.psyir.nodes.Member`
  Raises `InternalError` – if the first child of this node is not an instance of Member.

6.5 DataTypes

The PSyIR supports the following datatypes: `ScalarType`, `ArrayType`, `StructureType`, `UnresolvedType`, `UnsupportedType` and `NoType`. These datatypes are used when creating instances of DataSymbol, RoutineSymbol and Literal (although note that `NoType` may only be used with a RoutineSymbol). `UnresolvedType` and `UnsupportedType` are both used when processing existing code. The former is used when a symbol is being imported from some other scope (e.g. via a USE statement in Fortran) that hasn't yet been resolved and the latter is used when an unsupported form of declaration is encountered.

More information on each of these various datatypes is given in the following subsections.

6.5.1 Scalar DataType

A Scalar datatype consists of an intrinsic and a precision.

The intrinsic can be one of `INTEGER`, `REAL`, `BOOLEAN` and `CHARACTER`.

The precision can be `UNDEFINED`, `SINGLE`, `DOUBLE`, an integer value specifying the precision in bytes, or a datasymbol (see Section *Symbols and Symbol Tables*) that contains precision information. Note that `UNDEFINED`, `SINGLE` and `DOUBLE` allow the precision to be set by the system so may be different for different architectures. For example:

```
>>> char_type = ScalarType(ScalarType.Intrinsic.CHARACTER,
...                        ScalarType.Precision.UNDEFINED)
>>> int_type = ScalarType(ScalarType.Intrinsic.INTEGER,
...                       ScalarType.Precision.SINGLE)
>>> bool_type = ScalarType(ScalarType.Intrinsic.BOOLEAN, 4)
>>> symbol = DataSymbol("rdef", int_type, initial_value=4)
>>> scalar_type = ScalarType(ScalarType.Intrinsic.REAL, symbol)
```

For convenience PSyclone predefines a number of scalar datatypes:

`REAL_TYPE`, `INTEGER_TYPE`, `BOOLEAN_TYPE` and `CHARACTER_TYPE` all have precision set to `UNDEFINED`;

`REAL_SINGLE_TYPE`, `REAL_DOUBLE_TYPE`, `INTEGER_SINGLE_TYPE` and `INTEGER_DOUBLE_TYPE`;

`REAL4_TYPE`, `REAL8_TYPE`, `INTEGER4_TYPE` and `INTEGER8_TYPE`.

6.5.2 Array DataType

An Array datatype itself has another datatype (or `DataTypeSymbol`) specifying the type of its elements and a shape. The shape can have an arbitrary number of dimensions. Each dimension captures what is known about its extent. It is necessary to distinguish between four cases:

Description	Entry in shape list
An array has a static extent known at compile time.	`ArrayType.ArrayBounds` containing integer `Literal` values
An array has an extent defined by another symbol or (constant) PSyIR expression.	`ArrayType.ArrayBounds` containing `Reference` or `Operation` nodes
An array has a definite extent which is not known at compile time but can be queried at runtime.	`ArrayType.Extent.ATTRIBUTE`
It is not known whether an array has memory allocated to it in the current scoping unit.	`ArrayType.Extent.DEFERRED`

where `ArrayType.ArrayBounds` is a `namedtuple` with `lower` and `upper` members holding the lower- and upper-bounds of the extent of a given array dimension.

The distinction between the last two cases is that in the former the extents are known but are kept internally with the array (for example an assumed shape array in Fortran) and in the latter the array has not yet been allocated any memory (for example the declaration of an allocatable array in Fortran) so the extents may have not been defined yet.

For example:

```
>>> array_type = ArrayType(REAL4_TYPE, [5, 10])

>>> n_var = DataSymbol("n", INTEGER_TYPE)
>>> array_type = ArrayType(INTEGER_TYPE, [Reference(n_var),
...                                       Reference(n_var)])

>>> array_type = ArrayType(REAL8_TYPE, [ArrayType.Extent.ATTRIBUTE,
...                                     ArrayType.Extent.ATTRIBUTE])

>>> array_type = ArrayType(BOOLEAN_TYPE, [ArrayType.Extent.DEFERRED])
```

6.5.3 Structure Datatype

A Structure datatype consists of a dictionary of components where the name of each component is used as the corresponding key. Each component is stored as a named tuple with `name`, `datatype` and `visibility` members.

For example:

```
# Shorthand for a scalar type with REAL_KIND precision
SCALAR_TYPE = ScalarType(ScalarType.Intrinsic.REAL, REAL_KIND)

# Structure-type definition
GRID_TYPE = StructureType.create([
    ("dx", SCALAR_TYPE, Symbol.Visibility.PUBLIC),
    ("dy", SCALAR_TYPE, Symbol.Visibility.PUBLIC)])

GRID_TYPE_SYMBOL = DataTypeSymbol("grid_type", GRID_TYPE)
```

(continues on next page)

(continued from previous page)

```
# A structure-type containing other structure types
FIELD_TYPE_DEF = StructureType.create(
    [("data", ArrayType(SCALAR_TYPE, [10]), Symbol.Visibility.PUBLIC),
     ("grid", GRID_TYPE_SYMBOL, Symbol.Visibility.PUBLIC),
     ("sub_meshes", ArrayType(GRID_TYPE_SYMBOL, [3]),
      Symbol.Visibility.PUBLIC),
     ("flag", INTEGER4_TYPE, Symbol.Visibility.PUBLIC)])
```

6.5.4 Unknown DataType

If a PSyIR frontend encounters an unsupported declaration then the corresponding Symbol is given UnsupportedType. The text of the original declaration is stored in the type object and is available via the `declaration` property.

6.5.5 NoType

`NoType` represents the empty type, equivalent to `void` in C. It is currently only used to describe a RoutineSymbol that has no return type (such as a Fortran subroutine).

6.6 Symbols and Symbol Tables

Some PSyIR nodes have an associated Symbol Table (*psyclone.psyir.symbols.SymbolTable*) which keeps a record of the Symbols (*psyclone.psyir.symbols.Symbol*) specified and used within them.

Symbol Tables can be nested (i.e. a node with an attached symbol table can be an ancestor or descendent of a node with an attached symbol table). If the same symbol name is used in a hierarchy of symbol tables then the symbol within the symbol table attached to the closest ancestor node is in scope. By default, symbol tables are aware of other symbol tables and will return information about relevant symbols from all symbol tables.

The `SymbolTable` has the following interface:

class psyclone.psyir.symbols.**SymbolTable**(*node=None*, *default_visibility=Visibility.PUBLIC*)

Encapsulates the symbol table and provides methods to add new symbols and look up existing symbols. Nested scopes are supported and, by default, the add and lookup methods take any ancestor symbol tables into consideration (ones attached to nodes that are ancestors of the node that this symbol table is attached to). If the default visibility is not specified then it defaults to Symbol.Visbility.PUBLIC.

Parameters

- **node** (Optional[psyclone.psyir.nodes.Schedule | psyclone.psyir.nodes.Container]) – reference to the Schedule or Container to which this symbol table belongs.
- **default_visibility** – optional default visibility value for this symbol table, if not provided it defaults to PUBLIC visibility.

Raises TypeError – if node argument is not a Schedule or a Container.

Where each element is a `Symbol` with an immutable name:

class psyclone.psyir.symbols.**Symbol**(*name*, *visibility=Visibility.PUBLIC*, *interface=None*)

Generic Symbol item for the Symbol Table and PSyIR References. It has an immutable name label because it must always match with the key in the SymbolTable. If the symbol is private then it is only visible to those nodes that are descendants of the Node to which its containing Symbol Table belongs.

Parameters

- **name** (*str*) – name of the symbol.
- **visibility** (psyclone.psyir.symbols.Symbol.Visibility) – the visibility of the symbol.
- **interface** (Optional[psyclone.psyir.symbols.symbol.SymbolInterface]) – optional object describing the interface to this symbol (i.e. whether it is passed as a routine argument or accessed in some other way). Defaults to *psyclone.psyir.symbols.AutomaticInterface*

Raises TypeError – if the name is not a str.

There are several Symbol sub-classes to represent different labeled entities in the PSyIR. At the moment the available symbols are:

- **class** psyclone.psyir.symbols.**ContainerSymbol**(*name*, ***kwargs*)
  Symbol that represents a reference to a Container. The reference is lazy evaluated, this means that the Symbol will be created without parsing and importing the referenced container, but this can be imported when needed.

  Parameters

  - **name** (*str*) – name of the symbol.
  - **wildcard_import** (*bool*) – if all public Symbols of the Container are imported into the current scope. Defaults to False.
  - **is_intrinsic** (*bool*) – if the module is an intrinsic import. Defauts to False.
  - **kwargs** (*unwrapped dict.*) – additional keyword arguments provided by *psyclone.psyir.symbols.Symbol*.

- **class** psyclone.psyir.symbols.**DataSymbol**(*name*, *datatype*, *is_constant=False*, *initial_value=None*, ***kwargs*)
  Symbol identifying a data element. It contains information about: the datatype, the shape (in column-major order) and the interface to that symbol (i.e. Local, Global, Argument).

  Parameters

  - **name** (*str*) – name of the symbol.
  - **datatype** (psyclone.psyir.symbols.DataType) – data type of the symbol.
  - **is_constant** (*bool*) – whether this DataSymbol is a compile-time constant (default is False). If True then an *initial_value* must also be provided.
  - **initial_value** (Optional[item of TYPE_MAP_TO_PYTHON | psyclone.psyir.nodes.Node]) – sets a fixed known expression as an initial value for this DataSymbol. If *is_constant* is True then this Symbol will always have this value. If the value is None then this symbol does not have an initial value (and cannot be a constant). Otherwise it can receive PSyIR expressions or Python intrinsic types available in the TYPE_MAP_TO_PYTHON map. By default it is None.
  - **kwargs** (*unwrapped dict.*) – additional keyword arguments provided by psyclone.psyir.symbols.TypedSymbol

- **class** psyclone.psyir.symbols.**DataTypeSymbol**(*name*, *datatype*, *visibility=Visibility.PUBLIC*, *interface=None*)
  Symbol identifying a user-defined type (e.g. a derived type in Fortran).

  Parameters

  - **name** (*str*) – the name of this symbol.

- **datatype** (psyclone.psyir.symbols.DataType) – the type represented by this symbol.
- **visibility** (psyclone.psyir.symbols.Symbol.Visibility) – the visibility of this symbol.
- **interface** (psyclone.psyir.symbols.SymbolInterface) – the interface to this symbol.

- **class** psyclone.psyir.symbols.**IntrinsicSymbol**(*name*, *intrinsic*, ***kwargs*)
  Symbol identifying a callable intrinsic routine.

  Parameters

  - **name** (*str*) – name of the symbol.
  - **intrinsic** (psyclone.psyir.nodes.IntrinsicCall.Intrinsic) – the intrinsic enum describing this Symbol.
  - **kwargs** (*unwrapped dict.*) – additional keyword arguments provided by psyclone.psyir.symbols.TypedSymbol

  # TODO #2541: Currently name and the intrinsic should match, we really # just need the name, and make all the Intrinsic singature information # live inside the IntrinsicSymbol class.

- **class** psyclone.psyir.symbols.**RoutineSymbol**(*name*, *datatype=None*, ***kwargs*)
  Symbol identifying a callable routine.

  Parameters

  - **name** (*str*) – name of the symbol.
  - **datatype** (psyclone.psyir.symbols.DataType) – data type of the symbol. Default to NoType().
  - **kwargs** (*unwrapped dict.*) – additional keyword arguments provided by psyclone.psyir.symbols.TypedSymbol

- **class** psyclone.psyir.symbols.**GenericInterfaceSymbol**(*name*, *routines*, ***kwargs*)
  Symbol identifying a generic interface that maps to a number of different callable routines.

  Parameters

  - **name** (*str*) – name of the interface.
  - **routines** (list[tuple[*psyclone.psyir.symbols.RoutineSymbol*, bool]]) – the routines that this interface provides access to.
  - **kwargs** (*unwrapped dict.*) – additional keyword arguments provided by psyclone.psyir.symbols.TypedSymbol

See the reference guide for the full API documentation of the SymbolTable and the Symbol types.

6.6.1 Symbol Interfaces

Each symbol has a Symbol Interface with the information about how the variable data is provided into the local context. The currently available Interfaces are:

- **class** psyclone.psyir.symbols.**AutomaticInterface**
  The symbol is declared without attributes. Its data will live during the local context.
- **class** psyclone.psyir.symbols.**DefaultModuleInterface**
  The symbol contains data declared in a module scope without additional attributes.
- **class** psyclone.psyir.symbols.**ImportInterface**(*container_symbol*, *orig_name=None*)
  Describes the interface to a Symbol that is imported from an external PSyIR container. The symbol can be renamed on import and, if so, its original name in the Container is specified using the optional 'orig_name' argument.

  Parameters

  - **container_symbol** (*psyclone.psyir.symbols.ContainerSymbol*) – symbol representing the external container from which the symbol is imported.
  - **orig_name** (*Optional[str]*) – the name of the symbol in the external container before it is renamed, or None (the default) if it is not renamed.

  Raises TypeError – if the orig_name argument is an unexpected type.
- **class** psyclone.psyir.symbols.**ArgumentInterface**(*access=None*)
  Captures the interface to a Symbol that is accessed as a routine argument.

  Parameters **access** (psyclone.psyir.symbols.ArgumentInterface.Access) – specifies how the argument is used in the Schedule
- **class** psyclone.psyir.symbols.**StaticInterface**
  The symbol contains data that is kept alive through the execution of the program.
- **class** psyclone.psyir.symbols.**CommonBlockInterface**
  A symbol declared in the local scope but acts as a global that can be accessed by any scope referencing the same CommonBlock name.
- **class** psyclone.psyir.symbols.**UnresolvedInterface**
  We have a symbol but we don't know where it is declared.
- **class** psyclone.psyir.symbols.**UnknownInterface**
  We have a symbol with a declaration but PSyclone does not support its attributes.
- **class** psyclone.psyir.symbols.**PreprocessorInterface**
  The symbol exists in the file through compiler macros or preprocessor directives.

  Note that this is different from UnresolvedInterface because the backend will not check if is importing statements that could bring them into scope.

6.7 Creating PSyIR

6.7.1 Symbol names

PSyIR symbol names can be specified by a user. For example:

```
var_name = "my_name"
symbol_table = SymbolTable()
data = DataSymbol(var_name, REAL_TYPE)
symbol_table.add(data)
reference = Reference(data)
```

However, the `SymbolTable` `add()` method will raise an exception if a user tries to add a symbol with the same name as a symbol already existing in the symbol table.

Alternatively, the `SymbolTable` also provides the `new_symbol()` method (see Section *Symbols and Symbol Tables* for more details) that uses a new distinct name from any existing names in the symbol table. By default the generated name is the value `PSYIR_ROOT_NAME` variable specified in the `DEFAULT` section of the PSyclone config file, followed by an optional "_" and an integer. For example, the following code:

```
from psyclone.psyir.symbols import SymbolTable
symbol_table = SymbolTable()
for i in range(0, 3):
    var_name = symbol_table.new_symbol().name
    print(var_name)
```

gives the following output:

```
psyir_tmp
psyir_tmp_0
psyir_tmp_1
```

As the root name (`psyir_tmp` in the example above) is specified in PSyclone's config file it can be set to whatever the user wants.

Note: The particular format used to create a unique name is the responsibility of the SymbolTable class and may change in the future.

A user might want to create a name that has some meaning in the context in which it is used e.g. `idx` for an index, `i` for an iterator, or `temp` for a temperature field. To support more readable names, the `new_symbol()` method allows the user to specify a root name as an argument to the method which then takes the place of the default root name. For example, the following code:

```
from psyclone.psyir.symbols import SymbolTable
symbol_table = SymbolTable()
for i in range(0, 3):
    var_name = symbol_table.new_symbol(root_name="something")
    print(var_name)
```

gives the following output:

```
something
something_0
something_1
```

By default, `new_symbol()` creates generic symbols, but often the user will want to specify a Symbol subclass with some given parameters. The `new_symbol()` method accepts a `symbol_type` parameter to specify the subclass. Arguments for the constructor of that subclass may be supplied as keyword arguments. For example, the following code:

```
from psyclone.psyir.symbols import SymbolTable, DataSymbol, REAL_TYPE
symbol_table = SymbolTable()
symbol_table.new_symbol(root_name="something",
                        symbol_type=DataSymbol,
                        datatype=REAL_TYPE,
                        is_constant=True,
                        initial_value=3)
```

declares a symbol named "something" of REAL_TYPE datatype where the `is_constant` and `initial_value` arguments will be passed to the DataSymbol constructor.

An example of using the `new_symbol()` method can be found in the PSyclone `examples/psyir` directory.

6.7.2 Nodes

PSyIR nodes are connected together via parent and child methods provided by the `Node` baseclass.

These nodes can be created in isolation and then connected together. For example:

```
assignment = Assignment()
literal = Literal("0.0", REAL_TYPE)
reference = Reference(symbol)
assignment.children = [reference, literal]
```

However, as connections get more complicated, creating the correct connections can become difficult to manage and error prone. Further, in some cases children must be collected together within a `Schedule` (e.g. for `IfBlock`, `Loop` and `WhileLoop`).

To simplify this complexity, each of the Kernel-layer nodes which contain other nodes have a static `create` method which helps construct the PSyIR using a bottom up approach. Using this method, the above example then becomes:

```
literal = Literal("0.0", REAL_TYPE)
reference = Reference(symbol)
assignment = Assignment.create(reference, literal)
```

Creating the PSyIR to represent a complicated access of a member of a structure is best performed using the `create()` method of the appropriate `Reference` subclass. For a relatively straightforward access such as (the Fortran) `field1%region%nx`, this would be:

```
from psyclone.psyir.nodes import StructureReference
fld_sym = symbol_table.lookup("field1")
ref = StructureReference.create(fld_sym, ["region", "nx"])
```

where `symbol_table` is assumed to be a pre-populated Symbol Table containing an entry for "field1".

A more complicated access involving arrays of structures such as `field1%sub_grids(idx, 1)%nx` would be constructed as:

```
from psyclone.psyir.symbols import INTEGER_TYPE
from psyclone.psyir.nodes import StructureReference, Reference, Literal
idx_sym = symbol_table.lookup("idx")
fld_sym = symbol_table.lookup("field1")
ref = StructureReference.create(fld_sym,
    [("sub_grids", [Reference(idx_sym), Literal("1", INTEGER_TYPE)]),
     "nx"])
```

Note that the list of quantities passed to the `create()` method now contains a 2-tuple in order to describe the array access.

More examples of using this approach can be found in the PSyclone `examples/psyir` directory.

6.8 Comparing PSyIR nodes

The == (equality) operator for PSyIR nodes performs a specialised equality check to compare the value of each node. This is also useful when comparing entire subtrees since the equality operator automatically recurses through the children and compares each child with the appropriate equality semantics, e.g.

```
# Is the loop upper bound expression exactly the same?
if loop1.stop_expr == loop2.stop_expr:
        print("Same upper bound!")
```

The equality operator will handle expressions like `my_array%my_field(:3)` with the derived type fields and the range components automatically, but it cannot handle symbolically equivalent fields, i.e. `my_array%my_field(:3) != my_array%my_field(:2+1)`.

Annotations and code comments are ignored in the equality comparison since they don't alter the semantic meaning of the code. So these two statements compare to True:

```
a = a + 1
a = a + 1 !Increases a by 1
```

Sometimes there are cases where one really means to check for the specific instance of a node. In this case, Python provides the `is` operator, e.g.

```
# Is the self instance part of this routine?
is_here = any(node is self for node in routine.walk(Node))
```

Additionally, PSyIR nodes cannot be used as map keys or similar. The easiest way to do this is just use the id as the key:

```
node_map = {}
node_map[id(mynode)] = "element"
```

6.9 Modifying the PSyIR

Once we have a complete PSyIR AST there are 2 ways to modify its contents and/or structure: by applying transformations (see next section *Transformations*), or by direct PSyIR API methods. This section describes some of the methods that the PSyIR classes provide to modify the PSyIR AST in a consistent way (e.g. without breaking its many internal references). Some complete examples of modifying the PSyIR can be found in the PSyclone `examples/psyir/modify.py` script.

The rest of this section introduces examples of the available direct PSyIR modification methods.

6.9.1 Renaming symbols

The symbol table provides the method `rename_symbol()` that given a symbol and an unused name will rename the symbol. The symbol renaming will affect all the references in the PSyIR AST to that symbol. For example, the PSyIR representing the following Fortran code:

```
subroutine work(psyir_tmp)
    real, intent(inout) :: psyir_tmp
    psyir_tmp=0.0
end subroutine
```

could be modified by the following PSyIR statements:

```
symbol = symbol_table.lookup("psyir_tmp")
symbol_table.rename_symbol(tmp_symbol, "new_variable")
```

which would result in the following Fortran output code:

```
subroutine work(new_variable)
    real, intent(inout) :: new_variable
    new_variable=0.0
end subroutine
```

6.9.2 Specialising symbols

The Symbol class provides the method `specialise()` that given a subclass of Symbol will change the Symbol instance to the specified subclass. If the subclass has any additional properties then these would need to be set explicitly.

```
symbol = Symbol("name")
symbol.specialise(RoutineSymbol)
# Symbol is now a RoutineSymbol
```

This method is useful as it allows the class of a symbol to be changed without affecting any references to it.

6.9.3 Replacing PSyIR nodes

In certain cases one might want to replace a node in a PSyIR tree with another node. All nodes provide the *replace_with()* method to replace the node and its descendants with another given node and its descendants.

```
node.replace_with(new_node)
```

When the node being replaced is part of a named context (in Calls or Operations) the name of the argument is conserved by default. For example

```
call named_subroutine(name1=1)
```

```
call.children[0].replace_with(Literal('2', INTEGER_TYPE))
```

will become:

```
call named_subroutine(name1=2)
```

This behaviour can be changed with the *keep_name_in_context* parameter.

```
call.children[0].replace_with(
    Literal('3', INTEGER_TYPE),
    keep_name_in_context=False
)
```

will become:

```
call named_subroutine(3)
```

6.9.4 Detaching PSyIR nodes

Sometimes we just may wish to detach a certain PSyIR subtree in order to remove it from the root tree but we don't want to delete it altogether, as it may be re-inserted again in another location. To achieve this, all nodes provide the detach method:

```
tmp = node.detach()
```

6.9.5 Copying nodes

Copying a PSyIR node and its children is often useful in order to avoid repeating the creation of similar PSyIR subtrees. The result of the copy allows the modification of the original and the copied subtrees independently, without altering the other subtree. Note that this is not equivalent to the Python `copy` or `deepcopy` functionality provided in the `copy` library. This method performs a bespoke copy operation where some components of the tree, like children, are recursively copied, while others, like the top-level parent reference are not.

```
new_node = node.copy()
```

6.9.6 Named arguments

The *Call* node (and its sub-classes) support named arguments.

Named arguments can be set or modified via the *create()*, *append_named_arg()*, *insert_named_arg()* or *replace_named_arg()* methods.

If an argument is inserted directly (via the children list) then it is assumed that this is not a named argument. If the top node of an argument is replaced by removing and inserting a new node then it is assumed that this argument is no longer a named argument. If it is replaced with the *replace_with* method, it has a *keep_name_in_context* argument to choose the desired behaviour (defaults to True). If arguments are re-ordered then the names follow the re-ordering.

The names of named arguments can be accessed via the *argument_names* property. This list has an entry for each argument and either contains a name or None (if this is not a named argument).

The PSyIR does not constrain which arguments are specified as being named and what those names are. It is the developer's responsibility to make sure that these names are consistent with any intrinsics that will be generated by the back-end. In the future, it is expected that the PSyIR will know about the number and type of arguments expected by Operation nodes, beyond simply being unary, binary or nary.

One restriction that Fortran has (but the PSyIR does not) is that all named arguments should be at the end of the argument list. If this is not the case then the Fortran backend writer will raise an exception.

CHAPTER

SEVEN

TRANSFORMATIONS

As discussed in the previous section, transformations can be applied to the PSyIR to modify it. Typically transformations will be used to optimise the provided source file, or the PSy and/or Kernel layer(s) in the PSyKAl DSLs, for a particular architecture. However, transformations could be added for other reasons, such as to aid debugging or for performance monitoring.

7.1 Finding

Transformations can be imported directly, but the user needs to know what transformations are available. A helper class **TransInfo** is provided to show the available transformations

---

Note: The directory layout of PSyclone is currently being restructured. As a result of this some transformations are already in the new locations, while others have not been moved yet. Transformations in the new locations can at the moment not be found using the **TransInfo** approach, and need to be imported directly from the path indicated in the documentation.

---

class psyclone.psyGen.**TransInfo**(*module=None*, *base_class=None*)

This class provides information about, and access, to the available transformations in this implementation of PSyclone. New transformations will be picked up automatically as long as they subclass the abstract Transformation class.

For example:

```
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> print(t.list)
There is 1 transformation available:
  1: SwapTrans, A test transformation
>>> # accessing a transformation by index
>>> trans = t.get_trans_num(1)
>>> # accessing a transformation by name
>>> trans = t.get_trans_name("SwapTrans")
```

get_trans_name(*name*)

return the transformation with this name (use list() first to see available transformations)

get_trans_num(*number*)

return the transformation with this number (use list() first to see available transformations)

property list

return a string with a human readable list of the available transformations

property num_trans
return the number of transformations available

7.2 Standard Functionality

Each transformation must provide at least two functions for the user: one for validation, i.e. to verify that a certain transformation can be applied, and one to actually apply the transformation. They are described in detail in the *overview of all transformations*, but the following general guidelines apply.

7.2.1 Validation

Each transformation provides a function `validate`. This function can be called by the user, and it will raise an exception if the transformation can not be applied (and otherwise will return nothing). Validation will always be called when a transformation is applied. The parameters for `validate` can change from transformation to transformation, but each `validate` function accepts a parameter `options`. This parameter is either `None`, or a dictionary of string keys, that will provide additional parameters to the validation process. For example, some validation functions allow part of the validation process to be disabled in order to allow the HPC expert to apply a transformation that they know to be safe, even if the more general validation process might reject it. Those parameters are documented for each transformation, and will show up as a parameter, e.g.: `options["node-type-check"]`. As a simple example:

```
# The validation might reject the application, but in this
# specific case it is safe to apply the transformation,
# so disable the node type check:
my_transform.validate(node, {"node-type-check": False})
```

7.2.2 Application

Each transformation provides a function `apply` which will apply the transformation. It will first validate the transform by calling the `validate` function. Each `apply` function takes the same `options` parameter as the `validate` function described above. Besides potentially modifying the validation process, optional parameters for the transformation are also provided this way. A simple example:

```
kctrans = Dynamo0p3KernelConstTrans()
kctrans.apply(kernel, {"element_order": 0, "quadrature": True})
```

The same `options` dictionary will be used when calling `validate`.

7.3 Available transformations

Some transformations are generic as the schedule structure is independent of the API, however it often makes sense to specialise these for a particular API by adding API-specific errors checks. Some transformations are API-specific. Currently these different types of transformation are indicated by their names.

The generic transformations currently available are listed in alphabetical order below (a number of these have specialisations which can be found in the API-specific sections).

Note: PSyclone currently only supports OpenCL and KernelImportsToArguments transformations for the GOcean 1.0 API, the OpenACC Data transformation is limited to the generic code transformation and the GOcean 1.0 API and

the OpenACC Kernels transformation is limited to the generic code transformation and the LFRic API.

Note: The directory layout of PSyclone is currently being restructured. As a result of this some transformations are already in the new locations, while others have not been moved yet.

class psyclone.psyir.transformations.**Abs2CodeTrans**

Provides a transformation from a PSyIR ABS Operator node to equivalent code in a PSyIR tree. Validity checks are also performed.

The transformation replaces

```
R = ABS(X)
```

with the following logic:

```
IF X < 0.0:
    R = X*-1.0
ELSE:
    R = X
```

apply(*node*, *options=None*)

Apply the ABS intrinsic conversion transformation to the specified node. This node must be an ABS UnaryOperation. The ABS UnaryOperation is converted to equivalent inline code. This is implemented as a PSyIR transform from:

```
R = ... ABS(X) ...
```

to:

```
tmp_abs = X
if tmp_abs < 0.0:
    res_abs = tmp_abs*-1.0
else:
    res_abs = tmp_abs
R = ... res_abs ...
```

where `X` could be an arbitrarily complex PSyIR expression and `...` could be arbitrary PSyIR code.

This transformation requires the operation node to be a descendent of an assignment and will raise an exception if this is not the case.

Parameters

- **node** (psyclone.psyir.nodes.UnaryOperation) – an ABS UnaryOperation node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Warning: This transformation assumes that the ABS Intrinsic acts on PSyIR Real scalar data and does not check that this is not the case. Once issue #658 is on master then this limitation can be fixed.

class psyclone.transformations.**ACCDataTrans**

Add an OpenACC data region around a list of nodes in the PSyIR. COPYIN, COPYOUT and COPY clauses are added as required.

For example:

```
>>> from psyclone.psyir.frontend import FortranReader
>>> psyir = FortranReader().psyir_from_source(NEMO_SOURCE_FILE)
>>>
>>> from psyclone.transformations import ACCDataTrans
>>> from psyclone.psyir.transformations import ACCKernelsTrans
>>> ktrans = ACCKernelsTrans()
>>> dtrans = ACCDataTrans()
>>>
>>> schedule = psyir.children[0]
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Add a kernels construct for execution on the device
>>> kernels = schedule.children[9]
>>> ktrans.apply(kernels)
>>>
>>> # Enclose the kernels in a data construct
>>> kernels = schedule.children[9]
>>> dtrans.apply(kernels)
```

apply(*node*, *options=None*)

Put the supplied node or list of nodes within an OpenACC data region.

Parameters

- **node** ((list of) psyclone.psyir.nodes.Node) – the PSyIR node(s) to enclose in the data region.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

---

class psyclone.transformations.**ACCEnterDataTrans**

Adds an OpenACC "enter data" directive to a Schedule. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import         ACCEnterDataTrans, ACCLoopTrans,␣
↪ACCParallelTrans
>>> dtrans = ACCEnterDataTrans()
>>> ltrans = ACCLoopTrans()
>>> ptrans = ACCParallelTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
```

(continues on next page)

(continued from previous page)

```
>>> # print(schedule.view())
>>>
>>> # Apply the OpenACC Loop transformation to *every* loop in the schedule
>>> for child in schedule.children[:]:
...     ltrans.apply(child)
>>>
>>> # Enclose all of these loops within a single OpenACC parallel region
>>> ptrans.apply(schedule)
>>>
>>> # Add an enter data directive
>>> dtrans.apply(schedule)
>>>
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*sched*, *options=None*)

Adds an OpenACC "enter data" directive to the invoke associated with the supplied Schedule. Any fields accessed by OpenACC kernels within this schedule will be added to this data region in order to ensure they remain on the target device.

Parameters

- **sched** (sub-class of `psyclone.psyir.nodes.Schedule`) – schedule to which to add an "enter data" directive.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

---

class psyclone.psyir.transformations.**ACCKernelsTrans**

Enclose a sub-set of nodes from a Schedule within an OpenACC kernels region (i.e. within "!$acc kernels" ... "!$acc end kernels" directives).

For example:

```
>>> from psyclone.psyir.frontend import FortranReader
>>> psyir = FortranReader().psyir_from_source(NEMO_SOURCE_FILE)
>>>
>>> from psyclone.psyir.transformations import ACCKernelsTrans
>>> ktrans = ACCKernelsTrans()
>>>
>>> schedule = psyir.children[0]
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>> kernels = schedule.children[9]
>>> # Transform the kernel
>>> ktrans.apply(kernels)
```

apply(*node*, *options=None*)

Enclose the supplied list of PSyIR nodes within an OpenACC Kernels region.

Parameters

- **node** (`psyclone.psyir.nodes.Node` | list[`psyclone.psyir.nodes.Node`]) – a node or list of nodes in the PSyIR to enclose.

- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["default_present"]** (*bool*) – whether or not the kernels region should have the 'default present' attribute (indicating that data is already on the accelerator). When using managed memory this option should be False.

---

class psyclone.transformations.**ACCLoopTrans**

Adds an OpenACC loop directive to a loop. This directive must be within the scope of some OpenACC Parallel region (at code-generation time).

For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.parse.utils import ParseError
>>> from psyclone.psyGen import PSyFactory
>>> from psyclone.errors import GenerationError
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('ACCLoopTrans')
>>> rtrans = t.get_trans_name('ACCParallelTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Apply the OpenACC Loop transformation to *every* loop in the schedule
>>> for child in schedule.children[:]:
...     ltrans.apply(child)
>>>
>>> # Enclose all of these loops within a single OpenACC parallel region
>>> rtrans.apply(schedule)
>>>
```

apply(*node*, *options=None*)

Apply the ACCLoop transformation to the specified node. This node must be a Loop since this transformation corresponds to inserting a directive immediately before a loop, e.g.:

```
!$ACC LOOP
do ...
   ...
end do
```

At code-generation time (when `psyclone.psyir.nodes.ACCLoopDirective.gen_code()` is called), this node must be within (i.e. a child of) a PARALLEL region.

Parameters

- **node** (`psyclone.psyir.nodes.Loop`) – the supplied node to which we will apply the Loop transformation.

- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["collapse"]** (*int*) – number of nested loops to collapse.
- **options["independent"]** (*bool*) – whether to add the "independent" clause to the directive (not strictly necessary within PARALLEL regions).
- **options["sequential"]** (*bool*) – whether to add the "seq" clause to the directive.
- **options["gang"]** (*bool*) – whether to add the "gang" clause to the directive.
- **options["vector"]** (*bool*) – whether to add the "vector" clause to the directive.

---

class psyclone.transformations.**ACCParallelTrans**(*default_present=True*)

Create an OpenACC parallel region by inserting an 'acc parallel' directive.

```
>>> from psyclone.psyGen import TransInfo
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.nodes import Loop
>>> psyir = FortranReader().psyir_from_source("""
... program do_loop
...     real, dimension(10) :: A
...     integer i
...     do i = 1, 10
...       A(i) = i
...     end do
... end program do_loop
... """)
>>> ptrans = TransInfo().get_trans_name('ACCParallelTrans')
>>>
>>> # Enclose the loop within a OpenACC PARALLEL region
>>> ptrans.apply(psyir.walk(Loop))
>>> print(FortranWriter()(psyir))
program do_loop
  real, dimension(10) :: a
  integer :: i

  !$acc parallel default(present)
  do i = 1, 10, 1
    a(i) = i
  enddo
  !$acc end parallel

end program do_loop
```

apply(*target_nodes*, *options=None*)

Encapsulate given nodes with the ACCParallelDirective.

Parameters

- **target_nodes** (psyclone.psyir.nodes.Node | List[psyclone.psyir.nodes.Node]) – a single Node or a list of Nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

- **options["node-type-check"]** (*bool*) – this flag controls if the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.
- **options["default_present"]** (*bool*) – this flag controls if the inserted directive should include the default_present clause.

---

class psyclone.psyir.transformations.**AllArrayAccess2LoopTrans**

Provides a transformation from a PSyIR Assignment containing constant index accesses to an array into single-trip loops. For example:

```
>>> from psyclone.psyir.transformations import AllArrayAccess2LoopTrans
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Assignment
>>> code = ("program example\n"
...         "  real a(10,10), b(10,10)\n"
...         "  integer :: n\n"
...         "  a(1,n-1) = b(1,n-1)\n"
...         "end program example\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> assignment = psyir.walk(Assignment)[0]
>>> AllArrayAccess2LoopTrans().apply(assignment)
>>> print(FortranWriter()(psyir))
program example
  real, dimension(10,10) :: a
  real, dimension(10,10) :: b
  integer :: n
  integer :: idx
  integer :: idx_1

  do idx = 1, 1, 1
    do idx_1 = n - 1, n - 1, 1
      a(idx,idx_1) = b(idx,idx_1)
    enddo
  enddo

end program example
```

apply(*node*, *options=None*)

Apply the AllArrayAccess2Loop transformation if the supplied node is an Assignment with an Array Reference on its left-hand-side. Each constant array index access (i.e. one not containing a loop iterator or a range) is then transformed into an iterator and the assignment placed within a single-trip loop, subject to any constraints in the ArrayAccess2Loop transformation.

If any of the AllArrayAccess2Loop constraints are not satisfied for a loop index then this transformation does nothing for that index and silently moves to the next.

Parameters

- **node** (psyclone.psyir.nodes.Assignment) – an assignment.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations. This is an optional argument that defaults to None.

---

class psyclone.psyir.transformations.**ArrayAccess2LoopTrans**

Provides a transformation to transform a constant index access to an array (i.e. one that does not contain a loop iterator) to a single trip loop. For example:

```
>>> from psyclone.psyir.transformations import ArrayAccess2LoopTrans
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Assignment
>>> code = ("program example\n"
...         "  real a(10)\n"
...         "  a(1) = 0.0\n"
...         "end program example\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> assignment = psyir.walk(Assignment)[0]
>>> ArrayAccess2LoopTrans().apply(assignment.lhs.children[0])
>>> print(FortranWriter()(psyir))
program example
  real, dimension(10) :: a
  integer :: ji

  do ji = 1, 1, 1
    a(ji) = 0.0
  enddo

end program example
```

apply(*node*, *options=None*)

Apply the ArrayAccess2Loop transformation if the supplied node is an access to an array index within an Array Reference that is on the left-hand-side of an Assignment node. The access must be a scalar (i.e. not a range) and must not include a loop variable (as we are transforming a single access to a loop).

If the constraints are satisfied then the array access is replaced with a loop iterator and a single trip loop. The new loop will be placed immediately around the assignment.

Parameters

- **node** (psyclone.psyir.nodes.Node) – an array index.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations. This is an optional argument that defaults to None.

---

class psyclone.psyir.transformations.**ArrayAssignment2LoopsTrans**

Provides a transformation from a PSyIR Array Range to a PSyIR Loop. For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Assignment
>>> from psyclone.psyir.transformations import ArrayAssignment2LoopsTrans
>>> code = """
... subroutine sub()
...   real :: tmp(10)
...   tmp(:) = tmp(:) + 3
... end subroutine sub"""
>>> psyir = FortranReader().psyir_from_source(code)
>>> assignment = psyir.walk(Assignment)[0]
```

(continues on next page)

(continued from previous page)

```
>>> trans = ArrayAssignment2LoopsTrans()
>>> trans.apply(assignment)
>>> print(psyir.debug_string())
subroutine sub()
  real, dimension(10) :: tmp
  integer :: idx

  do idx = LBOUND(tmp, dim=1), UBOUND(tmp, dim=1), 1
    tmp(idx) = tmp(idx) + 3
  enddo

end subroutine sub
```

By default the transformation will reject character arrays, though this can be overriden by setting the 'allow_string' option to True. Note that PSyclone expresses syntax such as *character(LEN=100)* as UnsupportedFortranType, and this transformation will convert unknown or unsupported types to loops.

apply(*node*, *options=None*)

Apply the transformation to the specified array assignment node. Each range node within the assignment is replaced with an explicit loop. The bounds of the loop are determined from the bounds of the array range on the left hand side of the assignment.

Parameters

- **node** (psyclone.psyir.nodes.Assignment) – an Assignment node.
- **options["allow_string"]** (*bool*) – whether to allow the transformation on a character type array range. Defaults to False.
- **options["verbose"]** (*bool*) – log the reason the validation failed, at the moment with a comment in the provided PSyIR node.

---

class psyclone.psyir.transformations.**ChunkLoopTrans**

Apply a chunking transformation to a loop (in order to permit a chunked parallelisation). For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.psyir.transformations import ChunkLoopTrans
>>> psyir = FortranReader().psyir_from_source("""
... subroutine sub()
...     integer :: ji, tmp(100)
...     do ji=1, 100
...         tmp(ji) = 2 * ji
...     enddo
... end subroutine sub""")
>>> loop = psyir.walk(Loop)[0]
>>> ChunkLoopTrans().apply(loop)
```

will generate:

```
subroutine sub()
    integer :: ji
    integer, dimension(100) :: tmp
    integer :: ji_el_inner
```

(continues on next page)

(continued from previous page)

```
    integer :: ji_out_var
    do ji_out_var = 1, 100, 32
        ji_el_inner = MIN(ji_out_var + (32 - 1), 100)
        do ji = ji_out_var, ji_el_inner, 1
            tmp(ji) = 2 * ji
        enddo
    enddo
end subroutine sub
```

apply(*node*, *options=None*)

Converts the given Loop node into a nested loop where the outer loop is over chunks and the inner loop is over each individual element of the chunk.

Parameters

- **node** (psyclone.psyir.nodes.Loop) – the loop to transform.
- **options** (*Optional[Dict[str, Any]]*) – a dict with options for transformations.
- **options["chunksize"]** (*int*) – The size to chunk over for this transformation. If not specified, the value 32 is used.

---

class psyclone.transformations.**ColourTrans**

Apply a colouring transformation to a loop (in order to permit a subsequent parallelisation over colours). For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> ctrans = ColourTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>>     ctrans.apply(child)
>>>
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*node*, *options=None*)

Converts the Loop represented by node into a nested loop where the outer loop is over colours and the inner loop is over cells of that colour.

Parameters

- **node** (psyclone.psyir.nodes.Loop) – the loop to transform.
- **options** (*Optional[Dict[str, Any]]*) – options for the transformation.

---

class psyclone.psyir.transformations.**DotProduct2CodeTrans**

Provides a transformation from a PSyIR DOT_PRODUCT Operator node to equivalent code in a PSyIR tree. Validity checks are also performed.

If R is a scalar and A, and B have dimension N, The transformation replaces:

```
R = ... DOT_PRODUCT(A,B) ...
```

with the following code:

```
TMP = 0.0
do I=1,N
    TMP = TMP + A(i)*B(i)
R = ... TMP ...
```

For example:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import IntrinsicCall
>>> from psyclone.psyir.transformations import DotProduct2CodeTrans
>>> code = ("subroutine dot_product_test(v1,v2)\n"
...         "real,intent(in) :: v1(10), v2(10)\n"
...         "real :: result\n"
...         "result = dot_product(v1,v2)\n"
...         "end subroutine\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> trans = DotProduct2CodeTrans()
>>> trans.apply(psyir.walk(IntrinsicCall)[0])
>>> print(FortranWriter()(psyir))
subroutine dot_product_test(v1, v2)
  real, dimension(10), intent(in) :: v1
  real, dimension(10), intent(in) :: v2
  real :: result
  integer :: i
  real :: res_dot_product

  res_dot_product = 0.0
  do i = 1, 10, 1
    res_dot_product = res_dot_product + v1(i) * v2(i)
  enddo
  result = res_dot_product

end subroutine dot_product_test
```

apply(*node*, *options=None*)

Apply the DOT_PRODUCT intrinsic conversion transformation to the specified node. This node must be a DOT_PRODUCT BinaryOperation. If the transformation is successful then an assignment which includes a DOT_PRODUCT BinaryOperation node is converted to equivalent inline code.

Parameters

- **node** (psyclone.psyir.nodes.BinaryOperation) – a DOT_PRODUCT BinaryOperation node.
- **options** (*dict of str:str or None*) – a dictionary with options for transformations.

---

class psyclone.psyir.transformations.extract_trans.**ExtractTrans**(*node_class=<class 'psyclone.psyir.nodes.extract_node.ExtractNode'>*)

This transformation inserts an ExtractNode or a node derived from ExtractNode into the PSyIR of a schedule. At code creation time this node will use the PSyData API to create code that can write the input and output

parameters to a file. The node might also create a stand-alone driver program that can read the created file and then execute the instrumented region. Examples are given in the derived classes DynamoExtractTrans and GOceanExtractTrans.

After applying the transformation the Nodes marked for extraction are children of the ExtractNode. Nodes to extract can be individual constructs within an Invoke (e.g. Loops containing a Kernel or BuiltIn call) or entire Invokes. This functionality does not support distributed memory.

Parameters **node_class** (`psyclone.psyir.nodes.ExtractNode` or derived class) – The Node class of which an instance will be inserted into the tree (defaults to ExtractNode), but can be any derived class.

apply(*nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single PSyData region.

Parameters

- **nodes** (`psyclone.psyir.nodes.Node` or list of `psyclone.psyir.nodes.Node`) – can be a single node or a list of nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (`PREFIX_psy_data_mod`) and the PSyDataType (`PREFIX_PSYDATATYPE`) - a "_" will be added automatically. It defaults to "".
- **options["region_name"]** (*(str,str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

---

class psyclone.psyir.transformations.**HoistLocalArraysTrans**

This transformation takes a Routine and promotes any local, 'automatic' arrays to Container scope:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Assignment
>>> from psyclone.psyir.transformations import HoistLocalArraysTrans
>>> code = ("module test_mod\n"
...         "contains\n"
...         "  subroutine test_sub(n)\n"
...         "  integer :: i,j,n\n"
...         "  real :: a(n,n)\n"
...         "  real :: value = 1.0\n"
...         "  do i=1,n\n"
...         "    do j=1,n\n"
...         "      a(i,j) = value\n"
...         "    end do\n"
...         "  end do\n"
...         "  end subroutine test_sub\n"
...         "end module test_mod\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> hoist = HoistLocalArraysTrans()
>>> hoist.apply(psyir.walk(Routine)[0])
```

(continues on next page)

(continued from previous page)

```
>>> print(FortranWriter()(psyir).lower())
module test_mod
  implicit none
  real, allocatable, dimension(:,:), private :: a
  public

  public :: test_sub

  contains
  subroutine test_sub(n)
    integer :: n
    integer :: i
    integer :: j
    real :: value = 1.0

    if (.not.allocated(a) .or. ubound(a, 1) /= n .or. ubound(a, 2) /= n) then
      if (allocated(a)) then
        deallocate(a)
      end if
      allocate(a(1 : n, 1 : n))
    end if
    do i = 1, n, 1
      do j = 1, n, 1
        a(i,j) = value
      enddo
    enddo

  end subroutine test_sub

end module test_mod
```

By default, the target routine will be rejected if it is found to contain an ACCRoutineDirective since this usually implies that the routine will be launched in parallel on the OpenACC device. This check can be disabled by setting 'allow_accroutine' to True in the *options* dictionary.

apply(*node*, *options=None*)

Applies the transformation to the supplied Routine node, moving any local arrays up to Container scope and adding a suitable allocation when they are first accessed. If there are no local arrays or the supplied Routine is a program then this method does nothing.

Parameters

- **node** (psyclone.psyir.nodes.Routine) – target PSyIR node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["allow_accroutine"]** (*bool*) – permit the target routine to contain an ACCRoutineDirective. These are forbidden by default because their presence usually indicates that the routine will be run in parallel on the OpenACC device.

---

class psyclone.psyir.transformations.**HoistLoopBoundExprTrans**

This transformation moves complex bounds expressions out of the loop construct and places them in integer scalar assignments before the loop.

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.psyir.transformations import HoistTrans
>>> code = ("program test\n"
...         "  use mymod, only: mytype\n"
...         "  integer :: i,j,n\n"
...         "  real :: a(n)\n"
...         "  do i=mytype%start, UBOUND(a,1)\n"
...         "    a(i) = 1.0\n"
...         "  end do\n"
...         "end program\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> hoist = HoistLoopBoundExprTrans()
>>> hoist.apply(psyir.walk(Loop)[0])
>>> print(FortranWriter()(psyir))
program test
  use mymod, only : mytype
  integer :: i
  integer :: j
  integer :: n
  real, dimension(n) :: a
  integer :: loop_bound
  integer :: loop_bound_1

  loop_bound_1 = UBOUND(a, 1)
  loop_bound = mytype%start
  do i = loop_bound, loop_bound_1, 1
    a(i) = 1.0
  enddo

end program test
```

apply(*node*, *options=None*)

Move complex bounds expressions out of the given loop construct and place them in integer scalar assignments before the loop.

Parameters

- **node** (psyclone.psyir.nodes.Loop) – target PSyIR loop.
- **options** (*Dict[str, Any]*) – a dictionary with options for transformations.

---

class psyclone.psyir.transformations.**HoistTrans**

This transformation takes an assignment and moves it outside of its parent loop if it is valid to do so. If as a result the loop body becomes empty, the loop will be removed altogether. For example:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Assignment
>>> from psyclone.psyir.transformations import HoistTrans
>>> code = ("program test\n"
...         "  integer :: i,j,n\n"
```

(continues on next page)

(continued from previous page)

```
...           "  real :: a(n,n)\n"
...           "  real value\n"
...           "  do i=1,n\n"
...           "    value = 1.0\n"
...           "    do j=1,n\n"
...           "      a(i,j) = value\n"
...           "    end do\n"
...           "  end do\n"
...           "end program\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> hoist = HoistTrans()
>>> hoist.apply(psyir.walk(Assignment)[0])
>>> print(FortranWriter()(psyir))
program test
  integer :: i
  integer :: j
  integer :: n
  real, dimension(n,n) :: a
  real :: value

  value = 1.0
  do i = 1, n, 1
    do j = 1, n, 1
      a(i,j) = value
    enddo
  enddo

end program test
```

apply(*node*, *options=None*)

Applies the hoist transformation to the supplied assignment node within a loop, moving the assignment outside of the loop if it is valid to do so. Issue #1445 will also look to extend this transformation to other types of node.

Parameters

- **node** (subclass of `psyclone.psyir.nodes.Assignment`) – target PSyIR node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

---

class psyclone.psyir.transformations.**InlineTrans**

This transformation takes a Call (which may have a return value) and replaces it with the body of the target routine. It is used as follows:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Call, Routine
>>> from psyclone.psyir.transformations import InlineTrans
>>> code = """
... module test_mod
... contains
...   subroutine run_it()
```

(continues on next page)

(continued from previous page)

```
...     integer :: i
...     real :: a(10)
...     do i=1,10
...       a(i) = 1.0
...       call sub(a(i))
...     end do
...   end subroutine run_it
...   subroutine sub(x)
...     real, intent(inout) :: x
...     x = 2.0*x
...   end subroutine sub
... end module test_mod"""
>>> psyir = FortranReader().psyir_from_source(code)
>>> call = psyir.walk(Call)[0]
>>> inline_trans = InlineTrans()
>>> inline_trans.apply(call)
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(psyir.walk(Routine)[0].view())
>>> print(FortranWriter()(psyir.walk(Routine)[0]))
subroutine run_it()
  integer :: i
  real, dimension(10) :: a

  do i = 1, 10, 1
    a(i) = 1.0
    a(i) = 2.0 * a(i)
  enddo

end subroutine run_it
```

Warning: Routines/calls with any of the following characteristics are not supported and will result in a TransformationError:

- the routine is not in the same file as the call;
- the routine contains an early Return statement;
- the routine contains a variable with UnknownInterface;
- the routine contains a variable with StaticInterface;
- the routine contains an UnsupportedType variable with ArgumentInterface;
- the routine has a named argument;
- the shape of any array arguments as declared inside the routine does not match the shape of the arrays being passed as arguments;
- the routine accesses an un-resolved symbol;
- the routine accesses a symbol declared in the Container to which it belongs.

Some of these restrictions will be lifted by #924.

apply(*node*, *options=None*)

Takes the body of the routine that is the target of the supplied call and replaces the call with it.

Parameters

- **node** (psyclone.psyir.nodes.Routine) – target PSyIR node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["force"]** (*bool*) – whether or not to permit the inlining of Routines containing CodeBlocks. Default is False.

---

class psyclone.domain.common.transformations.**KernelModuleInlineTrans**

Brings the routine being called into the same Container as the call site. For example:

```
from psyclone.domain.common.transformations import \
        KernelModuleInlineTrans

inline_trans = KernelModuleInlineTrans()
inline_trans.apply(schedule.walk(CodedKern)[0])

print(schedule.parent.view())
```

> **Warning:** Not all Routines can be moved. This transformation will reject attempts to move routines that access private data in the original Container.

apply(*node*, *options=None*)

Bring the kernel subroutine into this Container.

Parameters

- **node** (psyclone.psyGen.CodedKern) – the kernel to module-inline.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises

- **TransformationError** – if the called Routine cannot be brought into this Container because of a name clash with another Routine.
- **NotImplementedError** – if node is a Call (rather than a CodedKern) and the name of the called routine does not match that of the caller.

---

class psyclone.psyir.transformations.**LoopFuseTrans**

Provides a generic loop-fuse transformation to two Nodes in the PSyIR of a Schedule after performing validity checks for the supplied Nodes. Examples are given in the descriptions of any children classes.

If loops have different named loop variables, when possible the loop variable of the second loop will be renamed to be the same as the first loop. This has the side effect that the second loop's variable will no longer have its value modified, with the expectation that that value isn't used anymore.

Note that the validation of this transformation still has several shortcomings, especially for domain API loops. Use at your own risk.

apply(*node1*, *node2*, *options=None*)

Fuses two loops represented by *psyclone.psyir.nodes.Node* objects after performing validity checks.

If the two loops don't have the same loop variable, the second loop's variable (and references to it inside the loop) will be changed to be references to the first loop's variable before merging. This has the side effect that the second loop's variable will no longer have its value modified, with the expectation that that value isn't used after.

Parameters

- **node1** (psyclone.psyir.nodes.Node) – the first Node that is being checked.
- **node2** (psyclone.psyir.nodes.Node) – the second Node that is being checked.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

---

class psyclone.psyir.transformations.**LoopSwapTrans**

Provides a loop-swap transformation, e.g.:

```
DO j=1, m
    DO i=1, n
```

becomes:

```
DO i=1, n
    DO j=1, m
```

This transform is used as follows:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> ast, invokeInfo = parse("shallow_alg.f90")
>>> psy = PSyFactory("gocean").create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> from psyclone.transformations import LoopSwapTrans
>>> swap = LoopSwapTrans()
>>> swap.apply(schedule.children[0])
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*node*, *options=None*)

The argument `outer` must be a loop which has exactly one inner loop. This transform then swaps the outer and inner loop.

Parameters

- **outer** (psyclone.psyir.nodes.Loop) – the node representing the outer loop.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises **TransformationError** – if the supplied node does not allow a loop swap to be done.

---

class psyclone.psyir.transformations.**LoopTiling2DTrans**

Apply a 2D loop tiling transformation to a loop. For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.psyir.transformations import LoopTiling2DTrans
>>> psyir = FortranReader().psyir_from_source("""
... subroutine sub()
...     integer :: ji, tmp(100)
...     do i=1, 100
...       do j=1, 100
...         tmp(i, j) = 2 * tmp(i, j)
...       enddo
...     enddo
... end subroutine sub""")
>>> loop = psyir.walk(Loop)[0]
>>> LoopTiling2DTrans().apply(loop)
```

will generate:

```
subroutine sub()
    integer :: ji
    integer, dimension(100) :: tmp
    integer :: ji_el_inner
    integer :: ji_out_var
    do i_out_var = 1, 100, 32
      i_el_inner = MIN(i_out_var + (32 - 1), 100)
      do j_out_var = 1, 100, 32
        do i = i_out_var, i_el_inner, 1
          j_el_inner = MIN(j_out_var + (32 - 1), 100)
          do j = j_out_var, j_el_inner, 1
            tmp(i, j) = 2 * tmp(i, j)
          enddo
        enddo
      enddo
    enddo
end subroutine sub
```

apply(*node*, *options=None*)

Converts the given 2D Loop construct into a tiled version of the nested loops.

Parameters

- **node** (psyclone.psyir.nodes.Loop) – the loop to transform.
- **options** (*Optional[Dict[str, Any]]*) – a dict with options for transformations.
- **options["tilesize"]** (*int*) – The size of the resulting tile, currently square tiles are always used. If not specified, the value 32 is used.

---

class psyclone.psyir.transformations.**Matmul2CodeTrans**

Provides a transformation from a PSyIR MATMUL Operator node to equivalent code in a PSyIR tree. Validity checks are also performed.

For a matrix-vector multiplication, if the dimensions of R, A, and B are R(N), A(N,M), B(M), the transformation replaces:

```
R=MATMUL(A,B)
```

with the following code:

```
do i=1,N
    R(i) = 0.0
    do j=1,M
        R(i) = R(i) + A(i,j) * B(j)
```

For a matrix-matrix multiplication, if the dimensions of `R`, `A`, and `B` are `R(P,M)`, `A(P,N)`, `B(N,M)`, the MATMUL is replaced with the following code:

```
do j=1,M
    do i=1,P
        R(i,j) = 0.0
        do ii=1,N
            R(i,j) = R(i,j) + A(i,ii) * B(ii,j)
```

Note that this transformation does *not* support the case where `A` is a rank-1 array.

apply(*node*, *options=None*)

Apply the MATMUL intrinsic conversion transformation to the specified node. This node must be a MATMUL IntrinsicCall. The first argument must currently have two dimensions while the second must have either one or two dimensions. Each argument is permitted to have additional dimensions (i.e. more than 2) but in each case it is only the first one or two which may be ranges. Further, the ranges must currently be for the full index space for that dimension (i.e. array subsections are not supported). If the transformation is successful then an assignment which includes a MATMUL IntrinsicCall node is converted to equivalent inline code.

Parameters

- **node** (`psyclone.psyir.nodes.IntrinsicCall`) – a MATMUL IntrinsicCall node.
- **options** (*Optional[Dict[str, Any]]*) – options for the transformation.

Note: This transformation is currently limited to translating the matrix vector form of MATMUL to equivalent PSyIR code.

---

class psyclone.psyir.transformations.**Max2CodeTrans**

Provides a transformation from a PSyIR MAX Intrinsic node to equivalent code in a PSyIR tree. Validity checks are also performed (by a parent class).

The transformation replaces

```
R = MAX(A, B, C ...)
```

with the following logic:

```
R = A
if B > R:
    R = B
if C > R:
    R = C
...
```

apply(*node*, *options=None*)

Apply this utility transformation to the specified node. This node must be a MIN or MAX BinaryOperation or NaryOperation. The operation is converted to equivalent inline code. This is implemented as a PSyIR transform from:

```
R = ... [MIN or MAX](A, B, C ...) ...
```

to:

```
res = A
tmp = B
IF tmp [< or >] res:
    res = tmp
tmp = C
IF tmp [< or >] res:
    res = tmp
...
R = ... res ...
```

where `A`, `B`, `C` ... could be arbitrarily complex PSyIR expressions and the `...` before and after `[MIN or MAX](A, B, C ...)` can be arbitrary PSyIR code.

This transformation requires the operation node to be a descendent of an assignment and will raise an exception if this is not the case.

Parameters

- **node** (`psyclone.psyir.nodes.BinaryOperation` or `psyclone.psyir.nodes.NaryOperation`) – a MIN or MAX Binary- or Nary-Operation node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Warning: This transformation assumes that the MAX Intrinsic acts on PSyIR Real scalar data and does not check that this is not the case. Once issue #658 is on master then this limitation can be fixed.

---

class psyclone.psyir.transformations.**Maxval2LoopTrans**

Provides a transformation from a PSyIR MAXVAL IntrinsicCall node to an equivalent PSyIR loop structure that is suitable for running in parallel on CPUs and GPUs. Validity checks are also performed.

If MAXVAL contains a single positional argument which is an array, the maximum value of all of the elements in the array is returned in the the scalar R.

```
R = MAXVAL(ARRAY)
```

For example, if the array is two dimensional, the equivalent code for real data is:

```
R = -HUGE(R)
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    R = MAX(R, ARRAY(I,J))
```

If the mask argument is provided then the mask is used to determine whether the maxval is applied:

```
R = MAXVAL(ARRAY, mask=MOD(ARRAY, 2.0)==1)
```

If the array is two dimensional, the equivalent code for real data is:

```
R = -HUGE(R)
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    IF (MOD(ARRAY(I,J), 2.0)==1) THEN
      R = MAX(R, ARRAY(I,J))
```

The dimension argument is currently not supported and will result in a TransformationError exception being raised.

```
R = MAXVAL(ARRAY, dimension=2)
```

The array passed to MAXVAL may use any combination of array syntax, array notation, array sections and scalar bounds:

```
R = MAXVAL(ARRAY) ! array syntax
R = MAXVAL(ARRAY(:,:)) ! array notation
R = MAXVAL(ARRAY(1:10,lo:hi)) ! array sections
R = MAXVAL(ARRAY(1:10,:)) ! mix of array section and array notation
R = MAXVAL(ARRAY(1:10,2)) ! mix of array section and scalar bound
```

An example use of this transformation is given below:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.transformations import Maxval2LoopTrans
>>> code = ("subroutine maxval_test(array)\n"
...         "  real :: array(10,10)\n"
...         "  real :: result\n"
...         "  result = maxval(array)\n"
...         "end subroutine\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> sum_node = psyir.children[0].children[0].children[1]
>>> Maxval2LoopTrans().apply(sum_node)
>>> print(FortranWriter()(psyir))
subroutine maxval_test(array)
  real, dimension(10,10) :: array
  real :: result
  integer :: idx
  integer :: idx_1

  result = -HUGE(result)
  do idx = 1, 10, 1
    do idx_1 = 1, 10, 1
      result = MAX(result, array(idx_1,idx))
    enddo
  enddo

end subroutine maxval_test
```

apply(*node*, *options=None*)
Apply the array-reduction intrinsic conversion transformation to the specified node. This node must be one of these intrinsic operations which is converted to an equivalent loop structure.

Parameters

- **node** (psyclone.psyir.nodes.IntrinsicCall) – an array-reduction intrinsic.
- **options** (*Optional[Dict[str, Any]]*) – options for the transformation.

---

class psyclone.psyir.transformations.**Min2CodeTrans**
Provides a transformation from a PSyIR MIN Intrinsic node to equivalent code in a PSyIR tree. Validity checks are also performed (by a parent class).

The transformation replaces

```
R = MIN(A, B, C ...)
```

with the following logic:

```
R = A
if B < R:
    R = B
if C < R:
    R = C
...
```

apply(*node*, *options=None*)
Apply this utility transformation to the specified node. This node must be a MIN or MAX BinaryOperation or NaryOperation. The operation is converted to equivalent inline code. This is implemented as a PSyIR transform from:

```
R = ... [MIN or MAX](A, B, C ...) ...
```

to:

```
res = A
tmp = B
IF tmp [< or >] res:
    res = tmp
tmp = C
IF tmp [< or >] res:
    res = tmp
...
R = ... res ...
```

where `A`, `B`, `C` ... could be arbitrarily complex PSyIR expressions and the `...` before and after `[MIN or MAX](A, B, C ...)` can be arbitrary PSyIR code.

This transformation requires the operation node to be a descendent of an assignment and will raise an exception if this is not the case.

Parameters

- **node** (psyclone.psyir.nodes.BinaryOperation or psyclone.psyir.nodes.NaryOperation) – a MIN or MAX Binary- or Nary-Operation node.

- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Warning: This transformation assumes that the MIN Intrinsic acts on PSyIR Real scalar data and does not check that this is not the case. Once issue #658 is on master then this limitation can be fixed.

---

class psyclone.psyir.transformations.**Minval2LoopTrans**

Provides a transformation from a PSyIR MINVAL IntrinsicCall node to an equivalent PSyIR loop structure that is suitable for running in parallel on CPUs and GPUs. Validity checks are also performed.

If MINVAL contains a single positional argument which is an array, the minimum value of all of the elements in the array is returned in the the scalar R.

```
R = MINVAL(ARRAY)
```

For example, if the array is two dimensional, the equivalent code for real data is:

```
R = HUGE(R)
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    R = MIN(R, ARRAY(I,J))
```

If the mask argument is provided then the mask is used to determine whether the minval is applied:

```
R = MINVAL(ARRAY, mask=MOD(ARRAY, 2.0)==1)
```

If the array is two dimensional, the equivalent code for real data is:

```
R = HUGE(R)
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    IF (MOD(ARRAY(I,J), 2.0)==1) THEN
      R = MIN(R, ARRAY(I,J))
```

The dimension argument is currently not supported and will result in a TransformationError exception being raised.

```
R = MINVAL(ARRAY, dimension=2)
```

The array passed to MINVAL may use any combination of array syntax, array notation, array sections and scalar bounds:

```
R = MINVAL(ARRAY) ! array syntax
R = MINVAL(ARRAY(:,:)) ! array notation
R = MINVAL(ARRAY(1:10,lo:hi)) ! array sections
R = MINVAL(ARRAY(1:10,:)) ! mix of array section and array notation
R = MINVAL(ARRAY(1:10,2)) ! mix of array section and scalar bound
```

For example:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
```

(continues on next page)

(continued from previous page)

```
>>> from psyclone.psyir.transformations import Minval2LoopTrans
>>> code = ("subroutine minval_test(array)\n"
...         "  real :: array(10,10)\n"
...         "  real :: result\n"
...         "  result = minval(array)\n"
...         "end subroutine\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> sum_node = psyir.children[0].children[0].children[1]
>>> Minval2LoopTrans().apply(sum_node)
>>> print(FortranWriter()(psyir))
subroutine minval_test(array)
  real, dimension(10,10) :: array
  real :: result
  integer :: idx
  integer :: idx_1

  result = HUGE(result)
  do idx = 1, 10, 1
    do idx_1 = 1, 10, 1
      result = MIN(result, array(idx_1,idx))
    enddo
  enddo

end subroutine minval_test
```

apply(*node*, *options=None*)

Apply the array-reduction intrinsic conversion transformation to the specified node. This node must be one of these intrinsic operations which is converted to an equivalent loop structure.

Parameters

- **node** (psyclone.psyir.nodes.IntrinsicCall) – an array-reduction intrinsic.
- **options** (*Optional[Dict[str, Any]]*) – options for the transformation.

---

class psyclone.transformations.**MoveTrans**

Provides a transformation to move a node in the tree. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> ast,invokeInfo=parse("dynamo.F90")
>>> psy=PSyFactory("lfric").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> from psyclone.transformations import MoveTrans
>>> trans=MoveTrans()
>>> trans.apply(schedule.children[0], schedule.children[2],
...             options = {"position":"after")
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

Nodes may only be moved to a new location with the same parent and must not break any dependencies otherwise an exception is raised.

apply(*node*, *location*, *options=None*)

Move the node represented by `node` before location `location` (which is also a node) by default and after if the optional *position* argument is set to 'after'.

Parameters

- **node** (`psyclone.psyir.nodes.Node`) – the node to be moved.
- **location** (`psyclone.psyir.nodes.Node`) – node before or after which the given node should be moved.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["position"]** (*str*) – either 'before' or 'after'.

Raises

- **TransformationError** – if the given node is not an instance of `psyclone.psyir.nodes.Node`
- **TransformationError** – if the location is not valid.

---

class psyclone.domain.gocean.transformations.**GOOpenCLTrans**

Switches on/off the generation of an OpenCL PSy layer for a given InvokeSchedule. Additionally, it will generate OpenCL kernels for each of the kernels referenced by the Invoke. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> API = "gocean"
>>> FILENAME = "shallow_alg.f90" # examples/gocean/eg1
>>> ast, invoke_info = parse(FILENAME, api=API)
>>> psy = PSyFactory(API, distributed_memory=False).create(invoke_info)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> ocl_trans = GOOpenCLTrans()
>>> ocl_trans.apply(schedule)
>>> print(schedule.view())
```

apply(*node*, *options=None*)

Apply the OpenCL transformation to the supplied GOInvokeSchedule. This causes PSyclone to generate an OpenCL version of the corresponding PSy-layer routine. The generated code makes use of the FortCL library (https://github.com/stfc/FortCL) in order to manage the OpenCL device directly from Fortran.

Parameters

- **node** (`psyclone.psyGen.GOInvokeSchedule`) – the InvokeSchedule to transform.
- **options** (*dict of str:value or None*) – set of option to tune the OpenCL generation.
- **options["enable_profiling"]** (*bool*) – whether or not to set up the OpenCL environment with the profiling option enabled.
- **options["out_of_order"]** (*bool*) – whether or not to set up the OpenCL environment with the out_of_order option enabled.
- **options["end_barrier"]** (*bool*) – whether or not to add an OpenCL barrier at the end of the transformed invoke.

class psyclone.transformations.**OMPDeclareTargetTrans**

Adds an OpenMP declare target directive to the specified routine.

For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.transformations import OMPDeclareTargetTrans
>>>
>>> tree = FortranReader().psyir_from_source("""
...     subroutine my_subroutine(A)
...         integer, dimension(10, 10), intent(inout) :: A
...         integer :: i
...         integer :: j
...         do i = 1, 10
...             do j = 1, 10
...                 A(i, j) = 0
...             end do
...         end do
...     end subroutine
...     """
>>> omptargettrans = OMPDeclareTargetTrans()
>>> omptargettrans.apply(tree.walk(Routine)[0])
```

will generate:

```
subroutine my_subroutine(A)
    integer, dimension(10, 10), intent(inout) :: A
    integer :: i
    integer :: j
    !$omp declare target
    do i = 1, 10
        do j = 1, 10
            A(i, j) = 0
        end do
    end do
end subroutine
```

apply(*node*, *options=None*)

Insert an OMPDeclareTargetDirective inside the provided routine or associated PSyKAl kernel.

Parameters

- **node** (psyclone.psyir.nodes.Routine | psyclone.psyGen.Kern) – the kernel or routine which is the target of this transformation.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["force"]** (*bool*) – whether to allow routines with CodeBlocks to run on the GPU.

class psyclone.psyir.transformations.**OMPLoopTrans**(*omp_directive='do'*, *omp_schedule='auto'*)

Adds an OpenMP directive to parallelise this loop. It can insert different directives such as "omp do/for", "omp parallel do/for", "omp teams distribute parallel do/for" or "omp loop" depending on the provided parameters. The OpenMP schedule to use can also be specified, but this will be ignored in case of the "omp loop" (as the 'schedule' clause is not valid for this specific directive). The configuration-defined 'reprod' parameter also specifies whether a manual reproducible reproduction is to be used. Note, reproducible in this case means obtaining the same results with the same number of OpenMP threads, not for different numbers of OpenMP threads.

Parameters

- **omp_schedule** (*str*) – the OpenMP schedule to use. Defaults to 'auto'.
- **omp_directive** (*str*) – choose which OpenMP loop directive to use. Defaults to "omp do"

For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.transformations import OMPLoopTrans, OMPParallelTrans
>>>
>>> psyir = FortranReader().psyir_from_source("""
...     subroutine my_subroutine()
...         integer, dimension(10, 10) :: A
...         integer :: i
...         integer :: j
...         do i = 1, 10
...             do j = 1, 10
...                 A(i, j) = 0
...             end do
...         end do
...     end subroutine
...     """)
>>> loop = psyir.walk(Loop)[0]
>>> omplooptrans1 = OMPLoopTrans(omp_schedule="dynamic",
...                              omp_directive="paralleldo")
>>> omplooptrans1.apply(loop)
>>> print(FortranWriter()(psyir))
subroutine my_subroutine()
  integer, dimension(10,10) :: a
  integer :: i
  integer :: j

  !$omp parallel do default(shared), private(i,j), schedule(dynamic)
  do i = 1, 10, 1
    do j = 1, 10, 1
      a(i,j) = 0
    enddo
  enddo
  !$omp end parallel do

end subroutine my_subroutine
```

apply(*node*, *options=None*)

Apply the OMPLoopTrans transformation to the specified PSyIR Loop.

Parameters

- **node** (psyclone.psyir.nodes.Node) – the supplied node to which we will apply the OMPLoopTrans transformation
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations and validation.
- **options["reprod"]** (*bool*) – indicating whether reproducible reductions should be used. By default the value from the config file will be used.

property omp_directive

Returns the type of OMP directive that this transformation will insert.

Return type str

property omp_schedule

Returns the OpenMP schedule that will be specified by this transformation.

Return type str

---

class psyclone.transformations.**OMPMasterTrans**

Create an OpenMP MASTER region by inserting directives. The most likely use case for this transformation is to wrap around task-based transformations. Note that adding this directive requires a parent OpenMP parallel region (which can be inserted by OMPParallelTrans), otherwise it will produce an error in generation-time.

For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import OMPParallelTrans, OMPMasterTrans
>>> mastertrans = OMPMasterTrans()
>>> paralleltrans = OMPParallelTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # MASTER region
>>> mastertrans.apply(schedule.children)
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> paralleltrans.apply(schedule.children)
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*target_nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Loops in the schedule within a single parallel region.

Parameters

- **target_nodes** ((list of) psyclone.psyir.nodes.Node) – a single Node or a list of Nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["node-type-check"]** (*bool*) – this flag controls if the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.

get_node_list(*nodes*)

This is a helper function for region based transformations. The parameter for any of those transformations is either a single node, a schedule, or a list of nodes. This function converts this into a list of nodes according to the parameter type. This function will always return a copy, to avoid issues e.g. if a child list of a node should be provided, and a transformation changes the order in this list (which would then also change the order of the nodes in the tree).

Parameters

- **nodes** (Union[psyclone.psyir.nodes.Node, psyclone.psyir.nodes.Schedule, List[psyclone.psyir.nodes.Node]) – can be a single node, a schedule or a list of nodes.
- **options** (*Optional[Dict[str,Any]]*) – a dictionary with options for transformations.

Returns a list of nodes.

Return type List[psyclone.psyir.nodes.Node]

Raises **TransformationError** – if the supplied parameter is neither a single Node, nor a Schedule, nor a list of Nodes.

validate(*node_list*, *options=None*)

Check that the supplied list of Nodes are eligible to be put inside a parallel region.

Parameters

- **node_list** (*list*) – list of nodes to put into a parallel region
- **options** – a dictionary with options for transformations. :type options: Optional[Dict[str, Any]]
- **options["node-type-check"]** (*bool*) – this flag controls whether or not the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.

Raises

- **TransformationError** – if the supplied node is an InvokeSchedule rather than being within an InvokeSchedule.
- **TransformationError** – if the supplied nodes are not all children of the same parent (siblings).

Note: PSyclone does not support (distributed-memory) halo swaps or global sums within OpenMP master regions. Attempting to create a master region for a set of nodes that includes halo swaps or global sums will produce an error. In such cases it may be possible to re-order the nodes in the Schedule such that the halo swaps or global sums are performed outside the single region. The *MoveTrans* transformation may be used for this.

---

class psyclone.transformations.**OMPParallelLoopTrans**(*omp_directive='do'*, *omp_schedule='auto'*)

Adds an OpenMP PARALLEL DO directive to a loop.

For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> ast, invokeInfo = parse("dynamo.F90")
>>> psy = PSyFactory("lfric").create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_v3_kernel_type').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> from psyclone.transformations import OMPParallelLoopTrans
>>> trans = OMPParallelLoopTrans()
>>> trans.apply(schedule.children[0])
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*node*, *options=None*)

Apply an OMPParallelLoop Transformation to the supplied node (which must be a Loop). In the generated code this corresponds to wrapping the Loop with directives:

```
!$OMP PARALLEL DO ...
do ...
  ...
end do
!$OMP END PARALLEL DO
```

Parameters

- **node** (psyclone.f2pygen.DoGen) – the node (loop) to which to apply the transformation.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations and validation.

---

class psyclone.transformations.**OMPParallelTrans**

Create an OpenMP PARALLEL region by inserting directives. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.parse.utils import ParseError
>>> from psyclone.psyGen import PSyFactory
>>> from psyclone.errors import GenerationError
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('GOceanOMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Apply the OpenMP Loop transformation to *every* loop
```

(continues on next page)

(continued from previous page)

```
>>> # in the schedule
>>> for child in schedule.children:
>>>     ltrans.apply(child)
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> rtrans.apply(schedule.children)
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*target_nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Loops in the schedule within a single parallel region.

Parameters

- **target_nodes** ((list of) psyclone.psyir.nodes.Node) – a single Node or a list of Nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["node-type-check"]** (*bool*) – this flag controls if the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.

get_node_list(*nodes*)

This is a helper function for region based transformations. The parameter for any of those transformations is either a single node, a schedule, or a list of nodes. This function converts this into a list of nodes according to the parameter type. This function will always return a copy, to avoid issues e.g. if a child list of a node should be provided, and a transformation changes the order in this list (which would then also change the order of the nodes in the tree).

Parameters

- **nodes** (Union[psyclone.psyir.nodes.Node, psyclone.psyir.nodes.Schedule, List[psyclone.psyir.nodes.Node]) – can be a single node, a schedule or a list of nodes.
- **options** (*Optional[Dict[str,Any]]*) – a dictionary with options for transformations.

Returns a list of nodes.

Return type List[psyclone.psyir.nodes.Node]

Raises **TransformationError** – if the supplied parameter is neither a single Node, nor a Schedule, nor a list of Nodes.

validate(*node_list*, *options=None*)

Perform OpenMP-specific validation checks.

Parameters

- **node_list** (list of psyclone.psyir.nodes.Node) – list of Nodes to put within parallel region.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["node-type-check"]** (*bool*) – this flag controls if the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.

Raises `TransformationError` – if the target Nodes are already within some OMP parallel region.

Note: PSyclone does not support (distributed-memory) halo swaps or global sums within OpenMP parallel regions. Attempting to create a parallel region for a set of nodes that includes halo swaps or global sums will produce an error. In such cases it may be possible to re-order the nodes in the Schedule such that the halo swaps or global sums are performed outside the parallel region. The *MoveTrans* transformation may be used for this.

---

class psyclone.transformations.**OMPSingleTrans**(*nowait=False*)

Create an OpenMP SINGLE region by inserting directives. The most likely use case for this transformation is to wrap around task-based transformations. The parent region for this should usually also be a OMPParallelTrans.

Parameters `nowait` (*bool*) – whether to apply a nowait clause to this transformation. The default value is False

For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import OMPParallelTrans, OMPSingleTrans
>>> singletrans = OMPSingleTrans()
>>> paralleltrans = OMPParallelTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # SINGLE region
>>> singletrans.apply(schedule.children)
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> paralleltrans.apply(schedule.children)
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*node_list*, *options=None*)

Apply the OMPSingleTrans transformation to the specified node in a Schedule.

At code-generation time this node must be within (i.e. a child of) an OpenMP PARALLEL region. Code generation happens when `OMPLoopDirective.gen_code()` is called, or when the PSyIR tree is given to a backend.

If the keyword "nowait" is specified in the options, it will cause a nowait clause to be added if it is set to True, otherwise no clause will be added.

Parameters

- **node_list** ((a list of) `psyclone.psyir.nodes.Node`) – the supplied node or node list to which we will apply the OMPSingleTrans transformation

- **options** (*Optional[Dict[str, Any]]*) – a list with options for transformations and validation.
- **options["nowait"]** (*bool*) – indicating whether or not to use a nowait clause on this single region.

get_node_list(*nodes*)

This is a helper function for region based transformations. The parameter for any of those transformations is either a single node, a schedule, or a list of nodes. This function converts this into a list of nodes according to the parameter type. This function will always return a copy, to avoid issues e.g. if a child list of a node should be provided, and a transformation changes the order in this list (which would then also change the order of the nodes in the tree).

Parameters

- **nodes** (Union[psyclone.psyir.nodes.Node, psyclone.psyir.nodes.Schedule, List[psyclone.psyir.nodes.Node]) – can be a single node, a schedule or a list of nodes.
- **options** (*Optional[Dict[str,Any]]*) – a dictionary with options for transformations.

Returns a list of nodes.

Return type List[psyclone.psyir.nodes.Node]

Raises **TransformationError** – if the supplied parameter is neither a single Node, nor a Schedule, nor a list of Nodes.

property omp_nowait

Returns whether or not this Single region uses a nowait clause to remove the end barrier.

Return type bool

validate(*node_list*, *options=None*)

Check that the supplied list of Nodes are eligible to be put inside a parallel region.

Parameters

- **node_list** (*list*) – list of nodes to put into a parallel region
- **options** – a dictionary with options for transformations. :type options: Optional[Dict[str, Any]]
- **options["node-type-check"]** (*bool*) – this flag controls whether or not the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.

Raises

- **TransformationError** – if the supplied node is an InvokeSchedule rather than being within an InvokeSchedule.
- **TransformationError** – if the supplied nodes are not all children of the same parent (siblings).

Note: PSyclone does not support (distributed-memory) halo swaps or global sums within OpenMP single regions. Attempting to create a single region for a set of nodes that includes halo swaps or global sums will produce an error. In such cases it may be possible to re-order the nodes in the Schedule such that the halo swaps or global sums are performed outside the single region. The *MoveTrans* transformation may be used for this.

---

class psyclone.psyir.transformations.**OMPTargetTrans**

Adds an OpenMP target directive to a region of code.

For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.psyir.transformations import OMPTargetTrans
>>>
>>> tree = FortranReader().psyir_from_source("""
...     subroutine my_subroutine()
...         integer, dimension(10, 10) :: A
...         integer :: i
...         integer :: j
...         do i = 1, 10
...             do j = 1, 10
...                 A(i, j) = 0
...             end do
...         end do
...     end subroutine
...     """)
>>> OMPTargetTrans().apply(tree.walk(Loop)[0])
>>> print(FortranWriter()(tree))
subroutine my_subroutine()
  integer, dimension(10,10) :: a
  integer :: i
  integer :: j

  !$omp target
  do i = 1, 10, 1
    do j = 1, 10, 1
      a(i,j) = 0
    enddo
  enddo
  !$omp end target

end subroutine my_subroutine
```

apply(*node*, *options=None*)

Insert an OMPTargetDirective before the provided node or list of nodes.

Parameters

- **node** (List[psyclone.psyir.nodes.Node]) – the PSyIR node or nodes to enclose in the OpenMP target region.
- **options** (*Optional[Dict[str,Any]]*) – a dictionary with options for transformations.

---

class psyclone.transformations.**OMPTaskloopTrans**(*grainsize=None*, *num_tasks=None*, *nogroup=False*)

Adds an OpenMP taskloop directive to a loop. Only one of grainsize or num_tasks must be specified.

TODO: #1364 Taskloops do not yet support reduction clauses.

Parameters

- **grainsize** (*int or None*) – the grainsize to use in for this transformation.

- **num_tasks** (*int or None*) – the num_tasks to use for this transformation.
- **nogroup** (*bool*) – whether or not to use a nogroup clause for this transformation. Default is False.

For example:

```
>>> from pysclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import OMPParallelTrans, OMPSingleTrans
>>> from psyclone.transformations import OMPTaskloopTrans
>>> from psyclone.psyir.transformations import OMPTaskwaitTrans
>>> singletrans = OMPSingleTrans()
>>> paralleltrans = OMPParallelTrans()
>>> tasklooptrans = OMPTaskloopTrans()
>>> taskwaittrans = OMPTaskwaitTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Apply the OpenMP Taskloop transformation to *every* loop
>>> # in the schedule.
>>> # This ignores loop dependencies. These can be handled
>>> # by the OMPTaskwaitTrans
>>> for child in schedule.children:
>>>     tasklooptrans.apply(child)
>>> # Enclose all of these loops within a single OpenMP
>>> # SINGLE region
>>> singletrans.apply(schedule.children)
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> paralleltrans.apply(schedule.children)
>>> # Ensure loop dependencies are satisfied
>>> taskwaittrans.apply(schedule.children)
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*node*, *options=None*)

Apply the OMPTaskloopTrans transformation to the specified node in a Schedule. This node must be a Loop since this transformation corresponds to wrapping the generated code with directives like so:

```
!$OMP TASKLOOP
do ...
   ...
end do
!$OMP END TASKLOOP
```

At code-generation time (when `OMPTaskloopDirective.gen_code()` is called), this node must be within (i.e. a child of) an OpenMP SERIAL region.

If the keyword "nogroup" is specified in the options, it will cause a nogroup clause be generated if it is set

to True. This will override the value supplied to the constructor, but will only apply to the apply call to which the value is supplied.

Parameters

- **node** (psyclone.psyir.nodes.Node) – the supplied node to which we will apply the OMPTaskloopTrans transformation
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations and validation.
- **options["nogroup"]** (*bool*) – indicating whether a nogroup clause should be applied to this taskloop.

property omp_grainsize

Returns the grainsize that will be specified by this transformation. By default the grainsize clause is not applied, so grainsize is None.

Returns The grainsize specified by this transformation.

Return type int or None

property omp_num_tasks

Returns the num_tasks that will be specified by this transformation. By default the num_tasks clause is not applied so num_tasks is None.

Returns The grainsize specified by this transformation.

Return type int or None

---

class psyclone.psyir.transformations.**OMPTaskTrans**

Apply an OpenMP Task Transformation to a Loop. The Loop must be within an OpenMP Serial region (Single or Master) at codegen time. Once lowering begins, no more modifications to the tree should occur as the task directives do not recompute dependencies after lowering. In the future it may be possible to do this through an _update_node implementation.

apply(*node*, *options=None*)

Apply the OMPTaskTrans to the specified node in a Schedule.

Can only be applied to a Loop.

The specified node is wrapped by directives during code generation like so:

```
!$OMP TASK
...
!$OMP END TASK
```

At code-generation time, this node must be within (i.e. a child of) an OpenMP Serial region (OpenMP Single or OpenMP Master)

Any kernels or Calls will be inlined into the region before the task transformation is applied.

Parameters

- **node** (psyclone.psyir.nodes.Loop) – the supplied node to which we will apply the OMPTaskTrans transformation
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations and validation.

---

class psyclone.psyir.transformations.**OMPTaskwaitTrans**

Adds zero or more OpenMP Taskwait directives to an OMP parallel region. This transformation will add directives to satisfy dependencies between Taskloop directives without an associated taskgroup (i.e. no nogroup clause). It also tries to minimise the number added to maximise available parallelism.

For example:

```
>>> from pysclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import OMPParallelTrans, OMPSingleTrans
>>> from psyclone.transformations import OMPTaskloopTrans
>>> from psyclone.psyir.transformations import OMPTaskwaitTrans
>>> singletrans = OMPSingleTrans()
>>> paralleltrans = OMPParallelTrans()
>>> tasklooptrans = OMPTaskloopTrans()
>>> taskwaittrans = OMPTaskwaitTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> print(schedule.view())
>>>
>>> # Apply the OpenMP Taskloop transformation to *every* loop
>>> # in the schedule.
>>> # This ignores loop dependencies. These are handled by the
>>> # taskwait transformation.
>>> for child in schedule.children:
>>>     taskloopstrans.apply(child, nogroup = true)
>>> # Enclose all of these loops within a single OpenMP
>>> # SINGLE region
>>> singletrans.apply(schedule.children)
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> paralleltrans.apply(schedule.children)
>>> taskwaittrans.apply(schedule.children)
>>> print(schedule.view())
```

apply(*node*, *options=None*)

Apply an OMPTaskwait Transformation to the supplied node (which must be an OMPParallelDirective). In the generated code this corresponds to adding zero or more OMPTaskwaitDirectives as appropriate:

```
!$OMP PARALLEL
  ...
  !$OMP TASKWAIT
  ...
  !$OMP TASKWAIT
  ...
!$OMP END PARALLEL
```

Parameters

- **node** (psyclone.psyir.nodes.OMPParallelDirective) – the node to which to apply

the transformation.

- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations and validation.
- **options["fail_on_no_taskloop"]** (*bool*) – indicating whether this should throw an error if no OMPTaskloop nodes are found in this tree. This can be safely disabled as if there are no Taskloop nodes the result of this transformation is valid OpenMP code. Default is True

---

class psyclone.psyir.transformations.**Product2LoopTrans**

Provides a transformation from a PSyIR PRODUCT IntrinsicCall node to an equivalent PSyIR loop structure that is suitable for running in parallel on CPUs and GPUs. Validity checks are also performed.

If PRODUCT contains a single positional argument which is an array, the maximum value of all of the elements in the array is returned in the the scalar R.

```
R = PRODUCT(ARRAY)
```

For example, if the array is two dimensional, the equivalent code for real data is:

```
R = 1.0
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    R = R * ARRAY(I,J)
```

If the mask argument is provided then the mask is used to determine whether the product is applied:

```
R = PRODUCT(ARRAY, mask=MOD(ARRAY, 2.0)==1)
```

If the array is two dimensional, the equivalent code for real data is:

```
R = 1.0
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    IF (MOD(ARRAY(I,J), 2.0)==1) THEN
      R = R * ARRAY(I,J)
```

The dimension argument is currently not supported and will result in a TransformationError exception being raised.

```
R = PRODUCT(ARRAY, dimension=2)
```

The array passed to PRODUCT may use any combination of array syntax, array notation, array sections and scalar bounds:

```
R = PRODUCT(ARRAY) ! array syntax
R = PRODUCT(ARRAY(:,:)) ! array notation
R = PRODUCT(ARRAY(1:10,lo:hi)) ! array sections
R = PRODUCT(ARRAY(1:10,:)) ! mix of array section and array notation
R = PRODUCT(ARRAY(1:10,2)) ! mix of array section and scalar bound
```

An example use of this transformation is given below:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.transformations import Product2LoopTrans
>>> code = ("subroutine product_test(array)\n"
...         "  real :: array(10,10)\n"
...         "  real :: result\n"
...         "  result = product(array)\n"
...         "end subroutine\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> product_node = psyir.children[0].children[0].children[1]
>>> Product2LoopTrans().apply(product_node)
>>> print(FortranWriter()(psyir))
subroutine product_test(array)
  real, dimension(10,10) :: array
  real :: result
  integer :: idx
  integer :: idx_1

  result = 1.0
  do idx = 1, 10, 1
    do idx_1 = 1, 10, 1
      result = result * array(idx_1,idx)
    enddo
  enddo

end subroutine product_test
```

apply(*node*, *options=None*)

Apply the array-reduction intrinsic conversion transformation to the specified node. This node must be one of these intrinsic operations which is converted to an equivalent loop structure.

Parameters

- **node** (psyclone.psyir.nodes.IntrinsicCall) – an array-reduction intrinsic.
- **options** (*Optional[Dict[str, Any]]*) – options for the transformation.

---

class psyclone.psyir.transformations.**ProfileTrans**

Create a profile region around a list of statements. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.parse.utils import ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> from psyclone.psyir.transformations import ProfileTrans
>>> api = "gocean"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> p_trans = ProfileTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> print(schedule.view())
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Enclose all children within a single profile region
>>> p_trans.apply(schedule.children)
>>> print(schedule.view())
```

This implementation relies completely on the base class PSyDataTrans for the actual work, it only adjusts the name etc, and the list of valid nodes.

apply(*nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single PSyData region.

Parameters

- **nodes** (psyclone.psyir.nodes.Node or list of psyclone.psyir.nodes.Node) – can be a single node or a list of nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (PREFIX_psy_data_mod) and the PSyDataType (PREFIX_PSYDATATYPE) - a "_" will be added automatically. It defaults to "".
- **options["region_name"]** (*(str,str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

---

class psyclone.psyir.transformations.**ReadOnlyVerifyTrans**(*node_class=<class 'psyclone.psyir.nodes.read_only_verify_node.ReadOnlyVerifyNo*

This transformation inserts a ReadOnlyVerifyNode or a node derived from ReadOnlyVerifyNode into the PSyIR of a schedule. At code creation time this node will use the PSyData API to create code that will verify that read-only quantities are not modified.

After applying the transformation the Nodes marked for verification are children of the ReadOnlyVerifyNode. Nodes to verify can be individual constructs within an Invoke (e.g. Loops containing a Kernel or BuiltIn call) or entire Invokes.

Parameters **node_class** (psyclone.psyir.nodes.ReadOnlyVerifyNode or derived class) – The class of Node which will be inserted into the tree (defaults to ReadOnlyVerifyNode), but can be any derived class.

apply(*nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single PSyData region.

Parameters

- **nodes** (psyclone.psyir.nodes.Node or list of psyclone.psyir.nodes.Node) – can be a single node or a list of nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (PREFIX_psy_data_mod) and the PSyDataType (PREFIX_PSYDATATYPE) - a "_" will be added automatically. It defaults to "".

- **options["region_name"]** (*(str, str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

---

class psyclone.psyir.transformations.**Reference2ArrayRangeTrans**

Provides a transformation from PSyIR Array Notation (a reference to an Array) to a PSyIR Range. For example:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Reference
>>> from psyclone.psyir.transformations import TransformationError
>>> CODE = ("program example\n"
...         "real :: a(:)\n"
...         "a = 0.0\n"
...         "end program\n")
>>> trans = Reference2ArrayRangeTrans()
>>> psyir = FortranReader().psyir_from_source(CODE)
>>> for reference in psyir.walk(Reference):
...     try:
...         trans.apply(reference)
...     except TransformationError:
...         pass
>>> print(FortranWriter()(psyir))
program example
  real, dimension(:) :: a

  a(:) = 0.0

end program example
```

This transformation does not currently support arrays within structures, see issue #1858.

apply(*node*, *options=None*)

Apply the Reference2ArrayRangeTrans transformation to the specified node. The node must be a Reference to an array. The Reference is replaced by an ArrayReference with appropriate explicit range nodes (termed colon notation in Fortran).

Parameters

- **node** (psyclone.psyir.nodes.Reference) – a Reference node.
- **options** (*Optional[Dict[str, Any]]*) – a dict with options for transformations.

---

class psyclone.psyir.transformations.**ReplaceInductionVariablesTrans**

Move all supported induction variables out of the loop, and replace their usage inside the loop. For example:

```
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.nodes import Loop
>>> from psyclone.psyir.transformations import
↪ReplaceInductionVariablesTrans
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> psyir = FortranReader().psyir_from_source("""
```

(continues on next page)

(continued from previous page)

```
... subroutine sub()
...     integer :: i, im, ic, tmp(100)
...     do i=1, 100
...         im = i - 1
...         ic = 2
...         tmp(i) = ic * im
...     enddo
... end subroutine sub""")
>>> loop = psyir.walk(Loop)[0]
>>> ReplaceInductionVariablesTrans().apply(loop)
>>> print(FortranWriter()(psyir))
subroutine sub()
  integer :: i
  integer :: im
  integer :: ic
  integer, dimension(100) :: tmp

  do i = 1, 100, 1
    tmp(i) = 2 * (i - 1)
  enddo
  ic = 2
  im = i - 1 - 1

end subroutine sub
```

The replaced induction variables assignments are added after the loop, so these variables will have the correct value if they are used elsewhere.

The following restrictions apply for the assignment to an induction variable:

- the variable must be a scalar (i.e. no array access at all, not even a constant like a(3) or a%b(3)%c)
- none of variables on the right-hand side can be written in the loop body (the loop variable is written in the Loop statement, not in the body, so it can be used).
- Only intrinsic function calls are allowed on the RHS (since they are known to be elemental)
- the assigned variable must not be read before the assignment.
- the assigned variable cannot occur on the right-hand side (e.g. $k = k + 3$).
- there must be only one assignment to this induction variable.

apply(*node*, *options=None*)

Apply the ReplaceInductionVariablesTrans transformation to the specified node. The node must be a loop. In case of nested loops, the transformation might need to be applied several times, from the inner-most loop outwards.

Parameters **node** (`psyclone.psyir.nodes.Loop`) – a Loop node.

---

class psyclone.psyir.transformations.**Sign2CodeTrans**

Provides a transformation from a PSyIR SIGN intrinsic node to equivalent code in a PSyIR tree. Validity checks are also performed.

The transformation replaces

```
R = SIGN(A, B)
```

with the following logic:

```
R = ABS(A)
if B < 0.0:
    R = R*-1.0
```

i.e. the value of A with the sign of B

apply(*node*, *options=None*)

Apply the SIGN intrinsic conversion transformation to the specified node. This node must be a SIGN IntrinsicCall. The SIGN IntrinsicCall is converted to equivalent inline code. This is implemented as a PSyIR transform from:

```
R = ... SIGN(A, B) ...
```

to:

```
tmp_abs = A
if tmp_abs < 0.0:
    res_abs = tmp_abs*-1.0
else:
    res_abs = tmp_abs
res_sign = res_abs
tmp_sign = B
if tmp_sign < 0.0:
    res_sign = res_sign*-1.0
R = ... res_sign ...
```

where A and B could be arbitrarily complex PSyIR expressions, ... could be arbitrary PSyIR code and where ABS has been replaced with inline code by the NemoAbsTrans transformation.

This transformation requires the IntrinsicCall node to be a child of an assignment and will raise an exception if this is not the case.

Parameters

- **node** (psyclone.psyir.nodes.IntrinsicCall) – a SIGN IntrinsicCall node.
- **symbol_table** (*psyclone.psyir.symbols.SymbolTable*) – the symbol table.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Warning: This transformation assumes that the SIGN Intrinsic acts on PSyIR Real scalar data and does not check whether or not this is the case. Once issue #658 is on master then this limitation can be fixed.

---

class psyclone.psyir.transformations.**Sum2LoopTrans**

Provides a transformation from a PSyIR SUM IntrinsicCall node to an equivalent PSyIR loop structure that is suitable for running in parallel on CPUs and GPUs. Validity checks are also performed.

If SUM contains a single positional argument which is an array, all elements of that array are summed and the result returned in the scalar R.

```
R = SUM(ARRAY)
```

For example, if the array is two dimensional, the equivalent code for real data is:

```
R = 0.0
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    R = R + ARRAY(I,J)
```

If the mask argument is provided then the mask is used to determine whether the sum is applied:

```
R = SUM(ARRAY, mask=MOD(ARRAY, 2.0)==1)
```

If the array is two dimensional, the equivalent code for real data is:

```
R = 0.0
DO J=LBOUND(ARRAY,2),UBOUND(ARRAY,2)
  DO I=LBOUND(ARRAY,1),UBOUND(ARRAY,1)
    IF (MOD(ARRAY(I,J), 2.0)==1) THEN
      R = R + ARRAY(I,J)
```

The dimension argument is currently not supported and will result in a TransformationError exception being raised.

```
R = SUM(ARRAY, dimension=2)
```

The array passed to MAXVAL may use any combination of array syntax, array notation, array sections and scalar bounds:

```
R = SUM(ARRAY) ! array syntax
R = SUM(ARRAY(:,:)) ! array notation
R = SUM(ARRAY(1:10,lo:hi)) ! array sections
R = SUM(ARRAY(1:10,:)) ! mix of array section and array notation
R = SUM(ARRAY(1:10,2)) ! mix of array section and scalar bound
```

For example:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> from psyclone.psyir.transformations import Sum2LoopTrans
>>> code = ("subroutine sum_test(array,n,m)\n"
...         "  integer :: n, m\n"
...         "  real :: array(10,10)\n"
...         "  real :: result\n"
...         "  result = sum(array)\n"
...         "end subroutine\n")
>>> psyir = FortranReader().psyir_from_source(code)
>>> sum_node = psyir.children[0].children[0].children[1]
>>> Sum2LoopTrans().apply(sum_node)
>>> print(FortranWriter()(psyir))
subroutine sum_test(array, n, m)
  integer :: n
  integer :: m
  real, dimension(10,10) :: array
```

(continues on next page)

(continued from previous page)

```
  real :: result
  integer :: idx
  integer :: idx_1

  result = 0.0
  do idx = 1, 10, 1
    do idx_1 = 1, 10, 1
      result = result + array(idx_1,idx)
    enddo
  enddo

end subroutine sum_test
```

apply(*node*, *options=None*)

Apply the array-reduction intrinsic conversion transformation to the specified node. This node must be one of these intrinsic operations which is converted to an equivalent loop structure.

Parameters

- **node** (psyclone.psyir.nodes.IntrinsicCall) – an array-reduction intrinsic.
- **options** (*Optional[Dict[str, Any]]*) – options for the transformation.

7.4 Algorithm-layer

The gocean API supports the transformation of the algorithm layer. In the future the LFRic API will also support this. The ability to transformation the algorithm layer is new and at this time no relevant transformations have been developed.

7.5 Kernels

PSyclone supports the transformation of Kernels as well as PSy-layer code. However, the transformation of kernels to produce new kernels brings with it additional considerations, especially regarding the naming of the resulting kernels. PSyclone supports two use cases:

1. the HPC expert wishes to optimise the same kernel in different ways, depending on where/how it is called;
2. the HPC expert wishes to transform the kernel just once and have the new version used throughout the Algorithm file.

The second case is really an optimisation of the first for the case where the same set of transformations is applied to every instance of a given kernel.

Since PSyclone is run separately for each Algorithm in a given application, ensuring that there are no name clashes for kernels in the application as a whole requires that some state is maintained between PSyclone invocations. This is achieved by requiring that the same kernel output directory is used for every invocation of PSyclone when building a given application. However, this is under the control of the user and therefore it is possible to use the same output directory for a subset of algorithms that require the same kernel transformation and then a different directory for another subset requiring a different transformation. Of course, such use would require care when building and linking the application since the differently-optimised kernels would have the same names.

By default, transformed kernels are written to the current working directory. Alternatively, the user may specify the location to which to write the modified code via the `-okern` command-line flag.

In order to support the two use cases given above, PSyclone supports two different kernel-renaming schemes: "multiple" and "single" (specified via the `--kernel-renaming` command-line flag). In the default, "multiple" scheme, PSyclone ensures that each transformed kernel is given a unique name (with reference to the contents of the kernel output directory). In the "single" scheme, it is assumed that any given kernel that is transformed is always transformed in the same way (or left unchanged) and thus just one transformed version of it is created. This assumption is checked by examining the Fortran code for any pre-existing transformed version of that kernel. If another transformed version of that kernel exists and does not match that created by the current transformation then PSyclone will raise an exception.

7.5.1 Rules

Kernel code that is to be transformed is subject to certain restrictions. These rules are intended to make kernel transformations as robust as possible, in particular by limiting the amount of code that must be parsed by PSyclone (via fparser). The rules are as follows:

1) Any variable or procedure accessed by a kernel must either be explicitly declared or named in the `only` clause of a module `use` statement within the scope of the subroutine containing the kernel implementation. This means that:
    1) Kernel subroutines are forbidden from accessing data using COMMON blocks;
    2) Kernel subroutines are forbidden from calling procedures declared via the EXTERN statement;
    3) Kernel subroutines must not access data or procedures made available via their parent (containing) module.
2) The full Fortran source of a kernel must be available to PSyclone. This includes the source of any modules from which it accesses either routines or data. (However, kernel routines are permitted to make use of Fortran intrinsic routines.)

For instance, consider the following Fortran module containing the `bc_ssh_code` kernel:

```
module boundary_conditions_mod
  real :: forbidden_var

contains

  subroutine bc_ssh_code(ji, jj, istep, ssha)
    use kind_params_mod, only: go_wp
    use model_mod, only: rdt
    integer,                      intent(in)    :: ji, jj, istep
    real(go_wp), dimension(:,:), intent(inout) :: ssha
    real(go_wp) :: rtime

    rtime = real(istep, go_wp) * rdt
    ...
  end subroutine bc_ssh_code

end module boundary_conditions_mod
```

Since the kernel subroutine accesses data (the `rdt` variable) from the `model_mod` module, the source of that module must be available to PSyclone if a transformation is applied to this kernel. Should `rdt` not actually be defined in `model_mod` (i.e. `model_mod` itself imports it from another module) then the source containing its definition must also be available to PSyclone. Note that the rules forbid the `bc_ssh_code` kernel from accessing the `forbidden_var` variable that is available to it from the enclosing module scope.

Note: these rules *only* apply to kernels that are the target of PSyclone kernel transformations.

7.5.2 Available Kernel Transformations

The transformations listed below have to be applied specifically to a PSyclone kernel. There are a number of transformations not listed here that can be applied to either or both the PSy-layer and Kernel-layer PSyIR.

Note: Some of these transformations modify the PSyIR tree of both the InvokeSchedule where the transformed CodedKernel is located and its associated KernelSchedule.

---

class psyclone.transformations.**ACCRoutineTrans**

Transform a kernel or routine by adding a "!$acc routine" directive (causing it to be compiled for the OpenACC accelerator device). For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean"
>>> ast, invokeInfo = parse(GOCEAN_SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import ACCRoutineTrans
>>> rtrans = ACCRoutineTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>> kern = schedule.children[0].children[0].children[0]
>>> # Transform the kernel
>>> rtrans.apply(kern)
```

apply(*node*, *options=None*)

Add the '!$acc routine' OpenACC directive into the code of the supplied Kernel (in a PSyKAl API such as GOcean or LFRic) or directly in the supplied Routine.

Parameters

- **node** (psyclone.psyGen.Kern | psyclone.psyir.nodes.Routine) – the kernel call or routine implementation to transform.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["force"]** (*bool*) – whether to allow routines with CodeBlocks to run on the GPU.
- **options["parallelism"]** (*str*) – the level of parallelism that the target routine (or a callee) exposes. One of "seq" (the default), "vector", "worker" or "gang".

validate(*node*, *options=None*)

Perform checks that the supplied kernel or routine can be transformed.

Parameters

- **node** (psyclone.psyGen.Kern | psyclone.psyir.nodes.Routine) – the kernel or routine which is the target of this transformation.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["force"]** (*bool*) – whether to allow routines with CodeBlocks to run on the GPU.

Raises

- **TransformationError** – if the node is not a kernel or a routine.
- **TransformationError** – if the target is a built-in kernel.
- **TransformationError** – if it is a kernel but without an associated PSyIR.
- **TransformationError** – if any of the symbols in the kernel are accessed via a module use statement.
- **TransformationError** – if the kernel contains any calls to other routines.
- **TransformationError** – if the 'parallelism' option is supplied but is not a recognised level of parallelism.

---

class psyclone.psyir.transformations.**FoldConditionalReturnExpressionsTrans**

Provides a transformation that folds conditional expressions with only a return statement inside so that the Return statement is moved to the end of the Routine and therefore it can be safely removed. This simplifies the control flow of the kernel to facilitate other transformations like kernel fusions. For example, the following code:

```
subroutine test(i)
    if (i < 5) then
        return
    endif
    if (i > 10) then
        return
    endif
    ! CODE
end subroutine
```

will be transformed to:

```
subroutine test(i)
    if (.not.(i < 5)) then
        if (.not.(i > 10)) then
            ! CODE
        endif
    endif
end subroutine
```

apply(*node*, *options=None*)

Apply this transformation to the supplied node.

Parameters

- **node** (psyclone.psyir.nodes.Routine) – the node to transform.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

property name
Returns the name of this transformation as a string.

validate(*node*, *options=None*)
Ensure that it is valid to apply this transformation to the supplied node.

Parameters

- **node** (psyclone.psyir.nodes.Routine) – the node to validate.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises **TransformationError** – if the node is not a Routine.

---

class psyclone.transformations.**KernelImportsToArguments**
Transformation that removes any accesses of imported data from the supplied kernel and places them in the caller. The values/references are then passed by argument into the kernel.

apply(*node*, *options=None*)
Convert the imported variables used inside the kernel into arguments and modify the InvokeSchedule to pass the same imported variables to the kernel call.

Parameters

- **node** (psyclone.psyGen.CodedKern) – a kernel call.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Note: This transformation is only supported by the GOcean 1.0 API.

7.6 Applying

Transformations can be applied either interactively or through a script.

7.6.1 Interactive

To apply a transformation interactively we first parse and analyse the code. This allows us to generate a "vanilla" PSy layer. For example:

```
>>> from fparser.common.readfortran import FortranStringReader
>>> from psyclone.parse.algorithm import Parser
>>> from psyclone.psyGen import PSyFactory
>>> from fparser.two.parser import ParserFactory

>>> example_str = (
...     "program example\n"
...     "  use field_mod, only: field_type\n"
...     "  type(field_type) :: field\n"
...     "  call invoke(setval_c(field, 0.0))\n"
...     "end program example\n")
```

(continues on next page)

(continued from previous page)

```
>>> parser = ParserFactory().create(std="f2008")
>>> reader = FortranStringReader(example_str)
>>> ast = parser(reader)
>>> invoke_info = Parser().invoke_info(ast)

# This example uses the LFRic API
>>> api = "lfric"

# Create the PSy-layer object using the invokeInfo
>>> psy = PSyFactory(api, distributed_memory=False).create(invoke_info)

# Optionally generate the vanilla PSy layer fortran
>>> print(psy.gen)
  MODULE example_psy
    USE constants_mod, ONLY: r_def, i_def
    USE field_mod, ONLY: field_type, field_proxy_type
    IMPLICIT NONE
    CONTAINS
    SUBROUTINE invoke_0(field)
      TYPE(field_type), intent(in) :: field
      INTEGER(KIND=i_def) df
      INTEGER(KIND=i_def) loop0_start, loop0_stop
      TYPE(field_proxy_type) field_proxy
      INTEGER(KIND=i_def) undf_aspc1_field
      !
      ! Initialise field and/or operator proxies
      !
      field_proxy = field%get_proxy()
      !
      ! Initialise number of DoFs for aspc1_field
      !
      undf_aspc1_field = field_proxy%vspace%get_undf()
      !
      ! Set-up all of the loop bounds
      !
      loop0_start = 1
      loop0_stop = undf_aspc1_field
      !
      ! Call our kernels
      !
      DO df=loop0_start,loop0_stop
        field_proxy%data(df) = 0.0
      END DO
      !
    END SUBROUTINE invoke_0
  END MODULE example_psy
```

We then extract the particular schedule we are interested in. For example:

```
# List the various invokes that the PSy layer contains
>>> print(psy.invokes.names)
dict_keys(['invoke_0'])
```

(continues on next page)

(continued from previous page)

```
# Get the required invoke
>>> invoke = psy.invokes.get('invoke_0')

# Get the schedule associated with the required invoke
> schedule = invoke.schedule
> print(schedule.view())
InvokeSchedule[invoke='invoke_0', dm=True]
    0: Loop[type='dof', field_space='any_space_1', it_space='dof', upper_bound='ndofs']
        Literal[value:'NOT_INITIALISED', Scalar<INTEGER, UNDEFINED>]
        Literal[value:'NOT_INITIALISED', Scalar<INTEGER, UNDEFINED>]
        Literal[value:'1', Scalar<INTEGER, UNDEFINED>]
        Schedule[]
            0: BuiltIn setval_c(field,0.0)
```

Now we have the schedule we can create and apply a transformation to it to create a new schedule and then replace the original schedule with the new one. For example:

```
# Create an OpenMPParallelLoopTrans
> from psyclone.transformations import OMPParallelLoopTrans
> ol = OMPParallelLoopTrans()

# Apply it to the loop schedule of the selected invoke
> ol.apply(schedule.children[0])
> print(schedule.view())

# Generate the Fortran code for the new PSy layer
> print(psy.gen)
```

7.6.2 Script

PSyclone provides a Python script (**psyclone**) that can be used from the command line to generate PSy layer code and to modify algorithm layer code appropriately. By default this script will generate "vanilla" (unoptimised) PSy-layer and algorithm layer code. For example:

```
> psyclone algspec.f90
> psyclone -oalg alg.f90 -opsy psy.f90 -api lfric algspec.f90
```

The **psyclone** script has an optional **-s** flag which allows the user to specify a script file to modify the PSy layer as required. Script files may be specified without a path. For example:

```
> psyclone -s opt.py algspec.f90
```

In this case, the current directory is prepended to the Python search path **PYTHONPATH** which will then be used to try to find the script file. Thus, the search begins in the current directory and continues over any pre-existing directories in the search path, failing if the file cannot be found.

Alternatively, script files may be specified with a path. In this case the file must exist in the specified location. This location is then added to the Python search path **PYTHONPATH** as before. For example:

```
> psyclone -s ./opt.py algspec.f90
> psyclone -s ../scripts/opt.py algspec.f90
> psyclone -s /home/me/PSyclone/scripts/opt.py algspec.f90
```

PSyclone also provides the same functionality via a function (which is what the **psyclone** script calls internally).

A valid script file must contain a **trans** function which accepts a *PSyIR node* representing the root of the psy-layer code (as a FileConatainer):

```
>>> def trans(psyir):
...     # ...
```

It is up to the script how to modify the PSyIR representation of the code. The example below does the same thing as the example in the *Interactive* section.

```
>>> def trans(psyir):
...     from psyclone.transformations import OMPParallelLoopTrans
...     from psyclone.psyir.node import Routine
...     for subroutine in psyir.walk(Routine):
...         if subroutine.name == 'invoke_0_v3_kernel_type':
...             ol = OMPParallelLoopTrans()
...             ol.apply(subroutine.children[0])
```

In the gocean API (and in the future the lfric API) an optional **trans_alg** function may also be supplied. This function accepts **PSyIR** (representing the algorithm layer) as an argument and returns **PSyIR** i.e.:

```
>>> def trans_alg(psyir):
...     # ...
```

As with the *trans()* function it is up to the script what it does with the algorithm PSyIR. Note that the *trans_alg()* script is applied to the algorithm layer before the PSy-layer is generated so any changes applied to the algorithm layer will be reflected in the **PSy** object that is passed to the *trans()* function.

For example, if the *trans_alg()* function in the script merged two *invoke* calls into one then the **Alg** object passed to the *trans()* function of the script would only contain one schedule associated with the merged invoke.

Of course the script may apply as many transformations as is required for a particular algorithm and/or schedule and may apply transformations to all the schedules (i.e. invokes and/or kernels) contained within the PSy layer.

Examples of the use of transformation scripts can be found in many of the examples, such as examples/lfric/eg3 and examples/lfric/scripts. Please read the examples/lfric/README file first as it explains how to run the examples (and see also the examples/check_examples script).

An example of the use of a script making use of the *trans_alg()* function can be found in examples/gocean/eg7.

7.7 OpenMP

OpenMP is added to a code by using transformations. The OpenMP transformations currently supported allow the addition of:

- an **OpenMP Parallel** directive
- an **OpenMP Target** directive
- an **OpenMP Declare Target** directive
- an **OpenMP Do/For/Loop** directive

- an **OpenMP Single** directive
- an **OpenMP Master** directive
- an **OpenMP Taskloop** directive
- multiple **OpenMP Taskwait** directives; and
- an **OpenMP Parallel Do** directive.

The generic versions of these transformations (i.e. ones that theoretically work for all APIs) were given in the *Standard Functionality* section. The API-specific versions of these transformations are described in the API-specific sections of this document.

7.7.1 Reductions

PSyclone supports parallel scalar reductions. If a scalar reduction is specified in the Kernel metadata (see the API-specific sections for details) then PSyclone ensures the appropriate reduction is performed.

In the case of distributed memory, PSyclone will add **GlobalSum's** at the appropriate locations. As can be inferred by the name, only "summation" reductions are currently supported for distributed memory.

In the case of an OpenMP parallel loop the standard reduction support will be used by default. For example

```
!$omp parallel do, reduction(+:x)
!loop
!$omp end parallel do
```

OpenMP reductions do not guarantee to give bit reproducible results for different runs of the same problem even if the same problem is run using the same resources. The reason for this is that the order in which data is reduced is not mandated.

Therefore, an additional **reprod** option has been added to the **OpenMP Do** transformation. If the reprod option is set to "True" then the OpenMP reduction support is replaced with local per-thread reductions which are reduced serially after the loop has finished. This implementation guarantees to give bit-wise reproducible results for different runs of the same problem using the same resources, but will not bit-wise compare if the code is rerun with different numbers of OpenMP threads.

7.7.2 Restrictions

If two reductions are used within an OpenMP region and the same variable is used for both reductions then PSyclone will raise an exception. In this case the solution is to use a different variable for each reduction.

PSyclone does not support (distributed-memory) halo swaps or global sums within OpenMP parallel regions. Attempting to create a parallel region for a set of nodes that includes halo swaps or global sums will produce an error. In such cases it may be possible to re-order the nodes in the Schedule using the *MoveTrans* transformation.

7.7.3 OpenMP Tasking

PSyclone supports OpenMP Tasking, through the *OMPTaskloopTrans* and *OMPTaskwaitTrans* transformations. *OMPTaskloopTrans* transformations can be applied to loops, whilst the *OMPTaskwaitTrans* operator is applied to an OpenMP Parallel Region, and computes the dependencies caused by Taskloops, and adds OpenMP Taskwait statements to satisfy those dependencies. An example of using OpenMP tasking is available in *PSyclone/examples/nemo/eg1/openmp_taskloop_trans.py*.

7.8 OpenCL

OpenCL is added to a code by using the `GOOpenCLTrans` transformation (see the *Standard Functionality* Section above). Currently this transformation is only supported for the GOcean1.0 API and is applied to the whole InvokeSchedule of an Invoke. This transformation will add an OpenCL driver infrastructure to the PSy layer and generate an OpenCL kernel for each of the Invoke kernels. This means that all kernels in that Invoke will be executed on the OpenCL device. The PSy-layer OpenCL code generated by PSyclone is still Fortran and makes use of the FortCL library (https://github.com/stfc/FortCL) to access OpenCL functionality. It also relies upon the device acceleration support provided by the dl_esm_inf library (https://github.com/stfc/dl_esm_inf).

Note: The generated OpenCL kernels are written in a file called opencl_kernels_<index>.cl where the index keeps increasing if the file name already exist.

The `GOOpenCLTrans` transformation accepts an *options* argument with a map of optional parameters to tune the OpenCL host code in the PSy layer. These options will be attached to the transformed InvokeSchedule. The current available options are:

Option	Description	Default
end_barrier	Whether a synchronization barrier should be placed at the end of the Invoke.	True
enable_profiling	Enables the profiling of OpenCL Kernels.	False
out_of_order	Allows the OpenCL implementation to execute the enqueued kernels out-of-order.	False

Additionally, each individual kernel (inside the Invoke that is going to be transformed) also accepts a map of options which are provided by the *set_opencl_options()* method of the *Kern* object. This can affect both the driver layer and/or the OpenCL kernels. The current available options are:

Option	Description	Default
local_size	Number of work-items to group together in a work-group execution (kernel instances executed at the same time).	64
queue_number	The identifier of the OpenCL command_queue to which the kernel should be submitted. If the kernel has a dependency on another kernel submitted to a different command_queue a barrier will be added to guarantee the execution order.	1

Below is an example of a PSyclone script that uses a `GOOpenCLTrans` with multiple InvokeSchedule and kernel-specific optimization options.

```
def trans(psyir):
    '''
    Applies OpenCL to the given PSy-layer.

```

(continues on next page)

(continued from previous page)

```
    :param psyir: the PSyIR of the PSy-layer.
    :type psyir: :py:class:`psyclone.psyir.nodes.FileContainer`

    '''
    ocl_trans = GOOpenCLTrans()
    fold_trans = FoldConditionalReturnExpressionsTrans()
    move_boundaries_trans = GOMoveIterationBoundariesInsideKernelTrans()

    # Provide kernel-specific OpenCL optimization options
    for idx, kern in enumerate(psyir.kernels()):
        # Move the PSy-layer loop boundaries inside the kernel as a kernel
        # mask, this allows to iterate through the whole domain
        move_boundaries_trans.apply(kern)
        # Change the syntax to remove the return statements introduced by the
        # previous transformation
        fold_trans.apply(kern.get_kernel_schedule())
        # Specify the OpenCL queue and workgroup size of the kernel
        # In this case we dispatch each kernel in a different queue to check
        # that the output code has the necessary barriers to guarantee the
        # kernel execution order.
        kern.set_opencl_options({"queue_number": idx+1, 'local_size': 4})

    # Transform the Schedule
    for schedule in psyir.walk(InvokeSchedule):
        ocl_trans.apply(schedule, options={"end_barrier": True})
```

OpenCL delays the decision of which and where kernels will execute until run-time, therefore it is important to use the environment variables provided by FortCL and DL_ESM_INF to inform how things should execute. Specifically:

- `FORTCL_KERNELS_FILE`: Point to the file containing the kernels to execute, they can be compiled ahead-of-time or providing the source for JIT compilation. To link more than a single kernel, one must merge all the kernels generated by PSyclone in a single source file.
- `FORTCL_PLATFORM`: If the system has more than 1 OpenCL platform. This environment variable may be used to select which platform on which to execute the kernels.
- `DL_ESM_ALIGNMENT`: When using OpenCL <= 1.2 the local_size should be exactly divisible by the total size. If this is not the case some implementations fail silently. A way to solve this issue is to set the *DL_ESM_ALIGNMENT* variable to be equal to the local size.

Note: The OpenCL generation can be combined with distributed memory generation. In the case where there is more than one accelerator available on each node, the PSyclone configuration file parameter `OCL_DEVICES_PER_NODE` has to be set to the appropriate value and the number of MPI-ranks-per-node set by the *mpirun* command has to match this value accordingly.

For instance if there are 2 accelerators per nodes, *psyclone.cfg* should have `OCL_DEVICES_PER_NODE=2` and the program must be executed with `mpirun -n <total_ranks> -ppn 2 ./application` (Note: *-ppn* is an Intel MPI specific parameter, use equivalent configuration parameters for other MPI implementations.)

For example, an execution of a PSyclone generated OpenCL code using all the mentioned run-time configuration options could look something like:

```
FORTCL_PLATFORM=3 FORTCL_KERNELS_FILE=allkernels.cl DL_ESM_ALIGNMENT=64 \
mpirun -n 2 ./application.exe
```

7.9 OpenACC

PSyclone supports the generation of code targetting GPUs through the addition of OpenACC directives. This is achieved by a user applying various OpenACC transformations to the PSyIR before the final Fortran code is generated. The steps to parallelisation are very similar to those in OpenMP with the added complexity of managing the movement of data to and from the GPU device. For the latter task PSyclone provides the `ACCDataTrans` and `ACCEnterDataTrans` transformations, as described in the *Standard Functionality* Section above. These two transformations add statically- and dynamically-scoped data regions, respectively. The former manages what data is on the remote device for a specific section of code while the latter allows run-time control of data movement. This second option is essential for minimising data movement as, without it, PSyclone-generated code would move data to and from the device upon every entry/exit of an Invoke. The first option is mainly provided as an aid to incremental porting and/or debugging of an OpenACC application as it provides explicit control over what data is present on a device for a given (part of an) Invoke routine.

The PGI compiler provides an alternative approach to controlling data movement through its 'unified memory' option (`-ta=tesla:managed`). When this is enabled the compiler itself takes on the task of ensuring that data is copied to/from the GPU when required. (Note that this approach can struggle with Fortran code containing derived types however.)

As well as ensuring the correct data is copied to and from the remote device, OpenACC directives must also be added to a code in order to tell the compiler how it should be parallelised. PSyclone provides the `ACCKernelsTrans`, `ACCParallelTrans` and `ACCLoopTrans` transformations for this purpose. The simplest of these is `ACCKernelsTrans` (currently only supported for the generic code transformation and LFRic API) which encloses the code represented by a sub-tree of the PSyIR within an OpenACC `kernels` region. This essentially gives free-reign to the compiler to automatically parallelise any suitable loops within the specified region. An example of the use of `ACCDataTrans` and `ACCKernelsTrans` may be found in PSyclone/examples/nemo/eg3 and an example of `ACCKernelsTrans` may be found in PSyclone/examples/lfric/eg14.

However, as with any "automatic" approach, a more performant solution can almost always be obtained by providing the compiler with more explicit direction on how to parallelise the code. The `ACCParallelTrans` and `ACCLoopTrans` transformations allow the user to define thread-parallel regions and, within those, define which loops should be parallelised. For an example of their use please see PSyclone/examples/gocean/eg2 or PSyclone/examples/lfric/eg14.

In order for a given section of code to be executed on a GPU, any routines called from within that section must also have been compiled for the GPU. This then requires either that any such routines are in-lined or that the OpenACC `routine` directive be added to any such routines. This situation will occur routinely in those PSyclone APIs that use the PSyKAl separation of concerns since the user-supplied kernel routines are called from within PSyclone-generated loops in the PSy layer. PSyclone therefore provides the `ACCRoutineTrans` transformation which, given a Kernel node in the PSyIR, creates a new version of that kernel with the `routine` directive added. See either PSyclone/examples/gocean/eg2 or PSyclone/examples/lfric/eg14 for an example.

7.10 SIR

It is currently not possible for PSyclone to output SIR code without using a script. Three examples of such scripts are given in example 4 for the NEMO examples directory. The first *sir_trans.py* simply outputs SIR. This will raise an exception if used with the *tracer advection* example as the example contains array-index notation which is not supported by the SIR backend, but will generate code for the other examples. The second, *sir_trans_loop.py* includes transformations to hoist code out of a loop, translate array-index notation into explicit loops and translate a single access to an array dimension to a one-trip loop (to make the code suitable for the SIR backend). This works with the *tracer-advection* example. The third script *sir_trans_all.py* additionally replaces any intrinsics with equivalent code and can also be used with the *tracer-advection* example (and the *intrinsic_example.f90* example).

CHAPTER

EIGHT

INTRODUCTION TO PSYKAL

PSyKAl is a kernel-based software architecture proposed in the GungHo project to design Fortran-embedded domain-specific languages that provide a clear separations of concerns between the science code and the optimisation/parallelisation details of an application. The model distinguishes between three layers: the Algorithm layer, the Kernel layer and the Parallelisation System (PSy) layer; which together give the model its name.

The Algorithm layer is responsible for providing a high-level description of the algorithm that the scientist wants to run. This layer operates on full fields and includes calls to kernels and built-ins.

The Kernel layer contains the actual implementation of the kernels as functors. Each functor implements a method that operates on an individual section of the fields. Depending on the DSL, these can be a single element, a vertical column, or a set of vertical columns. The kernels also specify some metadata that allow the data dependencies between kernels to be determined. Built-ins are similar to kernels but are provided by the PSyclone infrastructure itself.

The PSy layer acts as a bridge between the algorithm and kernel layers. It contains the code that is responsible for performing concurrent execution of kernels, including the way in which the iteration space is traversed, while respecting kernel dependencies. This layer can be tuned for specific platforms such as multi-node, multi-core and GPGPUs architectures without affecting the user-supplied Algorithm and Kernel layers.

Rather than requiring that the PSy layer be written manually, PSyclone uses the provided Algorithm and Kernel implementations to generate an inital PSyIR for the PSy-layer, optionally with distributed-memory parallelism. This then can be programatically optimised by applying PSyclone transformations (e.g. kernel fusing, colouring, inlining, ...) to better fit the target architecture.

The rest of this section describes how to use the `psyclone` command to process PSyKAl DSLs and how to implement each layer, providing examples for each of them.

8.1 Usage

To use PSyclone to process a PSyKAl algorithm file, the `-api API_NAME` parameter must be provided. In addition, distributed memory can be switched on or off by using the `-dm`/`--dist_mem` or `-nodm`/`--no_dist_mem` flags. For PSyKAl DSLs, the optional transformation script provided by the `-s SCRIPT` parameter will be applied to the PSyIR of the PSy-layer (but the scripts can also contain instructions to transform the code of the kernels used within it).

For example, the following command will process an LFRic PSyKAl algorithm file, generate a PSy-layer containing distributed-memory parallelism and then transform it using the `additional_optimisations.py` script:

```
psyclone -api lfric -dm -s additional_optimisations.py algorithm.f90 \
   -oalg algorithm_output.f90 -opsy psy_layer_output.f90
```

To date, there are two PSyKAl DSL implementations: the *LFRic PSyKAl API*, a mixed finite-element DSL used to implement the next-generation UK Met Office atmospheric model dynamical core; and the *GOcean PSyKAl API*, a finite difference ocean model benchmark. For more details on these see the corresponding sections of this User Guide.

8.2 Algorithm layer

As mentioned in the Introduction, the Algorithm layer provides a high-level description of the algorithm that the scientist would like to run, in terms of invocations to Kernel and Built-in operations. It operates on full fields and therefore it is not allowed to call the individual kernel executors nor include any parallelisation calls or directives. Instead, the algorithm layer uses the `invoke` subroutine, which takes as arguments one or more kernel functor constructors (as a Fortran type constructor) and, optionally, a name argument. The kernel functors, in turn, take as argument the full fields they operate on and any other quantities specified in their metadata.

For example:

```
...
call invoke(kernel1(arg1,arg2), kernel2(arg1, 3), name="Example_Invoke")
...
```

A complete application can consist of many algorithm files, each of them containing as many `invoke()` calls as required. PSyclone is applied to each individual algorithm layer file and must therefore be run multiple times if multiple algorithm files exist in a project.

The algorithm developer is also able to reference more than one Kernel/Built-in within an invoke call (as indicated in the previous example). In fact this feature is encouraged for performance reasons. **As a general guideline the developer should aim to use as few invokes as possible, each with as many Kernel functors in them as is possible**. The reason for this is that it allows for greater freedom for optimisations in the PSy-layer as these are limited to the contents of individual invokes - PSyclone currently does not attempt to optimise the PSy layer over multiple invoke calls.

As well as generating the PSy-layer code, PSyclone modifies the Algorithm layer code, replacing `invoke` calls with calls to the generated PSy-layer subroutine(s) (plus the associated `use` statements) so that the algorithm code is compilable and linkable. For example, the invoke above is translated into something like the following:

```
...
use psy, only : invoke_example_invoke
...
call invoke_example_invoke(arg1, arg2, 3)
...
```

The `name` argument in the invoke call is optional. If supplied it must be a string literal. Labels are not case-sensitive and must be valid Fortran names (e.g. `name="compute(1)"` is invalid). The label is used to name the corresponding PSy-layer routine generated by PSyclone in order to make debugging and profiling outputs more readable. So, for the above example, the generated PSy-layer subroutine will be named "invoke_example_invoke", otherwise a numeric index is given (e.g. "invoke_0"). Each invoke label must be unique within an Algorithm source file.

8.2.1 Limitations

There are limitations in the Fortran expressions that can be used inside the invoke call kernel arguments. The current list of known restrictions on the form of kernel arguments within an invoke is:

- No arithmetic expressions (e.g. `kernel_type(a+b)` or `kernel_type(-a)`)
- No named (optional) arguments (e.g. `kernel_type(fn(my_arg=a))`)

If you encounter any other limitations (or have a burning desire to use one of the above forms) then please contact the PSyclone developers.

8.3 Kernel layer

In the PSyKAl model, the Kernel code operates on an individual element of a field (such as a column of cells). The reason for doing this is that it gives the PSy layer flexibility in choosing the iteration order and exploiting the spatial domain parallelisation. The Kernel layer is not allowed to include any calls or directives related to parallelisation and works on raw Fortran arrays (to allow the compiler to optimise the code). Since a Kernel is called over the spatial domain (by the PSy layer) it must take at least one field or operator as an argument.

Kernels are implemented as Fortran Functors. Functors are objects that can be treated as if they are functions. As such they have two main interfaces for calling and providing arguments to them: the object constructor (used by the Algorithm layer) and a method that executes the code of the functor (used by the PSy-layer).

PSyKal applications accept one or more modules providing kernels, each of which can contain one or more kernel functors. Each kernel functor provides a set of meta-data attributes and a method with its implementation.

In the example below the module `w3_solver_kernel_mod` contains one kernel named `w3_solver_kernel_type` and its individual element execution method in subroutine `w3_solver_code`.

The metadata is API-specific and describes the kernel iteration space and dependencies, so that PSyclone can generate correct PSy-layer code (including any necessary halo-exchanges if DM is enabled). The metadata is provided by the kernel developer, who must guarantee its correctness. The example below shows meta-data for the `LFRic` API:

```
module w3_solver_kernel_mod

...

  type, public, extends(kernel_type) :: w3_solver_kernel_type
    private
    type(arg_type) :: meta_args(4) = (/                 &
         arg_type(GH_FIELD,   GH_REAL, GH_WRITE, W3),   &
         arg_type(GH_FIELD,   GH_REAL, GH_READ,  W3),   &
         arg_type(GH_FIELD*3, GH_REAL, GH_READ,  Wchi), &
         arg_type(GH_SCALAR,  GH_REAL, GH_READ)         &
         /)
    type(func_type) :: meta_funcs(2) = (/               &
         func_type(W3,   GH_BASIS),                     &
         func_type(Wchi, GH_DIFF_BASIS)                 &
         /)
    integer :: gh_shape = GH_QUADRATURE_XYoZ
    integer :: operates_on = CELL_COLUMN
  contains
    procedure, nopass :: solver_w3_code
  end type

contains

  subroutine solver_w3_code(nlayers,                                    &
                            x, rhs,                                     &
                            chi_1, chi_2, chi_3, ascalar,               &
                            ndf_w3, undf_w3, map_w3, w3_basis,          &
                            ndf_wchi, undf_wchi, map_wchi, wchi_diff_basis,  &
                            nqp_h, nqp_v, wqp_h, wqp_v)
    ...
  end subroutine solver_w3_code
```

(continues on next page)

(continued from previous page)

```
end module w3_solver_kernel_mod
```

Note that the executor method can also be declared as a module procedure interface to provide alternative implementations (e.g. different precisions) of the kernel code. These are selected as appropriate by the Fortran compiler, depending on the precision of the fields being passed to them:

```
type, extends(kernel_type) :: kernel1
  ...
  type(...) :: meta_args(...) = (/ ... /)
  ...
  integer :: operates_on = ...
  ...
end type kernel1

interface ...
  module procedure ...
end interface
```

8.4 Built-ins

Built-ins are operations which can be specified within an invoke call in the algorithm layer but do not require an associated kernel implementation because they are provided by the infrastructure.

These are useful for commonly-used operations, as they reduce the amount of code that the PSyKAl project has to maintain. In addition, they offer potential performance advantages as their implementation can completely change for different architectures and the PSy layer is free to implement these operations in whatever way it chooses.

The list of supported Built-ins is API-specific and therefore these are described under the documentation of each API. In general, PSyclone will need to know the types of the arguments being passed to any Built-ins. Each API provides a Fortran file that contains the metadata for all Built-in operations supported for that API.

Note: When a particular Built-in is used, the name of this Built-in should not be used for anything else within the same scope. For example, it is not valid to make use of a Built-in called `setval_c` and for its parent subroutine to also be called `setval_c`. In this case PSyclone will raise an exception.

8.4.1 Example

In the following algorithm-layer example from the LFRic API, the invoke call includes a call to two Built-ins (`setval_c` and `X_divideby_Y`) and a user-supplied kernel that operates on cell columns (`matrix_vector_kernel_mm_type`). The `setval_c` Built-in sets all values in the field `Ax` to `1.0` and the `X_divideby_Y` Built-in divides values in the field `rhs` by their equivalent (per degree of freedom) values in the field `lumped_weight` (see *supported LFRic API Built-ins*). Notice that, unlike the kernel call, no `use` association is required for the Built-ins since they are provided as part of the environment.

```
module solver_mod
  ...
  use matrix_vector_mm_mod, only: matrix_vector_kernel_mm_type
  ...
```

(continues on next page)

(continued from previous page)

```
  subroutine jacobi_solver_algorithm(lhs, rhs, mm, mesh, n_iter)

    integer(kind=i_def), intent(in)    :: n_iter
    type(field_type),    intent(inout) :: lhs
    type(field_type),    intent(in)    :: rhs
    type(operator_type), intent(in)    :: mm
    type(mesh_type),     intent(in)    :: mesh
    type(field_type)                   :: Ax, lumped_weight

    ...

    ! Compute mass lump
    call invoke( name = "Jacobi_mass_lump",                              &
                 setval_c(Ax, 1.0_r_def),                                &
                 matrix_vector_kernel_mm_type(lumped_weight, Ax, mm), &
                 X_divideby_Y(lhs, rhs, lumped_weight) )

  end subroutine jacobi_solver_algorithm
  ...

end module solver_mod
```

8.5 PSy layer

In the PSyKAl model, the PSy layer is the bridge between the Algorithm full-field operations and the Kernel/Built-Ins individual element operations. As such, it is responsible for:

1. calling any Kernel and expanding any Buit-In so that they iterate over their specified interation space;
2. map the Kernel and Built-In arguments supplied by an Algorithm `invoke` call to the arguments required by a Built-in or Kernel method;
3. include any required distributed-memory operations such as halo swaps and reductions to guarantee the correctness of the code;
4. providing an entry point for the optimisation expert to provide additional optimisations for the target architecture.

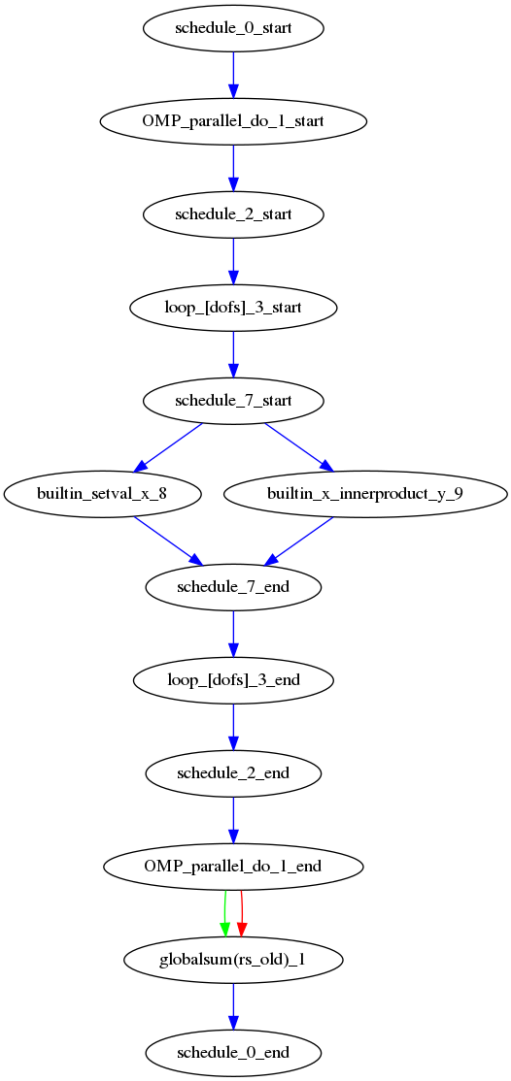
The PSy layer can be written manually but this is error prone and potentially complex to optimise. Therefore, the PSyclone code-generation system automatically generates an initial version of the PSy layer by parsing the associated Algorithm file and each of the kernels used in it. This initial version of the PSy-layer can be further tuned with the PSyclone code-transformation capabilities by providing a transformation script. Each API comes with a set of specialised transformations designed for that API.

Also, in addition to all the functionality available for PSyIR Nodes, the PSy-layer nodes have a *dag()* method (standing for directed acyclic graph), which outputs the PSyIR nodes and their data dependencies. By default a file in dot format is output with the name `dag` and a file in svg format is output with the name `dag.svg`. The file name can be changed using the `file_name` optional argument and the output file format can be changed using the `file_format` optional argument. The file_format value is simply passed on to graphviz so the graphviz documentation should be consulted for valid formats if svg is not required.

```
>>> schedule.dag(file_name="lovely", file_format="png")
```

Note: The dag method can be called from any node and will output the dag for that node and all of its children.

If we were to look at the LFRic eg6 example we would see the following image:



In the image, all PSyIR nodes with children are split into a start vertex and an end vertex (for example the InvokeSchedule node has both *schedule_start* and *schedule_end* vertices). Blue arrows indicate that there is a parent to child relationship (from a start node) or a child to parent relationship (to an end node). Green arrows indicate that a Node depends on another Node later in the schedule (which we call a forward dependence). Therefore the OMP parallel loop must complete before the globalsum is performed. Red arrows indicate that a Node depends on another Node that is earlier in the schedule (which we call a backward dependence). However the direction of the red arrows are reversed to improve the flow of the dag layout. In this example the forward and backward dependence is the same, however this is not always the case. The two built-ins do not depend on each other, so they have no associated green or red arrows.

The dependence graph output gives an indication of whether nodes can be moved within the InvokeSchedule. In this case it is valid to run the built-ins in either order. The underlying dependence analysis used to create this graph is used to determine whether a transformation of a Schedule is valid from the perspective of data dependencies.

CHAPTER

NINE

LFRIC API

This section describes the LFRic application programming interface (API). This API explains what a user needs to write in order to make use of the LFRic API in PSyclone.

As with the majority of PSyclone APIs, the LFRic API specifies how a user needs to write the algorithm layer and the kernel layer to allow PSyclone to generate the PSy layer. These algorithm and kernel APIs are discussed separately in the following sections.

The LFRic API supports the Met Office's finite element (hereafter FEM) based GungHo dynamical core (see Introduction). This dynamical core with atmospheric physics parameterisation schemes is a part of the Met Office LFRic modelling system [AFH+19], currently being developed in preparation for exascale computing in the 2020s. The LFRic repository and the associated wiki are hosted at the Met Office Science Repository Service. The code is BSD-licensed, however browsing the LFRic wiki and code repository requires login access to MOSRS. For more technical details on the implementation of LFRic, please see the LFRic documentation.

9.1 Algorithm

The general requirements for the structure of an Algorithm are explained in the *Algorithm layer* section. This section explains the LFRic-API-specific specialisations and extensions.

The LFRic API defines a set of objects, with specific meanings and data-structures, that can be provided as arguments to Kernels within invoke calls. These are: *scalar*, *field*, *field vector*, *operator*, *column-wise operator*, *Quadrature*, *Halo Depth* and *Stencil Extents*. The example below showcases the use of each of these arguments:

```
real(kind=r_def)           :: rscalar
integer(kind=i_def)        :: iscalar, halo_depth
logical(kind=l_def)        :: lscalar
integer(kind=i_def)        :: stencil_extent
type(field_type)           :: field1, field2, field3
type(field_type)           :: field5(3), field6(3)
type(integer_field_type)   :: field7
type(quadrature_type)      :: qr
type(operator_type)        :: operator1
type(columnwise_operator_type) :: cma_op1
...
call invoke( kernel1(field1, field2, operator1, qr),           &
             builtin1(rscalar, field2, field3),                &
             int_builtin2(iscalar, field7),                    &
             kernel2(field1, stencil_extent, field3, lscalar), &
             kernel3(field1, halo_depth),                      &
             assembly_kernel(cma_op1, operator1),              &
             name="some_calculation")
```

Each of these argument types is described in more detail in the next *section*.

The LFRic API has support for inter-grid kernels (those that map fields between grids of different resolution). At the Algorithm layer, an `invoke` of such kernels looks much like an `invoke` containing general-purpose kernels. The only restrictions to be aware of are that inter-grid kernels accept only field or field-vectors as arguments and that an `invoke` may not mix inter-grid kernels with any other kernel type.

9.2 Algorithm Argument Types

9.2.1 Scalar

In the LFRic API a scalar is a single-valued argument that is identified with `GH_SCALAR` metadata. Scalar arguments can have `real`, `integer` or `logical` data type in *user-defined Kernels* (`logical` data type is not supported in the *LFRic Built-ins*).

9.2.2 Field

LFRic API fields, identified with `GH_FIELD` metadata, represent FEM discretisations of various dynamical core prognostic and diagnostic variables. In FEM, variables are discretised by placing them into a function space (see *Supported Function Spaces*) from which they inherit a polynomial expansion via the basis functions of that space. Field values at points within a cell are evaluated as the sum of a set of basis functions multiplied by coefficients which are the data points. Points of evaluation are determined by a quadrature object (*Quadrature*) and are independent of the function space the field is on. Placement of field data points, also called degrees of freedom (hereafter "DoFs"), is determined by the function space the field is on. LFRic fields passed as arguments to any *LFRic kernel* can be of `real` or `integer` primitive type. In the LFRic infrastructure, these fields are represented by instances of the `field_type` and `integer_field_type` classes, respectively.

9.2.3 Field Vector

Depending on the function space a field lives on, the field data value at a point can be a scalar or a vector (see *Supported Function Spaces* for the list of scalar and vector function spaces). There is an additional option, called a *field vector*, to represent a bundle of either scalar- or vector-valued fields. Field vectors are represented as `GH_FIELD*N` where `N` is the size of the vector. The 3D coordinate field, for example, has `(x, y, z)` scalar values at the nodes and therefore has a vector size of 3.

9.2.4 Operator

Represents a matrix constructed on a per-cell basis using Local Matrix Assembly (LMA) and is identified with `GH_OPERATOR` metadata. In the LFRic infrastructure, operators are represented by instances of the `operator_type` class. LFRic operators can only have `real`-valued data in *user-defined Kernels* (*LFRic Built-ins* do not currently support operators).

9.2.5 Column-wise Operator

The LFRic API has support for the construction and use of column-wise/Column Matrix Assembly (CMA) operators whose metadata identifier is `GH_COLUMNWISE_OPERATOR`. In the LFRic infrastructure, column-wise operators are represented by instances of the `columnwise_operator_type` class. As for the LMA operators above, LFRic column-wise operators can only have `real`-valued *data*.

As the name suggests, these are operators constructed for a whole column of the mesh. These are themselves constructed from the Local Matrix Assembly (LMA) operators of each cell in the column. The rules governing Kernels that have CMA operators as arguments are given in the *Kernel* section below.

There are three recognised Kernel types involving CMA operations; construction, application (including inverse application) and matrix-matrix. The following example sketches-out what the use of such kernels might look like in the Algorithm layer:

```
use field_mod, only: field_type
use operator_mod, only : operator_type
use columnwise_operator_mod, only : columnwise_operator_type
type(field_type) :: field1, field2, field3
type(operator_type) :: lma_op1, lma_op2
type(columnwise_operator_type) :: cma_op1, cma_op2, cma_op3
real(kind=r_def) :: alpha
...
call invoke(                                                    &
        assembly_kernel(cma_op1, lma_op1, lma_op2),             &
        assembly_kernel2(cma_op2, lma_op1, lma_op2, field3),    &
        apply_kernel(field1, field2, cma_op1),                  &
        matrix_matrix_kernel(cma_op3, cma_op1, alpha, cma_op2), &
        apply_kernel(field3, field1, cma_op3),                  &
        name="cma_example")
```

The above invoke uses two LMA operators to construct the CMA operator `cma_op1`. A second CMA operator, `cma_op2`, is assembled from the same two LMA operators but also uses a field. The first of these CMA operators is then applied to `field2` and the result stored in `field1` (assuming that the metadata for `apply_kernel` specifies that it is the first field argument that is written to). The two CMA operators are then combined to produce a third, `cma_op3`. This is then applied to `field1` and the result stored in `field3`.

Note that PSyclone identifies the type of kernels performing column-wise operations based on their arguments as described in metadata (see *Rules for Kernels that work with CMA Operators* below). The names of the kernels in the above example are purely illustrative and are not used by PSyclone when determining kernel type.

A full example of CMA operator construction is available in `examples/lfric/eg7`.

9.2.6 Quadrature

Kernels conforming to the LFRic API may require quadrature information (specified using e.g. `gh_shape = gh_quadrature_XYoZ` in the kernel metadata - see Section *gh_shape and gh_evaluator_targets*). This information must be passed to the kernel from the Algorithm layer in the form of one or more `quadrature_type` objects. These must be the last arguments passed to the kernel (with the exception of `halo_depth` - *Halo Depth* - if the kernel requires it) and must be provided in the same order that they are specified in the kernel metadata, e.g. if the metadata for kernel `pressure_gradient_kernel_type` specified `gh_shape = gh_quadrature_XYoZ` and that for kernel `geopotential_gradient_kernel` had `gh_shape(2) = (\ gh_quadrature_XYoZ, gh_quadrature_face \)` then the corresponding invoke would look something like:

```
...
qr_xyoz = quadrature_xyoz_type(nqp_exact, rule)
qr_face = quadrature_face_type(nqp_exact, ..., rule)
call invoke(pressure_gradient_kernel_type(rhs_tmp(igh_u), rho, theta, qr_xyoz), &
            geopotential_gradient_kernel_type(rhs_tmp(igh_u), geopotential, &
                                              qr_xyoz, qr_face))
```

These quadrature objects specify the set(s) of points at which the basis/differential-basis functions required by the kernel are to be evaluated.

9.2.7 Halo Depth

If a Kernel is written to iterate into the halo (has an `OPERATES_ON` of `HALO_CELL_COLUMN` or `OWNED_AND_HALO_CELL_COLUMN`) then the halo depth must be passed as a final, `integer` argument to the Kernel.

9.2.8 Stencil Extent

The metadata for a Kernel which operates on a cell-column may specify that a Kernel performs a stencil operation on a field. Any such metadata must provide a stencil type. See the *meta_args* section for more details. The supported stencil types are `X1D`, `Y1D`, `XORY1D`, `CROSS`, `CROSS2D` or `REGION`.

If a stencil operation is specified by the Kernel metadata, the Algorithm layer must provide the `extent` of the stencil (the maximum distance from the central cell that the stencil extends). The LFRic API expects this information to be added as an additional `integer` argument immediately after the relevant field when specifying the Kernel via an `invoke`.

For example:

```
integer(kind=i_def) :: extent = 2
call invoke(kernel(field1, field2, extent))
```

where `field2` has kernel metadata specifying that it has a stencil access.

`extent` may also be passed as a literal. For example:

```
call invoke(kernel(field1, field2, 2))
```

where, again, `field2` has kernel metadata specifying that it has a stencil access.

Note: The stencil extent specified in the Algorithm layer is not the same as the stencil size passed in to the Kernel. The latter contains the number of cells in the stencil which is dependent on both the stencil type and extent.

If the Kernel metadata specifies that the stencil is of type `XORY1D` (which means `X1D` or `Y1D`) then the algorithm layer must specify whether the stencil is `X1D` or `Y1D` for that particular kernel call. The LFRic API expects this information to be added as an additional argument immediately after the relevant stencil extent argument. The argument should be an `integer` with valid values being `x_direction` or `y_direction`, both being supplied by the `LFRic` infrastructure via the `flux_direction_mod` fortran module

For example:

```
use flux_direction_mod, only : x_direction
integer(kind=i_def) :: direction = x_direction
integer(kind=i_def) :: extent = 2
! ...
call invoke(kernel(field1, field2, extent, direction))
```

`direction` may also be passed as a literal. For example:

```
use flux_direction_mod, only : x_direction
integer(kind=i_def) :: extent = 2
! ...
call invoke(kernel(field1, field2, extent, x_direction))
```

If the stencil is of type `CROSS2D` then the arrays passed to the kernel are of different dimensions to those of other stencils. The `CROSS2D` stencil is designed for use when it is necessary for a kernel to know where the stencil cells are, relative to the current cell. For this reason, the `stencil_size` passed to the kernel is an array of length 4 containing sizes for each branch of the stencil. The `stencil_size` array is always ordered: West, South, East, North. This branch dimension is also part of the `stencil_dofmap` array making it possible to loop over each branch of the stencil individually. The invoke call for the `CROSS2D` stencil remains of the same form as for other stencils.

If certain fields use the same value of extent and/or direction then the same variable, or literal value can be provided.

For example:

```
call invoke(kernel1(field1, field2, extent,  field3, extent, direction), &
            kernel2(field1, field2, extent2, field4, extent, direction))
```

In the above example `field2` and `field3` in `kernel1` and `field4` in `kernel2` will have the same `extent` value but `field2` in `kernel2` may have a different value. Similarly, `field3` in `kernel1` and `field4` in `kernel2` will have the same `direction` value.

An example of the use of stencils is available in `examples/lfric/eg5`.

There is currently no attempt to perform type checking in PSyclone so any errors in the type and/or position of arguments will not be picked up until compile time. However, PSyclone does check for the correct number of algorithm arguments. If the wrong number of arguments is provided then an exception is raised.

For example, running test 19.2 from the LFRic API test suite gives:

```
cd <PSYCLONEHOME>/src/psyclone/tests
psyclone test_files/dynamo0p3/19.2_single_stencil_broken.f90
"Generation Error: error: expected '5' arguments in the algorithm layer but found '4'.
Expected '4' standard arguments, '1' stencil arguments and '0' qr_arguments'"
```

9.3 Mixed Precision

The LFRic API supports the ability to specify the precision required by the model via precision variables. To make use of this, the code developer must declare scalars, fields and operators in the algorithm layer with the required LFRic-supported precision. In the current implementation there are two supported precisions for `REAL` data and one each for `INTEGER` and `LOGICAL` data. The actual precision used in the code can be set in a configuration file. For example, `INTEGER` data could be set to be 32-bit precision. As `REAL` data has more than one supported precision, different parts of the code can be configured to have different precision.

The table below gives the currently supported datatypes, their associated kernel metadata description and their precision:

Data Type	Kernel Metadata	Precision
REAL(R_DEF)	GH_SCALAR, GH_REAL	R_DEF
REAL(R_BL)	GH_SCALAR, GH_REAL	R_BL
REAL(R_PHYS)	GH_SCALAR, GH_REAL	R_PHYS
REAL(R_SOLVER)	GH_SCALAR, GH_REAL	R_SOLVER
REAL(R_TRAN)	GH_SCALAR, GH_REAL	R_TRAN
INTEGER(I_DEF)	GH_SCALAR, GH_INTEGER	I_DEF
LOGICAL(L_DEF)	GH_SCALAR, GH_LOGICAL	L_DEF
FIELD_TYPE	GH_FIELD, GH_REAL	R_DEF
R_BL_FIELD_TYPE	GH_FIELD, GH_REAL	R_BL
R_PHYS_FIELD_TYPE	GH_FIELD, GH_REAL	R_PHYS
R_SOLVER_FIELD_TYPE	GH_FIELD, GH_REAL	R_SOLVER
R_TRAN_FIELD_TYPE	GH_FIELD, GH_REAL	R_TRAN
INTEGER_FIELD_TYPE	GH_FIELD, GH_INTEGER	I_DEF
OPERATOR_TYPE	GH_OPERATOR, GH_REAL	R_DEF
R_SOLVER_OPERATOR_TYPE	GH_OPERATOR, GH_REAL	R_SOLVER
R_TRAN_OPERATOR_TYPE	GH_OPERATOR, GH_REAL	R_TRAN
COLUMNWISE_OPERATOR_TYPE	GH_COLUMNWISE_OPERATOR, GH_REAL	R_SOLVER

As can be seen from the above table, the kernel metadata does not capture all of the precision options. For example, from the metadata it is not possible to determine whether a `REAL` scalar, `REAL` field or `REAL` operator has precision `R_DEF`, `R_SOLVER` or `R_TRAN`.

If a scalar, field, or operator is specified with a particular precision in the algorithm layer then any associated kernels that it is passed to must have been written so that they support this precision. If a kernel needs to support data that can be stored with different precisions then appropriate precision-specific subroutines should be written. These precision-specific subroutine should be called via a generic interface (which lets Fortran choose the appropriate subroutine based on the precision of its argument(s)).

Below is a simple example of an algorithm code calling the same generic kernel twice with potentially different precision. The implementation of the generic kernel such that it supports both 32- and 64-bit precision is also shown. The use of LFRic names for precision in the algorithm code allows precision to be controlled in a simple way. For example, `r_solver` could be set to be 32-bits in one configuration and 64-bits in another:

```
program test

  use constants_mod,      only : r_def, r_solver
  use field_mod,          only : field_type
  use r_solver_field_mod, only : r_solver_field_type
  use example_mod,        only : example_type

  type(field_type)          :: field_r_def
  type(r_solver_field_type) :: field_r_solver
  real(kind=r_def)          :: x_r_def
  real(kind=r_solver)       :: x_r_solver

  call invoke( example_type(field_r_def, x_r_def), &
               example_type(field_r_solver, x_r_solver))

end program test

module example_mod
```

(continues on next page)

(continued from previous page)

```
  use argument_mod
  use kernel_mod

  implicit none

  type, extends(kernel_type) :: example_type
    type(arg_type), dimension(2) :: meta_args = (/       &
         arg_type(gh_field,  gh_real, gh_readwrite, w3), &
         arg_type(gh_scalar, gh_real, gh_read )          &
         /)
     integer :: operates_on = cell_column
   contains
     procedure, nopass :: code => example_code
  end type example_type

  private
  public :: example_code

  interface example_code
    module procedure example_code_32
    module procedure example_code_64
  end interface example_code

contains

  subroutine example_code_32(..., field1, x, ...)
    real*4, dimension(...), intent(inout) :: field1
    real*4, intent(in) :: x
    print *, "32-bit example called"
  end subroutine example_code_32

  subroutine example_code_64(..., field1, x, ...)
    real*8, dimension(...), intent(inout) :: field1
    real*8, intent(in) :: x
    print *, "64-bit example called"
  end subroutine example_code_64

end module example_mod
```

In order to support mixed precision, PSyclone needs to know the precision (as specified in the algorithm layer) of any kernel arguments that are of a type that supports different precisions (e.g. `GH_FIELD`). The reason for this is that PSyclone needs to be able to declare data with the correct precision information within the PSy-layer to ensure that the correct flavour of kernels are called.

PSyclone must therefore determine this information from the algorithm layer. The rules for whether PSyclone requires information for particular LFRic datatypes and what it does with or without this information are given below:

9.3.1 Fields

PSyclone must be able to determine the datatype of a field from the algorithm layer declarations. If it is not able to do this, PSyclone will abort with a message that indicates the problem.

Supported field types, their Fortran datatype and precisions are outlined in the table below:

Field Type	Fortran Datatype	Precision
`field_type`	`real`	`r_def`
`r_bl_field_type`	`real`	`r_bl`
`r_phys_field_type`	`real`	`r_phys`
`r_solver_field_type`	`real`	`r_solver`
`r_tran_field_type`	`real`	`r_tran`
`integer_field_type`	`integer`	`i_def`

9.3.2 Field Vectors

In addition to fields, LFRic supports an abstract vector type for fields, used in the LFRic solver API. Please note that these structures are different from the *field vector* implementation of field bundles in the PSyclone LFRic API interface.

The LFRic abstract vector type has precision-specific implementations. If PSyclone finds such a specifically declared field vector argument in the algorithm layer, e.g. `r_solver_field_vector_type`, it will assume that the actual field being referenced is of the same datatype and precision (see *above* for details). The correspondence between the available field types and their vector implementations is given in the table below (note that only `real`-valued fields have abstract vector implementations for now):

Field Type	Field Vector Type
`field_type`	`field_vector_type`
`r_bl_field_type`	`r_bl_field_vector_type`
`r_phys_field_type`	`r_phys_field_vector_type`
`r_solver_field_type`	`r_solver_field_vector_type`
`r_tran_field_type`	`r_tran_field_vector_type`

If PSyclone finds an argument that is declared as an `abstract_field_type` then it will not know the actual type of the argument. For instance, the following algorithm layer code will cause PSyclone to raise an exception:

```
! ...
class (abstract_vector_type), intent(inout) :: x
! ...
select type (x)
type is (field_vector_type)
  call invoke(testkern_type(x%vector(1)))
class default
  print *,"Error"
end select
! ...
```

The suggested solution to this is to add a pointer variable to the code that is of the required type. This pointer can then be associated with the argument and passed into the routine:

```
! ...
class (abstract_vector_type), target, intent(inout) :: x
```

(continues on next page)

(continued from previous page)

```
type(field_vector_type), pointer :: x_ptr
! ...
select type (x)
type is (field_vector_type)
  x_ptr => x
  call invoke(testkern_type(x_ptr%vector(1)))
class default
  print *,"Error"
end select
! ...
```

9.3.3 Scalars

It is not mandatory for PSyclone to be able to determine the datatype of a scalar from the algorithm layer. This constraint was considered to be too restrictive as PSyclone currently only examines the declarations in the same source file as the `invoke` when determining datatype. This means that if scalars are imported from other modules (as is often the case) then their datatype cannot be determined.

If the precision information for a scalar is found by PSyclone then this is used. If the scalar declaration is found and it contains no precision information then PSyclone will abort with a message that indicates the problem (since this violates LFRic coding standards). If no declaration information is found then default precision values are used, as specified in the PSyclone config file (`r_def` for `real`, `i_def` for `integer` and `l_def` for `logical`).

Supported precisions for scalars are outlined in the table below. If an unsupported scalar precision is found then PSyclone will abort with a message that indicates the problem.

Fortran Datatype	Supported Precision
`real`	`r_def`, `r_bl`, `r_phys`, `r_solver`, `r_tran`
`integer`	`i_def`
`logical`	`l_def`

9.3.4 LMA Operators

PSyclone must be able to determine the datatype of an LMA operator. If it is not able to do this, PSyclone will abort with a message that indicates the problem.

Supported LMA operator types, their Fortran datatype and precisions are outlined in the table below:

Operator Type	Fortran Datatype	Precision
`operator_type`	`real`	`r_def`
`r_solver_operator_type`	`real`	`r_solver`
`r_tran_operator_type`	`real`	`r_tran`

9.3.5 Column-wise Operators

It is not mandatory for PSyclone to be able to determine the datatype of a column-wise (CMA) operator. The reason for this is that only one datatype is supported, a `columnwise_operator_type` which contains `real`-valued data with precision `r_solver`. PSyclone can therefore simply add this datatype in the PSy-layer. However, if the datatype information is found in the algorithm layer and it is not of the expected type then PSyclone will abort with a message that indicates the problem.

9.3.6 Consistency

If PSyclone is able to determine the datatype of an LFRic datatype then PSyclone also checks that this datatype is consistent with the associated kernel metadata. If it is not consistent then PSyclone will abort with a message that indicates the problem.

9.4 PSy-layer

The general details of the PSy-layer are explained in the *PSy layer* section. This section describes any LFRic-specific issues.

9.4.1 Module name

The PSy-layer code is contained within a Fortran module. The name of the module is determined from the algorithm-layer name with "_psy" appended. The algorithm-layer name is the algorithm's module name if it is a module, its subroutine name if it is a subroutine that is not within a module, or the program name if it is a program.

So, for example, if the algorithm code is contained within a module called "fred" then the PSy-layer module name will be "fred_psy".

Argument Intents

LFRic *fields*, *field vectors*, *operators* and *column-wise operators* are objects that contain pointers to data rather than data. The data are accessed by proxies of these objects and modified in *kernels*. As the objects themselves are not modified in the PSy layer, their Fortran intents there are always `intent(in)`.

The Fortran intent of *scalars* is still defined by their *access metadata* as they are actual data. This means `intent(in)` for `GH_READ` and `intent(out)` for `GH_SUM` (more details in *meta_args* section below).

The intent of other data structures is mandated by the relevant LFRic API rules described in sections below.

9.5 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. In the LFRic API there are six different Kernel types; general purpose, CMA, inter-grid, domain, dof and *Built-ins*. In the case of built-ins, PSyclone generates the source of the kernels. This section explains the rules for the other five user-supplied kernel types and then goes on to describe their metadata and subroutine arguments.

Domain kernels are distinct from the other four user-supplied kernel types because they must be passed data for the whole domain rather than a single cell-column or dof. This permits the use of kernels that have not been written to conform to the single-column/dof approach which simplifies the integration with existing code. Obviously, any parallelisation in the 'domain' kernel must be consistent with that in the rest of the application. The motivation for

such kernels in LFRic is that they allow existing, "i-first" physics code to be called from the PSy layer. Since those routines currently contain their own, i-first looping structure (and associated OpenMP parallelisation), the most efficient way to use them is to avoid enclosing them within a loop in the PSy layer. This is a temporary measure and these kernels will ultimately be replaced once the LFRic infrastructure has support for i-first kernels (https://code.metoffice.gov.uk/trac/lfric/ticket/2154). At that point the looping (and associated parallelisation) will be put back into the PSy layer.

9.5.1 Rules for all User-Supplied Kernels that Operate on Cell-Columns

In the following, 'operator' refers to both LMA and CMA operator types.

1) A Kernel must have at least one argument that is a field, field vector, or operator. This rule reflects the fact that a Kernel operates on some subset of the whole domain (e.g. a cell-column) and is therefore designed to be called from within a loop that iterates over those subsets of the domain.

2) The continuity of the iteration space of the Kernel is determined from the function space of the modified argument (see Section *Supported Function Spaces* below). If more than one argument is modified then the iteration space is taken to be the largest required by any of those arguments. E.g. if a Kernel writes to two fields, the first on `W3` (discontinuous) and the second on `W1` (continuous), then the iteration space of that Kernel will be determined by the field on the continuous space.

3) If any of the modified arguments are declared with the generic function space metadata (e.g. `ANY_SPACE_<n>`, see *Supported Function Spaces*) and their actual space cannot be determined statically then the iteration space is assumed to be

    1) discontinuous for `ANY_DISCONTINUOUS_SPACE_<n>`;

    2) continuous for `ANY_SPACE_<n>` and `ANY_W2`. This assumption is always safe but leads to additional computation if the quantities being updated are actually on discontinuous function spaces.

4) Operators do not have halo operations operating on them as they are either cell- (LMA) or column-based (CMA) and therefore act like discontinuous fields.

5) Any Kernel that writes to an operator will have its iteration space expanded such that valid values for the operator are computed in the level-1 halo.

6) Any Kernel that reads from an operator must not access halos beyond level 1. In this case PSyclone will check that the Kernel does not require values beyond the level-1 halo. If it does then PSyclone will abort.

7) Any Kernel that takes an operator argument must not also take an `integer`-valued field as an argument.

9.5.2 Rules specific to General-Purpose Kernels without CMA Operators

1) General-purpose kernels with `operates_on = CELL_COLUMN` accept arguments of any of the following types: field, field vector, LMA operator, scalar (`real`, `integer` or `logical`).

2) A Kernel is permitted to write to more than one quantity (field or operator) and these quantities may be on the same or different function spaces.

3) A Kernel may not write to a scalar argument. (Only *built-ins* are permitted to do this.) Any scalar arguments must therefore be declared in the metadata as `GH_READ` - see *below*.

9.5.3 Rules for Kernels that work with CMA Operators

The LFRic API has support for kernels that assemble, apply (or inverse-apply) column-wise/Column Matrix Assembly (CMA) operators. Such operators may also be used by matrix-matrix kernels. There are thus three types of CMA-related kernels. Since, by definition, CMA operators only act on data within a column, they have no horizontal dependencies. Therefore, kernels that write to them may be parallelised without colouring.

All three CMA-related kernel types must obey the following rules:

1) Since a CMA operator only acts within a single column of data, stencil operations are not permitted.
2) No vector quantities (e.g. `GH_FIELD*3` - see below) are permitted as arguments.
3) The kernel must operate on cell-columns.

There are then additional rules specific to each of the three CMA kernel types. These are described below.

Assembly

CMA operators are themselves constructed from Local-Matrix-Assembly (LMA) operators. Therefore, any kernel which assembles a CMA operator must obey the following rules:

1) Have one or more LMA operators as read-only arguments.
2) Have exactly one CMA operator argument which must have write access.
3) Other types of argument (e.g. scalars or fields) are permitted but must be read-only.

Application and Inverse Application

Column-wise operators can only be applied to fields. CMA-Application kernels must therefore:

1) Have a single CMA operator as a read-only argument.
2) Have exactly two field arguments, one read-only and one that is written to.
3) The function spaces of the read and written fields must match the from and to spaces, respectively, of the supplied CMA operator.

Matrix-Matrix

A kernel that has just column-wise operators as arguments and zero or more read-only scalars is identified as performing a matrix-matrix operation. In this case:

1) Arguments must be CMA operators and, optionally, one or more scalars.
2) Exactly one of the CMA arguments must be written to while all other arguments must be read-only.

9.5.4 Rules for Inter-Grid Kernels

1) An inter-grid kernel is identified by the presence of a field or field-vector argument with the optional `mesh_arg` metadata element (see *Inter-Grid Metadata*).
2) An invoke that contains one or more inter-grid kernels must not contain any other kernel types. (This restriction is an implementation decision and could be lifted in future if there is a need.)
3) An inter-grid kernel is only permitted to have field or field-vector arguments.
4) All inter-grid kernel arguments must have the `mesh_arg` metadata entry.
5) An inter-grid kernel (and metadata) must have at least one field on each of the fine and coarse meshes. Specifying all fields as coarse or fine is forbidden.
6) Fields on different meshes must always live on different function spaces.
7) All fields on a given mesh must be on the same function space.
8) An inter-grid kernel must operate on cell-columns.

A consequence of Rules 5-7 is that an inter-grid kernel will only involve two function spaces.

9.5.5 Rules for User-Supplied Kernels that Operate on the Domain

The rules for kernels that have `operates_on = DOMAIN` are a subset of *those* for kernels that operate on a `CELL_COLUMN` without CMA Operators. Specifically:

1) Only *scalar*, *field* and *field vector* arguments are permitted.
2) All fields must be on discontinuous function spaces.
3) Stencil accesses are not permitted.

9.5.6 Rules for all User-Supplied Kernels that Operate on DoFs (DoF Kernels)

Kernels that have `operates_on = DOF` and *LFRic Built-ins* overlap significantly in their scope, and the conventions that DoF Kernels must follow are influenced by those for built-ins as a result. This includes *metadata arguments* and *valid data types and access modes*. Naming conventions for DoF Kernels should follow those for General-Purpose Kernels.

The list of rules for DoF Kernels is as follows:

1) A DoF Kernel must have at least one argument that is a field. This rule reflects that a Kernel operates on some subset of the whole domain and is therefore designed to be called from within a loop that iterates over those subsets of the domain. Only fields (as opposed to e.g. field vectors or operators) are accepted for DoF Kernels because only they have a single value at each DoF.
2) All Kernel arguments must be either fields or scalars (*real*- and/or *integer*-valued). DoF Kernels cannot accept operators.
3) All field arguments to a given DoF Kernel must be on the same function space so they have the same number of DoFs.
4) They must have at least one modified (i.e. written to) field argument. Unlike built-ins, this is not limited and more than one modified argument is allowed.
5) A Kernel may not write to a scalar argument. (Only built-ins are permitted to do this.) Any scalar arguments must therefore be declared in the metadata as *GH_READ* - see *below*

6) Kernels must be written to operate on a single DoF, such that field values at the same dof location/index can be provided to the Kernel within a loop over the DoFs of the function space of the field that is being updated.

9.5.7 Metadata

The code below outlines the elements of the LFRic API Kernel metadata, 1) '*meta_args*', 2) '*meta_funcs*', 3) '*meta_reference_element*', 4) '*meta_mesh*', 5) 'gh_shape' (*gh_shape and gh_evaluator_targets*), 6) '*operates_on*' and 7) '*procedure*':

```
type, public, extends(kernel_type) :: my_kernel_type
  type(arg_type) :: meta_args(...) = (/ ... /)
  type(func_type) :: meta_funcs(...) = (/ ... /)
  type(reference_element_data_type) :: meta_reference_element(...) = (/ ... /)
  type(mesh_data_type) :: meta_mesh(...) = (/ ... /)
  integer :: gh_shape = gh_quadrature_XYoZ
  integer :: operates_on = cell_column
contains
  procedure, nopass :: my_kernel_code
end type
```

These various metadata elements are discussed in order in the following sections.

meta_args

The `meta_args` array specifies information about data that the kernel code expects to be passed to it via its argument list. There is one entry in the `meta_args` array for each **scalar**, **field**, or **operator** passed into the Kernel and the order that these occur in the `meta_args` array must be the same as they are expected in the kernel code argument list. The entry must be of `arg_type` which itself contains metadata about the associated argument. The size of the `meta_args` array must correspond to the number of **scalars**, **fields** and **operators** passed into the Kernel.

Note: It makes no sense for a Kernel to have only **scalar** arguments (because the PSy layer will call a Kernel for each point in the spatial domain) and PSyclone will reject such Kernels.

For example, if there are a total of 2 **scalar / field / operator** entities being passed to the Kernel then the `meta_args` array will be of size 2 and there will be two `arg_type` entries:

```
type(arg_type) :: meta_args(2) = (/                                    &
     arg_type( ... ),                                                  &
     arg_type( ... )                                                   &
     /)
```

Argument metadata (information contained within the brackets of an `arg_type` entry), describes either a **scalar**, a **field** or an **operator** (either LMA or CMA).

The first argument-metadata entry describes whether the data that is being passed is for a scalar (`GH_SCALAR`), a field (`GH_FIELD`) or an operator (either `GH_OPERATOR` for LMA or `GH_COLUMNWISE_OPERATOR` for CMA). This information is mandatory.

Additionally, argument metadata can be used to describe a vector of fields (see the *Field Vector* section for more details).

As an example, the following `meta_args` metadata describes 4 entries, the first is a scalar, the next two are fields and the fourth is an operator. The third entry is a field vector of size 3.

```
type(arg_type) :: meta_args(4) = (/                                    &
     arg_type(GH_SCALAR, GH_REAL, ...),                                &
     arg_type(GH_FIELD, GH_INTEGER, ... ),                             &
     arg_type(GH_FIELD*3, GH_REAL, ... ),                              &
     arg_type(GH_OPERATOR, GH_REAL, ...)                               &
     /)
```

The second item in a metadata entry describes the Fortran primitive (intrinsic) type of the data of a kernel argument. The currently supported values are `GH_REAL`, `GH_INTEGER` and `GH_LOGICAL` for `real`, `integer` and `logical` data, respectively. This information is mandatory. Valid data types for each LFRic API argument type are specified later in this section (see *Valid Data Types*).

The third component of argument metadata describes how the Kernel makes use of the data being passed into it (the way it is accessed within a Kernel). This information is mandatory. There are currently 6 possible values of this metadata `GH_READ`, `GH_WRITE`, `GH_READWRITE`, `GH_INC`, `GH_READINC` and `GH_SUM`. However, not all combinations of metadata entries are valid and PSyclone will raise an exception if an invalid combination is specified. Valid combinations are specified later in this section (see *Valid Access Modes*).

- `GH_READ` indicates that the data is read and is unmodified.
- `GH_WRITE` indicates the data is modified in the Kernel before (optionally) being read. If any shared DoFs are written to then different iterations of the Kernel must write the same value.
- `GH_READWRITE` indicates that different iterations of a Kernel update quantities which do not share DoFs, such as operators and fields over discontinuous function spaces. If a Kernel modifies only discontinuous fields and/or operators there is no need for synchronisation or colouring when running such Kernels in parallel. However, modifying another field with a `GH_INC` access in a Kernel means that synchronisation or colouring is required for parallel runs.
- `GH_INC` indicates that different iterations of a Kernel make contributions to shared values. For example, values at cell faces may receive contributions from cells on either side of the face. This means that such a Kernel needs appropriate synchronisation (or colouring) to run in parallel.
- `GH_READINC` indicates that the data is first read and then subsequently incremented. Therefore this is equivalent to a `GH_READ` followed by a `GH_INC`.
- `GH_SUM` is an example of a reduction and is the only reduction currently supported in PSyclone. This metadata indicates that values are summed over calls to Kernel code.

For example:

```
type(arg_type) :: meta_args(6) = (/                                    &
     arg_type(GH_OPERATOR, GH_REAL,    GH_READ,      ... ),           &
     arg_type(GH_FIELD*3,  GH_REAL,    GH_WRITE,     ... ),           &
     arg_type(GH_FIELD,    GH_REAL,    GH_READWRITE, ... ),           &
     arg_type(GH_FIELD,    GH_INTEGER, GH_INC,       ... ),           &
     arg_type(GH_FIELD,    GH_REAL,    GH_READINC,   ... ),           &
     arg_type(GH_SCALAR,   GH_REAL,    GH_SUM)                        &
     /)
```

Warning: It is important that `GH_INC` is not incorrectly used in place of a `GH_READINC` access as it could result in the reading of data from a dirty outermost halo when run in parallel, giving incorrect results. The reason for this is that PSyclone does not add a halo exchange for the outermost modified halo level of a field before a loop that contains a `GH_INC` access to that field, i.e. a loop iterating to the level-`n` halo will result in a halo exchange to the level-(`n-1`) halo being added before the loop (which means no halo exchange is added when `n==1`). The reason this can be performed is because any computation in the outermost halo will be incorrect (will only compute partial sums) and

PSyclone therefore sets this halo level to dirty after the loop has completed. There is, therefore, no reason to make the values of the incremented field clean for the outermost modified halo. However, this optimisation does require that any (dirty) data in the outermost modified halo does not result in exceptions. With some compilers an exception can occur for a field that has not yet had its outermost halo data written to, i.e. if the uninitialised data is read. To avoid this potential problem in user code it is recommended that a redundant computation *transformation* is added to compute all `setval_c`, `setval_x` and `setval_random` Built-in calls (see *Built-ins*) to the same halo depth as the associated `GH_INC` access - which is level-1 without any redundant computation transformations being applied to the associated loops. This will guarantee that all data has been initialised with a value before it is incremented and avoid any potential exceptions.

Note: In the LFRic API only *Built-ins* are permitted to write to scalar arguments (and hence perform reductions). Furthermore, this permission is currently restricted to `real` scalars (`GH_SCALAR, GH_REAL`) as the LFRic infrastructure does not yet support `integer` and `logical` reductions.

For a scalar, the argument metadata contains only these three entries. However, fields and operators require further entries specifying function-space information. The meaning of these further entries differs depending on whether a field or an operator is being described.

In the case of an operator, the fourth and fifth arguments describe the `to` and `from` function spaces respectively. In the case of a field the fourth argument specifies the function space that the field lives on. More details about the supported function spaces are in subsection *Supported Function Spaces*.

For example, the metadata for a kernel that applies a column-wise operator to a field might look like:

```
type(arg_type) :: meta_args(3) = (/                                  &
     arg_type(GH_FIELD, GH_REAL, GH_INC, W1),                        &
     arg_type(GH_FIELD, GH_REAL, GH_READ, W2H),                      &
     arg_type(GH_COLUMNWISE_OPERATOR, GH_REAL, GH_READ, W1, W2H) &
     /)
```

In some cases a Kernel may be written so that it works for fields and/or operators from any type of a vector `W2*` space (all `W2*` spaces except for the `W2*trace` spaces, see Section *Supported Function Spaces* below). In this case the metadata should be specified as being `ANY_W2`.

Warning: In the current implementation it is assumed that all fields and/or operators specifying `ANY_W2` within a kernel will use the **same** function space. It is up to the user to ensure this is the case as otherwise invalid code would be generated.

It may be that a Kernel is written such that a field and/or operators may be on/map-between any function space(s). In this case the metadata should be specified as being one of `ANY_SPACE_1`, ..., `ANY_SPACE_<nmax>` (see *Supported Function Spaces*), with the number of spaces, `<nmax>`, being set in the *PSyclone configuration file* (see *here* for more details on this option).

If the generic function spaces are known to be discontinuous the metadata may be specified as being one of `ANY_DISCONTINUOUS_SPACE_1`, ..., `ANY_DISCONTINUOUS_SPACE_<nmax>` in order to avoid unnecessary computation into the halos (see rules for *user-supplied kernels* above). The reason for having different names is that a Kernel might be written to allow 2 or more arguments to be able to support any function space but for a particular call the function spaces may have to be the same as each other. Again, `<nmax>` is the *configurable* number of generalised discontinuous function spaces.

In the example below, the first field entry supports any function space but it must be the same as the operator's `to` function space. Similarly, the second field entry supports any function space but it must be the same as the operator's

from function space. Note, the metadata does not forbid ANY_SPACE_1 and ANY_SPACE_2 from being the same.

```
type(arg_type) :: meta_args(3) = (/                                          &
     arg_type(GH_FIELD,    GH_REAL, GH_INC,  ANY_SPACE_1),                   &
     arg_type(GH_FIELD*3,  GH_REAL, GH_INC,  ANY_SPACE_2),                   &
     arg_type(GH_OPERATOR, GH_REAL, GH_READ, ANY_SPACE_1, ANY_SPACE_2) &
     /)
```

Note also that the scope of this naming of any-space function spaces is restricted to the argument list of individual kernels. I.e. if an Invoke contains say, two kernel calls that each support arguments on any function space, e.g. ANY_SPACE_1, there is no requirement that these two function spaces be the same. Put another way, if an Invoke contained two calls of a kernel with arguments described by the above metadata then the first field argument passed to each kernel call need not be on the same space.

Valid Data Types

As mentioned earlier, the currently supported Fortran primitive (intrinsic) types for kernel argument data are real, integer and logical, described by the GH_REAL, GH_INTEGER and GH_LOGICAL metadata descriptors. Supported data types for each argument type are given in the table below (please note that *field vectors* follow the same rules as the *LFRic fields*):

Argument Type	Data Type
GH_SCALAR	GH_REAL, GH_INTEGER, GH_LOGICAL
GH_FIELD	GH_REAL, GH_INTEGER
GH_OPERATOR	GH_REAL
GH_COLUMNWISE_OPERATOR	GH_REAL

Valid Access Modes

As mentioned earlier, not all combinations of metadata are valid. Valid combinations for each argument type in user-defined Kernels are summarised here. All argument types (GH_SCALAR, GH_FIELD, GH_OPERATOR and GH_COLUMNWISE_OPERATOR) may be read within a Kernel and this is specified in metadata using GH_READ. At least one kernel argument must be listed as being modified. When data is *modified* in a user-supplied Kernel (i.e. a Kernel that operates on a CELL_COLUMN, see *iteration space metadata*) then the permitted access modes depend upon the argument type and the function space it is on:

Argument Type	Function Space	Access Type
GH_SCALAR	n/a	GH_READ
GH_FIELD	Discontinuous	GH_READ, GH_WRITE, GH_READWRITE
GH_FIELD	Continuous	GH_READ, GH_WRITE, GH_INC, GH_READINC
GH_OPERATOR	Any for both ‘to’ and ‘from’	GH_READ, GH_WRITE, GH_READWRITE
GH_COLUMNWISE_OPERATOR	Any for both ‘to’ and ‘from’	GH_READ, GH_WRITE, GH_READWRITE

Note that scalar arguments to user-defined Kernels must be read-only. Only *Built-ins* are permitted to modify scalar arguments. In practice this means that the only allowed access for the scalars in user-defined Kernels is GH_READ (see the allowed accesses for arguments in Built-ins in the *section below*).

Note also that a GH_FIELD argument that has GH_WRITE or GH_READWRITE as its access pattern must typically (see below) be on a horizontally-discontinuous function space (see *Supported Function Spaces* for the list of discontinuous function spaces). Parallelisation of the loop over the horizontal domain for a Kernel that updates such a field will not require colouring for either of the above cases (since there are no shared entities).

There is however an exception to this - certain Kernels may write to shared entities but each Kernel iteration is guaranteed to write the *same value* to a given shared DoF. In this case, provided that the first access to any such shared DoF is a write, the loop containing such a Kernel may be parallelised without colouring. Therefore, `GH_WRITE` access is permitted for `GH_FIELD` arguments on continuous function spaces. Obviously, care must be taken to ensure that the Kernel implementation satisfies the constraints just described as PSyclone cannot currently check this.

If a field is described as being on `ANY_SPACE_*`, there is currently no way to determine its continuity from the metadata (unless we can statically determine the space of the field being passed in). At the moment this type of a user-supplied Kernel is always treated as if it is updating a field that is on a function space that is continuous in the horizontal, even if it is not (see rules for *user-supplied kernels* above).

There is no restriction on the number and function spaces of other quantities that a general-purpose kernel can modify other than that it must modify at least one. The rules for kernels involving CMA operators, however, are stricter and only one argument may be modified (the CMA operator itself for assembly, a field for CMA-application and a CMA operator for matrix-matrix kernels). If a kernel writes to quantities on different function spaces then PSyclone generates loop bounds appropriate to the largest iteration space. This means that if a single kernel updates one quantity on a continuous function space and one on a discontinuous space then the resulting loop will include cells in the level-1 halo since they are required for a quantity on a continuous space. As a consequence, any quantities on a discontinuous space will then be computed redundantly in the level-1 halo. Currently PSyclone makes no attempt to take advantage of this (by e.g. setting the appropriate level-1 halo to 'clean').

PSyclone ensures that both CMA and LMA operators are computed (redundantly) out to the level-1 halo cells. This permits their use in kernels which modify quantities on continuous function spaces and also in subsequent redundant computation of other quantities on discontinuous function spaces. In conjunction with this, PSyclone also checks (when generating the PSy layer) that any kernels which read operator values do not do so beyond the level-1 halo. If any such accesses are found then PSyclone aborts.

Supported Function Spaces

As mentioned in the *Field* and *Field Vector* sections, the function space of an argument specifies how it maps onto the underlying topology and, additionally, whether the data at a point is a vector. In LFRic API the dimension of the basis function set for the scalar function spaces is 1 and for the vector function spaces is 3 (see the table in *Rules for General-Purpose Kernels* for the dimensions of the basis and differential basis functions).

Function spaces can share DoFs between cells in the horizontal, vertical or both directions. Depending on the function space and FEM order, the shared DoFs can lie on one or more cell entities (faces, edges and vertices) in each direction. This property is referred to as the **continuity** of a function space (horizontal, vertical or full). Alternatively, if there are no shared DoFs a function space is described as **discontinuous** (fully or in a particular direction).

The mixed FEM formulation is built on a foundation set of four function spaces described below.

- `W0` is the space of scalar functions with full continuity. The shared DoFs lie on cell vertices in the lowest order FEM and on all three entities in higher order FEM.
- `W1` is the space of vector functions with full continuity in the tangential direction only. In the lowest order FEM the shared DoFs lie on cell edges for each component, whereas in higher order they also lie on cell faces.
- `W2` is the space of vector functions with full continuity in the normal direction only. The shared DoFs lie on cell faces for each component.
- `W3` is the space of scalar functions with full discontinuity. All DoFs lie within the cell volume and are not shared across the cell boundaries.

Other spaces required for representation of scalar or component-wise vector variables are:

- `Wtheta` is the space of scalar functions based on the vertical part of `W2`, discontinuous in the horizontal and continuous in the vertical;

- `W2H` is the space of vector functions based on the horizontal part of `W2`, continuous in the horizontal and discontinuous in the vertical;
- `W2V` is the space of vector functions based on the vertical part of `W2`, discontinuous in the horizontal and continuous in the vertical;
- `W2broken` is the space of vector functions, locally identical to the `W2` space. However, DoFs are topologically discontinuous in all directions despite their placement on cell faces;
- `W2trace` is the space of scalar functions defined only on cell faces, resulting from taking the trace of a `W2` space. DoFs are shared between faces, hence making this space fully continuous;
- `W2Htrace` is the space of scalar functions defined only on cell faces in the horizontal, resulting from taking the trace of a `W2H` space. DoFs are shared between horizontal faces, hence making this space continuous in the horizontal and discontinuous in the vertical;
- `W2Vtrace` is the space of scalar functions defined only on cell faces in the vertical, resulting from taking the trace of a `W2V` space. DoFs are shared between vertical faces, hence making this space discontinuous in the horizontal and continuous in the vertical;
- `Wchi` is the space of scalar functions used to store coordinates in LFRic. It is fully discontinuous except for the coordinate order `0` when it becomes the `W0` space (i.e. fully continuous). Please see the next section for more details on this function space.

In addition to the specific function space metadata, there are also three generic function space metadata descriptors mentioned in sections above:

- `ANY_SPACE_<n>`, *n = 1, 2, ... nmax*, for when the function space of the argument(s) cannot be determined and/or for when a Kernel has been written so that it works with fields on any of the available spaces (as mentioned in the *meta_args section*, the number of spaces, `<nmax>`, is *configurable*);
- `ANY_DISCONTINUOUS_SPACE_<n>`, *n = 1, 2, ... nmax*, for when the function space of the argument(s) cannot be determined but is known to be discontinuous and/or for when a Kernel has been written so that it works with fields on any of the discontinuous spaces (again, the number of spaces, `<nmax>`, is *configurable*);
- `ANY_W2` for any type of a vector `W2*` function space, i.e. `W2`, `W2H`, `W2V` and `W2broken` but not `W2*trace` spaces.

As mentioned *previously* , `ANY_SPACE_<n>` and `ANY_W2` function space types are treated as continuous while `ANY_DISCONTINUOUS_SPACE_<n>` spaces are treated as discontinuous.

Note: The name and use of `ANY_W2` metadata (e.g. continuity and vector or/and scalar basis of `W2*` spaces the metadata can represent) are being reviewed in PSyclone issue #540.

Since the LFRic API operates on columns of data, function spaces are categorised as continuous or discontinuous with regard to their **continuity in the horizontal**. For example, a `GH_FIELD` that specifies `GH_INC` as its access pattern (see :ref:lfric-kernel-valid-access: above) may be continuous in the vertical (and discontinuous in the horizontal), continuous in the horizontal (and discontinuous in the vertical), or continuous in both. In each case the code is the same. This principle of horizontal continuity also applies to the three generic `ANY_*_*` function space identifiers above. The valid metadata values for continuous and discontinuous function spaces are summarised in the table below.

Function Space Continuity	Function Space Name
Continuous	W0, W1, W2, W2H, W2trace, W2Htrace, ANY_W2, ANY_SPACE_<n>
Discontinuous	W2broken, W2V, W2Vtrace, W3, Wtheta, ANY_DISCONTINUOUS_SPACE_<n>

Horizontally discontinuous function spaces and fields over them will not need colouring so PSyclone does not perform it. If such attempt is made, PSyclone will raise a `Generation Error` in the **Dynamo0p3ColourTrans** transformation (see *Transformations* for more details on transformations). An example of fields iterating over a discontinuous function space `Wtheta` is given in `examples/lfric/eg9`, with the `GH_READWRITE` access descriptor denoting an update to

the relevant fields. This example also demonstrates how to only colour loops over continuous function spaces when transformations are applied.

Read-Only Function Spaces

LFRic supports the concept of a **read-only function space**. A field on such a function space must not be modified by any kernels contained within `invoke` calls (i.e. within any code that PSyclone is responsible for). Further, a field on a read-only function space must contain clean halos in order to avoid any halo exchanges that would occur if the field is read within a kernel where redundant computation is performed.

The primary reason for including a read-only function space is that it does not need any halo-exchange support e.g. it does not require a routing table, which can reduce the memory footprint.

Currently `Wchi` is the only read-only function space in LFRic.

As a read-only function space is not modified, it does not matter whether it is classified as continuous or discontinuous. LFRic therefore treats read-only as a third category of function space.

Optional Field Metadata

A field entry in the meta_args array may have an optional fifth element. This element describes either a stencil access or, for inter-grid kernels, which mesh the field is on. Since an inter-grid kernel is not permitted to have stencil accesses, these two options are mutually exclusive. The metadata for each case is described in the following sections.

Stencil Metadata

Stencil metadata specifies that the corresponding field argument is accessed as a stencil operation within the Kernel. Stencil metadata only makes sense if the associated field is read within a Kernel i.e. it only makes sense to specify stencil metadata if the first entry is `GH_FIELD` and the second entry is `GH_READ`.

Stencil metadata is written in the following format:

```
STENCIL(type)
```

where `type` may be one of `X1D`, `Y1D`, `XORY1D`, `CROSS`, `CROSS2D` or `REGION`. As the stencil `extent` (the maximum distance from the central cell that the stencil extends) is not provided in the metadata, it is expected to be provided by the algorithm writer as part of the `invoke` call (see Section *Stencil Extent*). As there is currently no way to specify a fixed extent value for stencils in the Kernel metadata, Kernels must therefore be written to support different values of extent (i.e. stencils with a variable number of cells).

The `XORY1D` stencil type indicates that the Kernel can accept either `X1D` or `Y1D` stencils. In this case it is up to the algorithm developer to specify which of these it is from the algorithm layer as part of the `invoke` call (see Section *Stencil Extent*).

For example, the following stencil (with `extent=2`):

```
| 3 | 2 | 1 | 4 | 5 |
```

would be declared as:

```
STENCIL(X1D)
```

and the following stencil (with `extent=2`):

```
|   |   | 9 |   |   |
|   |   | 8 |   |   |
| 3 | 2 | 1 | 6 | 7 |
|   |   | 4 |   |   |
|   |   | 5 |   |   |
```

would be declared as:

```
STENCIL(CROSS)
```

The `REGION` stencil references a block of cells:

```
| 9 | 8 | 7 |
| 2 | 1 | 6 |
| 3 | 4 | 5 |
```

and would be declared as:

```
STENCIL(REGION)
```

Below is an example of stencil information within the full kernel metadata:

```
type(arg_type) :: meta_args(3) = (/                                    &
     arg_type(GH_FIELD,    GH_REAL, GH_INC,  W1),                      &
     arg_type(GH_FIELD,    GH_REAL, GH_READ, W2H, STENCIL(CROSS)),     &
     arg_type(GH_OPERATOR, GH_REAL, GH_READ, W1, W2H)                  &
     /)
```

There is a full example of this distributed with PSyclone. It may be found in `examples/lfric/eg5`.

Inter-Grid Metadata

The alternative form of the optional fifth metadata argument for a field specifies which mesh the associated field is on. This is required for inter-grid kernels which perform prolongation or restriction operations on fields (or field vectors) existing on grids of different resolutions.

Mesh metadata is written in the following format:

```
mesh_arg=type
```

where `type` may be one of `GH_COARSE` or `GH_FINE`. Any kernel having a field argument with this metadata is assumed to be an inter-grid kernel and, as such, all of its other arguments (which must also be fields) must have it specified too. An example of the metadata for such a kernel is given below:

```
type(arg_type) :: meta_args(2) = (/                                        &
    arg_type(GH_FIELD, GH_REAL, GH_READWRITE, ANY_DISCONTINUOUS_SPACE_1, &
                                              mesh_arg=GH_COARSE), &
    arg_type(GH_FIELD, GH_REAL, GH_READ,      ANY_DISCONTINUOUS_SPACE_2, &
                                              mesh_arg=GH_FINE  )  &
    /)
```

Note that an inter-grid kernel must have at least one field (or field- vector) argument on each mesh type. Fields that are on different meshes cannot be on the same function space while those on the same mesh must also be on the same function space.

Column-wise Operators (CMA)

In this section we provide example metadata for each of the three recognised kernel types involving CMA operators.

Column-wise operators are constructed from cell-wise (local) operators. Therefore, in order to **assemble** a CMA operator, a kernel must have at least one read-only LMA operator, e.g.:

```
type(arg_type) :: meta_args(2) = (/                                              &
     arg_type(GH_OPERATOR,            GH_REAL, GH_READ,  ANY_SPACE_1, ANY_SPACE_2), &
     arg_type(GH_COLUMNWISE_OPERATOR, GH_REAL, GH_WRITE, ANY_SPACE_1, ANY_SPACE_2)  &
     /)
```

CMA operators (and their inverse) are **applied** to fields. Therefore any kernel of this type must have one read-only CMA operator, one read-only field and a field that is updated, e.g.:

```
type(arg_type) :: meta_args(3) = (/                                              &
     arg_type(GH_FIELD,               GH_REAL, GH_INC,  ANY_SPACE_1),            &
     arg_type(GH_FIELD,               GH_REAL, GH_READ, ANY_SPACE_2),            &
     arg_type(GH_COLUMNWISE_OPERATOR, GH_REAL, GH_READ, ANY_SPACE_1, ANY_SPACE_2) &
     /)
```

Matrix-matrix kernels compute the product/linear combination of CMA operators. They must therefore have one such operator that is updated while the rest are read-only. They may also have read-only scalar arguments, e.g.:

```
type(arg_type) :: meta_args(3) = (/                                              &
     arg_type(GH_COLUMNWISE_OPERATOR, GH_REAL, GH_WRITE, ANY_SPACE_1, ANY_SPACE_2), &
     arg_type(GH_COLUMNWISE_OPERATOR, GH_REAL, GH_READ,  ANY_SPACE_1, ANY_SPACE_2), &
     arg_type(GH_COLUMNWISE_OPERATOR, GH_REAL, GH_READ,  ANY_SPACE_1, ANY_SPACE_2), &
     arg_type(GH_SCALAR,              GH_REAL, GH_READ) /)
```

Note: The order with which arguments are specified in metadata for CMA kernels does not affect the process of identifying the type of kernel (whether it is assembly, matrix-matrix etc.)

meta_funcs

The (optional) second component of kernel metadata specifies whether any quadrature or evaluator data is required for a given function space. (If no quadrature or evaluator data is required then this metadata should be omitted.) Consider the following kernel metadata:

```
type, extends(kernel_type) :: testkern_operator_type
  type(arg_type), dimension(3) :: meta_args =                 &
      (/ arg_type(gh_operator, gh_real,    gh_write, w0, w0), &
         arg_type(gh_field*3,  gh_real,    gh_read,  w1),     &
         arg_type(gh_scalar,   gh_integer, gh_read)           &
      /)
  type(func_type) :: meta_funcs(2) =                          &
      (/ func_type(w0, gh_basis, gh_diff_basis)               &
         func_type(w1, gh_basis)                              &
      /)
  integer :: gh_shape = gh_quadrature_XYoZ
  integer :: operates_on = cell_column
```

(continues on next page)

(continued from previous page)

```
contains
  procedure, nopass :: code => testkern_operator_code
end type testkern_operator_type
```

The `arg_type` component of this metadata describes a kernel that takes three arguments (an operator, a field and an `integer` scalar). Following the `meta_args` array we now have a `meta_funcs` array. This allows the user to specify that the kernel requires basis functions (`gh_basis`) and/or the differential of the basis functions (`gh_diff_basis`) on one or more of the function spaces associated with the arguments listed in `meta_args`. In this case we require both for the W0 function space but only basis functions for W1.

Note: Basis and differential basis functions for both `real`- and `integer`-valued field arguments have `real` values on the points on which these functions are *required*.

---

meta_reference_element

A kernel that requires properties of the reference element in LFRic specifies those properties through the `meta_reference_element` metadata entry. (If no reference element properties are required then this metadata should be omitted.) Consider the following example kernel metadata:

```
type, extends(kernel_type) :: testkern_type
  type(arg_type), dimension(2) :: meta_args =      &
      (/ arg_type(gh_field, gh_real, gh_read, w1), &
         arg_type(gh_field, gh_real, gh_inc,  w0) /)
  type(reference_element_data_type), dimension(2) ::               &
    meta_reference_element =                                       &
      (/ reference_element_data_type(normals_to_horizontal_faces), &
         reference_element_data_type(normals_to_vertical_faces) /)
contains
  procedure, nopass :: code => testkern_code
end type testkern_type
```

This metadata specifies that the `testkern_type` kernel requires two properties of the reference element. The supported properties are listed below:

Name	Description
normals_to_horizontal_faces	Array of normals pointing in the positive (x, y, z) axis direction for each horizontal face indexed as (component, face).
normals_to_vertical_faces	Array of normals pointing in the positive (x, y, z) axis direction for each vertical face indexed as (component, face).
normals_to_faces	Array of normals pointing in the positive (x, y, z) axis direction for each face indexed as (component, face).
outward_normals_to_horizontal_faces	Array of outward-pointing normals for each horizontal face indexed as (component, face).
outward_normals_to_vertical_faces	Array of outward-pointing normals for each vertical face indexed as (component, face).
outward_normals_to_faces	Array of outward-pointing normals for each face indexed as (component, face).

meta_mesh

A kernel that requires properties of the LFRic mesh object specifies those properties through the `meta_mesh` metadata entry. (If no mesh properties are required then this metadata should be omitted.) Consider the following example kernel metadata:

```
type, extends(kernel_type) :: testkern_type
  type(arg_type), dimension(2) :: meta_args =      &
      (/ arg_type(gh_field, gh_real, gh_read, w1), &
         arg_type(gh_field, gh_real, gh_inc,  w0) /)
  type(mesh_data_type), dimension(1) ::            &
    meta_mesh =                                    &
      (/ mesh_data_type(adjacent_face) /)
contains
  procedure, nopass :: code => testkern_code
end type testkern_type
```

This metadata specifies that the `testkern_type` kernel requires one property of the mesh. There is currently one supported property:

Name	Description
adjacent_face	Local ID of a neighbouring face in each horizontally-adjacent cell indexed as (face).

gh_shape and gh_evaluator_targets

If a kernel requires basis or differential-basis functions then the metadata must also specify the set of points on which these functions are required. This information is provided by the `gh_shape` component of the metadata. Currently PSyclone supports four shapes; `gh_quadrature_XYoZ` for Gaussian quadrature points, `gh_quadrature_face` for quadrature points on cell faces, `gh_quadrature_edge` for quadrature points on cell edges and `gh_evaluator` for evaluation at nodal points. If a kernel requires just one of these then `gh_shape` is an `integer` scalar. However, if more than one is required then `gh_shape` becomes a one-dimensional, `integer` array, e.g.:

```
integer :: gh_shape(2) = (/ gh_quadrature_face, gh_quadrature_edge /)
```

If a kernel requires an evaluator then there are two options: if an evaluator is required for multiple function spaces then these can be specified using the additional `gh_evaluator_targets` metadata entry. This entry is a one-dimensional, `integer` array containing the desired function spaces. For example, to request basis/differential-basis functions evaluated on both W0 and W1, the metadata would be:

```
integer :: gh_shape = gh_evaluator
integer :: gh_evaluator_targets(2) = (/W0, W1/)
```

The kernel must have an argument (field or operator) on each of the function spaces listed in `gh_evaluator_targets`. The default behaviour if `gh_evaluator_targets` is not specified is to provide evaluators for each function space associated with the quantities that the kernel is updating. All necessary data is extracted in the PSy layer and passed to the kernel(s) as required - nothing is required from the Algorithm layer. If a kernel requires quadrature on the other hand, the Algorithm writer must supply a `quadrature_type` object for each specified quadrature as the last argument(s) to the kernel (see Section *Quadrature*).

Note that it is an error for kernel metadata to specify a value for `gh_shape` if no basis or differential-basis functions are required. It is also an error to specify `gh_evaluator_targets` if the kernel does not require an evaluator (i.e. `gh_shape != gh_evaluator`).

operates_on

The fourth type of metadata provided is `OPERATES_ON`. This specifies that the Kernel has been written with the assumption that it is supplied with the specified data for each field/operator argument. The possible values for `OPERATES_ON` and their interpretation are summarised in the following table:

operates_on	Data passed for each field/operator argument
`cell_column`	Single column of cells from the owned region (except when performing an INC operation on continuous fields when it will include one level of halo cells).
`halo_cell_column`	Single column of cells exclusively from halo region.
`owned_and_halo_cell_column`	Single column of cells but iteration space will include both owned and halo regions.
`dof`	Single DoF .
`domain`	All columns of cells in the (sub-)domain.

(For a description of the concepts of 'owned' and 'halo' cells please see the LFRic.)

procedure

The fifth and final type of metadata is `procedure` metadata. This specifies the name of the Kernel subroutine that this metadata describes.

For example:

```
procedure, nopass :: my_kernel_subroutine
```

9.5.8 Subroutine

Rules for General-Purpose Kernels

The arguments to general-purpose kernels (those that do not involve either CMA operators or prolongation/restriction operations) that operate on cell-columns follow a set of rules which have been specified for the LFRic API. These rules are encoded in the `generate()` method within the `ArgOrdering` abstract class in the `dynamo0p3.py` file. The rules, along with PSyclone's naming conventions, are:

1) If an LMA operator is passed then include the `cells` argument. `cells` is an `integer` of kind `i_def` and has intent `in`.
2) Include `nlayers`, the number of layers in a column. `nlayers` is an `integer` of kind `i_def` and has intent `in`. PSyclone will obtain the value of `nlayers` to use for a particular kernel from the first field (in the argument list) that is written to.
3) For each scalar/field/vector_field/operator in the order specified by the meta_args metadata:
    1) If the current entry is a scalar quantity then include the Fortran variable in the argument list. The intent is determined from the metadata (see *meta_args* for an explanation).
    2) If the current entry is a field then include the field array. The field array name is currently specified as being `"field_"<argument_position>"_"<field_function_space>`. A field array is a rank-1, `real` array with extent equal to the number of unique degrees of freedom for the space that the field is on. Its precision (kind) depends on how it is defined in the algorithm layer, see the *Mixed Precision* section for more details. This value is passed in separately. Again, the intent is determined from the metadata (see *meta_args*).
        1) If the field entry has a stencil access then add an `integer` (or if the stencil is of type `CROSS2D`, an `integer` rank-1 array of extent 4 and kind `i_def`) stencil-size argument with intent `in`. This will

supply the number of cells in the stencil or, in the case of the `CROSS2D` stencil, the number of cells in each branch of the stencil.

2) If the stencil is of type `CROSS2D` then an `integer` of kind `i_def` and intent `in` for the max branch length is needed. This is used in defining the dimensions of the stencil dofmap array and is required due to the varying length of the branches of the stencil when used on planar meshes.

3) Also needed is a stencil dofmap array of type `integer`, kind `i_def` and intent `in` in either 2 or 3 dimensions. For a `CROSS2D` stencil the array needs dimensions of (number-of-dofs-in-cell, max-branch-length, 4). All other stencils need dimensions of (number-of-dofs-in-cell, stencil-size).

4) If the field entry stencil access is of type `XORY1D` then add an additional `integer` direction argument of kind `i_def` and with intent `in`.

3) If the current entry is a field vector then for each dimension of the vector, include a field array. The field array name is specified as being using `"field_"<argument_position>"_"<field_function_space>"_v"<vector_position>`. A field array in a field vector is declared in the same way as a field array (described in the previous step).

4) If the current entry is an operator then first include an `integer` extent of kind `i_def`. The name of this extent is `<operator_name>"_ncell_3d"`. Next include the operator. This is a rank-3, `real` array. Its precision (kind) depends on how it is defined in the algorithm layer, see the *Mixed Precision* section for more details. The extents of the first two dimensions are the local degrees of freedom for the `to` and `from` function spaces, respectively, and that of the third is `<operator_name>"_ncell_3d"`. The name of the operator is `"op_"<argument_position>`. Again the intent is determined from the metadata (see *meta_args*).

4) For each function space in the order they appear in the metadata arguments (the `to` function space of an operator is considered to be before the `from` function space of the same operator as it appears first in lexicographic order)

1) Include the number of local degrees of freedom (i.e. number per-cell) for the function space. This is an `integer` of kind `i_def` and has intent `in`. The name of this argument is `"ndf_"<field_function_space>`.

2) If there is a field on this space

1) Include the unique number of degrees of freedom for the function space. This is an `integer` of kind `i_def` and has intent `in`. The name of this argument is `"undf_"<field_function_space>`.

2) Include the **dofmap** for this function space. This is an `integer` array of kind `i_def` with intent `in`. It has one dimension sized by the local degrees of freedom for the function space.

3) For each operation on the function space (`basis`, `diff_basis`), in the order specified in the metadata, pass `real` arrays of kind `r_def` with intent `in`. For each shape specified in the `gh_shape` metadata entry:

1) If shape is `gh_quadrature_*` then the arrays are of rank four and are named `"basis_"<field_function_space>_<quadrature_arg_name>` or `"diff_basis_"<field_function_space>_<quadrature_arg_name>`, as appropriate:

1) If shape is `gh_quadrature_xyoz` then the arrays have extent (`dimension`, `number_of_dofs`, `np_xy`, `np_z`).

2) If shape is `gh_quadrature_face` or `gh_quadrature_edge` then the arrays have extent (`dimension`, `number_of_dofs`, `np_xyz`, `nfaces` or `nedges`).

2) If shape is `gh_evaluator` then we pass one array for each target function space (i.e. as specified by `gh_evaluator_targets`). Each of these arrays are of rank three with extent (`dimension`, `number_of_dofs`, `ndf_<target_function_space>`). The name of the argument is `"basis_"<field_function_space>"_on_"<target_function_space>` or `"diff_basis_"<field_function_space>"_on_"<target_function_space>`, as appropriate.

Here <quadrature_arg_name> is the name of the corresponding quadrature object being passed to the Invoke. `dimension` is 1 or 3 and depends upon the function space (see *Supported Function Spaces* above for more information) and whether or not it is a basis or a differential basis function (see the table below). `number_of_dofs` is the number of degrees of freedom (DoFs) associated with the function space and `np_*` are the number of points to be evaluated: i) `*_xyz` in all directions (3D); ii) `*_xy` in the horizontal plane (2D); iii) `*_x, *_y` in the horizontal (1D); and iv) `*_z` in the vertical (1D). `nfaces` and `nedges` are the number of horizontal faces/edges obtained from the appropriate quadrature object supplied to the Invoke.

Function Type	Dimension	Function Space Name
Basis	1	W0, W2trace, W2Htrace, W2Vtrace, W3, Wtheta, Wchi
	3	W1, W2, W2H, W2V, W2broken, ANY_W2
Differential Basis	1	W2, W2H, W2V, W2broken, ANY_W2
	3	W0, W1, W2trace, W2Htrace, W2Vtrace, W3, Wtheta, Wchi

5) If either the `normals_to_horizontal_faces` or `outward_normals_to_horizontal_faces` properties of the reference element are required then pass the number of horizontal faces of the reference element (`nfaces_re_h`). Similarly, if either the `normals_to_vertical_faces` or `outward_normals_to_vertical_faces` are required then pass the number of vertical faces (`nfaces_re_v`). This also holds for the `normals_to_faces` and `outward_normals_to_faces` where the number of all faces of the reference element (`nfaces_re`) is passed to the kernel. (All of these quantities are integers of kind `i_def`.) Then, in the order specified in the `meta_reference_element` metadata:
    1) For the `normals_to_horizontal/vertical_faces`, pass a rank-2 `integer` array of kind `i_def` with dimensions `(3, nfaces_re_h/v)`.
    2) For the `outward_normals_to_horizontal/vertical_faces`, pass a rank-2 `integer` array of kind `i_def` with dimensions `(3, nfaces_re_h/v)`.
    3) For `normals_to_faces` or `outward_normals_to_faces` pass a rank-2 `integer` array of kind `i_def` with dimensions `(3, nfaces_re)`.
6) If the `adjacent_face` mesh property is required then:
    1) If the number of horizontal cell faces obtained from the reference element (`nfaces_re_h`) is not already being passed to the kernel (due to rule 5 above) then supply it here. This is an `integer` of kind `i_def`.
    2) Pass a rank-1, `integer` array of kind `i_def` and extent `nfaces_re_h`.
7) If Quadrature is required (`gh_shape = gh_quadrature_*`) then, for each shape in the order specified in the `gh_shape` metadata:
    1) Include `integer`, scalar arguments of kind `i_def` with intent `in` that specify the extent of the basis/diff-basis arrays:
        1) If `gh_shape` is `gh_quadrature_XYoZ` then pass `np_xy_`<quadrature_arg_name> and `np_z_`<quadrature_arg_name>.
        2) If `gh_shape` is `gh_quadrature_face/_edge` then pass `nfaces/nedges_`<quadrature_arg_name> and `np_xyz_`<quadrature_arg_name>.
    2) Include weights which are `real` arrays of kind `r_def`:
        1) If `gh_quadrature_XYoZ` pass in `weights_xz_`<quadrature_arg_name> (rank one, extent `np_xy_`<quadrature_arg_name>) and `weights_z_`<quadrature_arg_name> (rank one, extent `np_z_`<quadrature_arg_name>).
        2) If `gh_quadrature_face/_edge` pass in `weights_xyz_`<quadrature_arg_name> (rank two with extents [`np_xyz_`<quadrature_arg_name>, `nfaces/nedges_`<quadrature_arg_name>]).

Examples

For instance, if a kernel has only one written argument and requires an evaluator then its metadata might be:

```
type, extends(kernel_type) :: testkern_operator_type
   type(arg_type), dimension(2) :: meta_args =                &
        (/ arg_type(gh_operator, gh_real, gh_write, w0, w1), &
           arg_type(gh_field*3,  gh_real, gh_read,  w0) /)
   type(func_type) :: meta_funcs(1) =                         &
        (/ func_type(w0, gh_basis) /)
   integer :: operates_on = cell_column
   integer :: gh_shape = gh_evaluator
 contains
   procedure, nopass :: code => testkern_operator_code
end type testkern_operator_type
```

then we only pass the basis functions evaluated on `W0` (the space of the written kernel argument). The subroutine arguments will therefore be:

```
subroutine testkern_operator_code(cell, nlayers, ncell_3d,        &
     local_stencil, xdata, ydata, zdata, ndf_w0, undf_w0, map_w0, &
     basis_w0_on_w0, ndf_w1)
```

where `local_stencil` is the operator, `xdata`, `ydata` etc. are the three components of the field vector and `map_w0` is the dofmap for the `W0` function space.

If instead, `gh_evaluator_targets` is specified in the metadata:

```
type, extends(kernel_type) :: testkern_operator_type
   type(arg_type), dimension(2) :: meta_args =                &
        (/ arg_type(gh_operator, gh_real, gh_write, w0, w1), &
           arg_type(gh_field*3,  gh_real, gh_read,  w0) /)
   type(func_type) :: meta_funcs(1) =                         &
        (/ func_type(w0, gh_basis) /)
   integer :: operates_on = cell_column
   integer :: gh_shape = gh_evaluator
   integer :: gh_evaluator_targets(2) = (/W0, W1/)
 contains
   procedure, nopass :: code => testkern_operator_code
end type testkern_operator_type
```

then we will need to pass two sets of basis functions (evaluated at `W0` and at `W1`):

```
subroutine testkern_operator_code(cell, nlayers, ncell_3d,        &
     local_stencil, xdata, ydata, zdata, ndf_w0, undf_w0, map_w0, &
     basis_w0_on_w0, basis_w0_on_w1, ndf_w1)
```

If the metadata specifies that a kernel requires both an evaluator and quadrature:

```
type, extends(kernel_type) :: testkern_operator_type
   type(arg_type), dimension(2) :: meta_args =                &
        (/ arg_type(gh_operator, gh_real, gh_write, w0, w1), &
           arg_type(gh_field*3,  gh_real, gh_read,  w0) /)
   type(func_type) :: meta_funcs(1) =                         &
```

(continues on next page)

(continued from previous page)

```
        (/ func_type(w0, gh_basis) /)
   integer :: operates_on = cell_column
   integer :: gh_shape(2) = (/ gh_evaluator, gh_quadrature_face /)
 contains
   procedure, nopass :: code => testkern_operator_code
end type testkern_operator_type
```

then we will need to pass basis functions for both the evaluator and the quadrature (where `qr_face` is the name of the face-quadrature object passed to the Invoke):

```
subroutine testkern_operator_code(cell, nlayers, ncell_3d,              &
     local_stencil, xdata, ydata, zdata, ndf_w0, undf_w0, map_w0,       &
     basis_w0_on_w0, basis_w0_qr_face, ndf_w1,                          &
     np_xyz_qr_face, nfaces_qr_face, weights_xyz_qr_face)
```

If the metadata specifies that the kernel requires a property of the reference element:

```
type, extends(kernel_type) :: testkern_operator_type
   type(arg_type), dimension(2) :: meta_args =                &
        (/ arg_type(gh_operator, gh_real, gh_write, w0, w1), &
           arg_type(gh_field*3,  gh_real, gh_read,  w0) /)
   type(reference_element_data_type) :: meta_reference_element(1) =  &
        (/ reference_element_data_type(normals_to_horizontal_faces) /)
   integer :: operates_on = cell_column
 contains
   procedure, nopass :: code => testkern_operator_code
end type testkern_operator_type
```

then the kernel must be passed the number of faces of the reference element and the array of face normals in the specified direction (here horizontal):

```
subroutine testkern_operator_code(cell, nlayers, ncell_3d,        &
     local_stencil, xdata, ydata, zdata, ndf_w0, undf_w0, map_w0, &
     nfaces_re_h, normals_face_h)
```

Rules for CMA Kernels

Kernels involving CMA operators are restricted to just three types; assembly, application/inverse-application and matrix-matrix. We give the rules for each of these in the sections below.

Assembly

An assembly kernel requires the column-banded dofmap for both the to- and from-function spaces of the CMA operator being assembled as well as the number of DoFs for each of the dofmaps. The full set of rules is:

1) Include the `cell` argument. `cell` is an `integer` of kind `i_def``and has intent ``in`.
2) Include `nlayers`, the number of layers in a column. `nlayers` is an `integer` of kind `i_def` and has intent `in`.
3) Include the total number of cells in the 2D mesh (including halos), `ncell_2d`, which is an `integer` of kind `i_def` with intent `in`.
4) Include the total number of cells, `ncell_3d`, which is an `integer` of kind `i_def` with intent `in`.

5) For each argument in the `meta_args` metadata array:

    1) If it is a LMA operator, include a `real`, 3-dimensional array. The first two dimensions are the local degrees of freedom for the `to` and `from` spaces, respectively. The third dimension is `ncell_3d`. The precision of the array depends on how it is defined in the algorithm layer, see the *Mixed Precision* section for more details;

    2) If it is a CMA operator, include a `real`, 3-dimensional array of kind `r_solver`. The first dimension is `"bandwidth_"<operator_name>`, the second is `"nrow_"<operator_name>`, and the third is `ncell_2d`.

        1) Include the number of rows in the banded matrix. This is an `integer` of kind `i_def` with intent `in` and is named as `"nrow_"<operator_name>`.

        2) If the from-space of the operator is *not* the same as the to-space then include the number of columns in the banded matrix. This is an `integer` of kind `i_def` with intent `in` and is named as `"ncol_"<operator_name>`.

        3) Include the bandwidth of the banded matrix. This is an `integer` of kind `i_def` with intent `in` and is named as `"bandwidth_"<operator_name>`.

        4) Include banded-matrix parameter `alpha`. This is an `integer` of kind `i_def` with intent `in` and is named as `"alpha_"<operator_name>`.

        5) Include banded-matrix parameter `beta`. This is an `integer` of kind `i_def` with intent `in` and is named as `"beta_"<operator_name>`.

        6) Include banded-matrix parameter `gamma_m`. This is an `integer` of kind `i_def` with intent `in` and is named as `"gamma_m_"<operator_name>`.

        7) Include banded-matrix parameter `gamma_p`. This is an `integer` of kind `i_def` with intent `in` and is named as `"gamma_p_"<operator_name>`.

    3) If it is a field or scalar argument then include arguments following the same rules as for general-purpose kernels.

6) For each unique function space in the order they appear in the metadata arguments (the `to` function space of an operator is considered to be before the `from` function space of the same operator as it appears first in lexicographic order):

    1) Include the number of degrees of freedom per cell for the space. This is an `integer` of kind `i_def` with intent `in`. The name of this argument is `"ndf_"<arg_function_space>`.

    2) If there is a field on this space then:

        1) Include the unique number of degrees of freedom for the function space. This is an `integer` of kind `i_def` and has intent `in`. The name of this argument is `"undf_"<field_function_space>`.

        2) Include the dofmap for this space. This is an `integer` array of kind `i_def` with intent `in`. It has one dimension sized by the local degrees of freedom for the function space.

    3) If the CMA operator has this space as its to/from space then include the column-banded dofmap, the list of offsets for the to/from-space. This is an `integer` array of rank 2 and kind `i_def`. The first dimension is `"ndf_"<arg_function_space>` and the second is `nlayers`.

Application/Inverse-Application

A kernel applying a CMA operator requires the column-indirection dofmap for both the to- and from-function spaces of the CMA operator. Since it does not have any LMA operator arguments it does not require the `ncell_3d` and `nlayers` scalar arguments. (Since a column-wise operator is, by definition, assembled for a whole column, there is no loop over levels when applying it.) The full set of rules is then:

1) Include the `cell` argument. `cell` is an `integer` of kind `i_def` and has intent `in`.
2) Include the total number of cells in the 2D mesh (including halos), `ncell_2d`, which is an `integer` of kind `i_def` with intent `in`.
3) For each argument in the `meta_args` metadata array:
    1) If it is a field, include the field array. This is a `real` array of rank 1. Its precision (kind) depends on how it is defined in the algorithm layer, see the *Mixed Precision*. The field array name is currently specified as being `"field_"<argument_position>"_"<field_function_space>`. The extent of the array is the number of unique degrees of freedom for the function space that the field is on. This value is passed in separately. The intent of the argument is determined from the metadata (see *meta_args*);
    2) If it is a CMA operator, include it and its associated parameters (see Rule 5 of CMA Assembly kernels).
4) For each of the unique function spaces encountered in the metadata arguments (the `to` function space of an operator is considered to be before the `from` function space of the same operator as it appears first in lexicographic order):
    1) Include the number of degrees of freedom per cell for the associated function space. This is an `integer` of kind `i_def` with intent `in`. The name of this argument is `"ndf_"<field_function_space>`;
    2) Include the number of unique degrees of freedom for the associated function space. This is an `integer` of kind `i_def` with intent `in`. The name of this argument is `"undf_"<field_function_space>`;
    3) Include the dofmap for this function space. This is a rank-1 `integer` array of kind `i_def` with extent equal to the number of degrees of freedom of the space (`"ndf_"<field_function_space>`).
5) Include the indirection map for the to-space of the CMA operator. This is a rank-1 `integer` array of kind `i_def` with extent `nrow`.
6) If the from-space of the operator is *not* the same as the to-space then include the indirection map for the from-space of the CMA operator. This is a rank-1 `integer` array of kind `i_def` with extent `ncol`.

Matrix-Matrix

Does not require any dofmaps and also does not require the `nlayers` and `ncell_3d` scalar arguments. The full set of rules are then:

1) Include the `cell` argument. `cell` is an `integer` of kind `i_def` and has intent `in`.
2) Include the total number of cells in the 2D mesh (including halos), `ncell_2d`, which is an `integer` of kind `i_def` with intent `in`.
3) For each CMA operator or scalar argument specified in metadata:
    1) If it is a CMA operator, include it and its associated parameters (see Rule 5 of CMA Assembly kernels);
    2) If it is a scalar argument include the corresponding Fortran variable in the argument list with intent `in`.

Rules for Inter-Grid Kernels

As already specified, inter-grid kernels are only permitted to take fields and/or field-vectors as arguments. Fields (and field-vectors) that are on different meshes must be on different function spaces. Fields on the same mesh must also be on the same function space.

Argument ordering follows the general pattern used for 'normal' kernels with field data being followed by dofmap data. The rules for arguments to inter-grid kernels are as follows:

1) Include `nlayers`, the number of layers in a column. `nlayers` is an `integer` of kind `i_def` and has intent `in`.
2) Include the `cell_map` for the current cell (column). This is an `integer` array of rank two, kind `i_def` and intent `in` which provides the mapping from the coarse to the fine mesh. It has extent `(ncell_f_per_c_x, ncell_f_per_c_y)`.
3) Include `ncell_f_per_c_x`, and `ncell_f_per_c_y`, the numbers of fine cells per coarse cell in the `x` and `y` directions, respectively. These are integers of kind `i_def` and have intent `in`.
4) Include `ncell_f`, the number of cells (columns) in the fine mesh. This is an `integer` of kind `i_def` and has intent `in`.
5) For each argument in the `meta_args` metadata array (which must be a field or field-vector):
    1) Pass in field data as done for a regular kernel.
6) For each unique function space (of which there will currently be two) in the order in which they are encountered in the `meta_args` metadata array, include dofmap information:

    If the dofmap is associated with an argument on the fine mesh:

    1) Include `ndf_fine`, the number of DoFs per cell for the FS of the field on the fine mesh;
    2) Include `undf_fine`, the number of unique DoFs per cell for the FS of the field on the fine mesh;
    3) Include `dofmap_fine`, the *whole* dofmap for the fine mesh. This is an `integer` array of rank two and kind `i_def` with intent `in`. The extent of the first dimension is `ndf_fine` and that of the second is `ncell_f`.

    else, the dofmap is associated with an argument on the coarse mesh:

    1) Include `undf_coarse`, the number of unique DoFs for the coarse field. This is an `integer` of kind `i_def` with intent `in`;
    2) Include `dofmap_coarse`, the dofmap for the current cell (column) in the coarse mesh. This is an `integer` array of rank one, kind `i_def``and has intent ``in`.

Rules for Domain Kernels

The rules for kernels that have `operates_on = DOMAIN` are almost identical to those for general-purpose kernels (described *above*), allowing for the fact that they are not permitted any type of operator argument or any argument with a stencil access. The only difference is that, since the kernel operates on the whole domain, the number of columns in the mesh excluding those in the halo (`ncell_2d_no_halos`), must be passed in. This is provided as the second argument to the kernel (after `nlayers`). `ncell_2d_no_halos` is an `integer` of kind `i_def` with intent `in`.

Rules for DoF Kernels

The rules for kernels that have `operates_on = DOF` are similar to those for general-purpose kernels but, due to the restriction that only fields and scalars can be passed to them, are much fewer. The full set of rules, along with PSyclone's naming conventions, are:

1) Include *df*, the index of the single dof to be operated on. This is an `integer` of of kind `i_def` with intent `in`.
2) For each scalar/field in the order specified by the meta_args metadata:
    1) If the current entry is a scalar quantity then include the Fortran variable in the argument list. The intent is determined from the metadata (see meta_args for an explanation).
    2) If the current entry is a field then include the field array. The field array name is currently specified as being `"field_" <argument_position>`. A field array is a rank-1, real array with extent equal to the number of unique degrees of freedom for the space that the field is on. Its precision (kind) depends on how it is defined in the algorithm layer, see the *Mixed Precision* section for more details. This value is passed in separately. Again, the intent is determined from the metadata (see *meta_args*).

Argument Intents

As described *above*, LFRic kernels read and/or update the data pointed to by objects such as *fields* or *operators*. This data is passed to the kernels as *subroutine arguments* and their Fortran intents usually follow the logic determined by their *access modes*.

- `GH_READ` indicates `intent(in)` as the argument is only ever read from.
- `GH_WRITE` (for discontinuous function spaces) indicates that the argument is only written to in a kernel. The field and operator arguments' data in LFRic are always defined outside of a kernel so the argument intent for this access type is `intent(inout)`.
- `GH_INC`, `GH_READINC` and `GH_READWRITE` indicate `intent(inout)` as the arguments are updated (albeit in a different way due to different access to DoFs, see *meta_args* for more details).

9.5.9 Kernel Naming Conventions

LFRic development uses strict naming conventions related to kernels. While they are not a requirement for PSyclone itself, any LFRic development should follow these conventions (see e.g. *LFRic examples* in PSyclone):

Module name: `<base_name>_kernel_mod`

Kernel type name: `<base_name>_kernel_type`

Subroutine name: `<base_name>_code`

The latest version of the LFRic coding style guidelines are availabe in this LFRic wiki page (requires login access to MOSRS, see the above *introduction* to the LFRic API).

9.6 Built-ins

The basic concept of a PSyclone Built-in is described in the *Built-ins* section. In the LFRic API, calls to Built-ins generally follow a convention that the field/scalar written to comes first in the argument list. LFRic Built-ins must conform to the following rules:

1) They must have one and only one modified (i.e. written to) argument.
2) They must operate on a DoF (`operates_on = DOF` metadata).
3) There must be at least one field in the argument list. This is so that we know the number of DoFs to iterate over in the PSy layer.
4) Kernel arguments must be either fields or scalars (`real`- and/or `integer`-valued).
5) All field arguments to a given Built-in must be on the same function space. This is because all current Built-ins operate on DoFs and therefore all fields should have the same number. It also means that we can determine the number of DoFs uniquely when a scalar is written to;
6) Built-ins that update `real`-valued fields can, in general, only read from other `real`-valued fields, but they can take both `real` and `integer` scalar arguments (see rule 8 for exceptions);
7) Built-ins that update `integer`-valued fields can, in general, only read from other `integer`-valued fields and take `integer` scalar arguments (see rule 8 for exceptions);
8) The only two exceptions from the rules 6) and 7) above regarding the same data type of "write" and "read" field arguments are Built-ins that convert field data from `real` to `integer`, `real_to_int_X`, and from `integer` to `real`, `int_to_real_X`.

The Built-ins supported for the LFRic API are listed in the related subsections, grouped first by the data type of fields they operate on (*real-valued* and *integer-valued*) and then by the mathematical operation they perform.

The field arguments in Built-ins are the derived types that represent the *LFRic fields*, however mathematical operations are actually performed on the data of the *field proxies* (e.g. `field1_proxy%data(:)`). For instance, `X_plus_Y` Built-in adds the values of two fields accessed via their proxies in a loop over DoFs:

```
DO df=loop0_start,loop0_stop
   field3_proxy%data(df) = field1_proxy%data(df) + field2_proxy%data(df)
```

where the precise values of the loop limits depend on the use of *distributed memory*, *annexed DoFs* or both.

As described in the PSy-layer *Argument Intents* section, the Fortran intent of LFRic *field* objects is always `in` (because it is only the data pointed to from within the object that is modified). The field or scalar that has its data modified by a Built-in is marked in **bold**.

For clarity, the calculation performed by each Built-in is described using Fortran array syntax without the details about field proxies. The actual implementation of the Built-in may change in future (*e.g.* it could be implemented by PSyclone generating a call to an optimised Maths library).

9.6.1 Metadata

The code below outlines the elements of the LFRic API Built-in metadata for the Built-ins that update a `real`-valued field, 1) 'meta_args', 2) 'operates_on' and 3) 'procedure':

```
type, public, extends(kernel_type) :: aX_plus_bY
   private
   type(arg_type) :: meta_args(5) = (/                                &
        arg_type(GH_FIELD,  GH_REAL  GH_WRITE, ANY_SPACE_1),          &
        arg_type(GH_SCALAR, GH_REAL, GH_READ               ),         &
        arg_type(GH_FIELD,  GH_REAL, GH_READ,  ANY_SPACE_1),          &
        arg_type(GH_SCALAR, GH_REAL, GH_READ               ),         &
        arg_type(GH_FIELD,  GH_REAL  GH_READ,  ANY_SPACE_1)           &
        /)
   integer :: operates_on = DOF
 contains
   procedure, nopass :: aX_plus_bY_code
end type aX_plus_bY
```

As can be seen, the metadata for a Built-in kernel is a subset of that for a *user-defined Kernel* with the exception that `operates_on` must be `DOF` instead of `CELL_COLUMN`.

The metadata for the LFRic Built-ins that update an `integer`-valued field is similar:

```
!> ifield3 = ifield1 + ifield2
type, public, extends(kernel_type) :: int_X_plus_Y
   private
   type(arg_type) :: meta_args(3) = (/                                &
        arg_type(GH_FIELD, GH_INTEGER, GH_WRITE, ANY_SPACE_1),        &
        arg_type(GH_FIELD, GH_INTEGER, GH_READ,  ANY_SPACE_1),        &
        arg_type(GH_FIELD, GH_INTEGER, GH_READ,  ANY_SPACE_1)         &
        /)
   integer :: operates_on = DOF
 contains
   procedure, nopass :: int_X_plus_Y_code
end type int_X_plus_Y
```

Valid Data Types and Access Modes

The allowed data types and accesses for arguments in LFRic Built-in kernels are a bit different than for the *user-defined Kernels* and are listed in the table below.

Argument Type	Data Type	Function Space	Access Type
GH_SCALAR	GH_INTEGER	n/a	GH_READ
GH_SCALAR	GH_REAL	n/a	GH_READ, GH_SUM
GH_FIELD	GH_REAL, GH_INTEGER	ANY_SPACE_<n>	GH_READ, GH_WRITE, GH_READWRITE

Note: Since the LFRic infrastructure does not currently support `integer` reductions, `integer` scalar arguments in Built-ins are restricted to having read-only access. Also, `logical` scalar arguments are not permitted.

9.6.2 Naming scheme

The supported Built-ins in the LFRic API are named according to the scheme presented below. Any new Built-in needs to comply with these rules.

1) Ordering of arguments in Built-ins calls follows *LHS (result) <- RHS (operation on arguments)* direction, except where a Built-in returns the *LHS* result to one of the *RHS* arguments. In that case ordering of arguments remains as in the *RHS* expression, with the returning *RHS* argument written as close to the *LHS* as it can be without affecting the mathematical expression.
2) Field names begin with upper case in short form (e.g. **X**, **Y**, **Z**) and any case in long form (e.g. **Field1**, **field**).
3) Scalar names begin with lower case: e.g. **a**, **b**, are **scalar1**, **scalar2**. Special names for scalars are: **constant** (or **c**), **innprod** (inner/scalar product of two fields) and **sumfld** (sum of a field).
4) Arguments in Built-ins variable declarations and constructs (PSyclone Fortran and Python definitions):
    1) Are always written in long form and lower case (e.g. **field1**, **field2**, **scalar1**, **scalar2**);
    2) *LHS* result arguments are always listed first;
    3) *RHS* arguments are listed in order of appearance in the mathematical expression, except when one of them is the *LHS* result.
5) Built-ins names in Fortran consist of:
    1) *RHS* arguments in short form (e.g. **X**, **Y**, **a**, **b**) only;
    2) Descriptive name of mathematical operation on *RHS* arguments in the form `<operationname>_<RHSargs>` or `<RHSargs>_<operationname>_<RHSargs>`;
    3) Prefix `"inc_"` where the result is returned to one of the *RHS* arguments (i.e. `"inc_"<RHSargs>_<operationname>_<RHSargs>`);
    4) Prefix `"int_"` for the Built-in operations on the `integer`-valued field arguments (i.e. `"int_inc_"<RHSargs>_<operationname>_<RHSargs>`).
6) Built-ins names in Python definitions are similar to their Fortran counterparts, with a few differences:
    1) Operators and *RHS* arguments are all in upper case (e.g. **X**, **Y**, **A**, **B**, **Plus**, **Minus**);
    2) There are no underscores;
    4) Common suffix is `"Kern"`;
    3) Common prefix is `"LFRic"` for the Built-in operations on the `real`-valued arguments and `"LFRicInt"` for the Built-in operations on the `integer`-valued fields.

9.6.3 Built-in operations on `real`-valued fields

As described *above*, Built-ins that operate on `real`-valued fields mandate `GH_REAL` as the kernel metadata for fields and scalars.

The precision of fields and scalars, however, is determined by the algorithm layer via precision variables as described in the *Mixed Precision* section (see subsections on *fields* and *scalars*).

For instance, field and scalar declarations for the `aX_plus_Y` Built-in that operates on `r_solver_field_type` and uses `r_solver` scalar will be:

```
real(kind=r_solver), intent(in) :: ascalar
type(r_solver_field_type), intent(in) :: zfield, xfield, yfield
```

Mixing precisions is not explicitly forbidden, so we may have e.g. `X_divideby_a` Built-in where:

```
real(kind=r_def), intent(in) :: ascalar
type(r_tran_field_type), intent(in) :: yfield, xfield
```

Certain Built-ins are currently restricted in the precision of the arguments that they accept. Those that calculate the inner product and sum of a field are restricted to `r_def` precision because the scalar global reductions in the LFRic infrastructure are currently only able to support `field_type` and hence have `r_def` precision. In addition, all integer arguments to Built-ins are currently restricted to `i_def` precision.

Addition

Built-ins that add (scaled) `real`-valued fields and return the result as a `real`-valued field are denoted with the keyword **plus**.

X_plus_Y

X_plus_Y (**field3**, *field1*, *field2*)

Sums two fields and stores the result in the third field (`Z = X + Y`):

```
field3(:) = field1(:) + field2(:)
```

inc_X_plus_Y

inc_X_plus_Y (**field1**, *field2*)

Adds the second field to the first and returns it (`X = X + Y`):

```
field1(:) = field1(:) + field2(:)
```

a_plus_X

a_plus_X (**field2**, *rscalar*, *field1*)

Adds a `real` scalar value to all elements of a field and stores the result in another field (`Y = a + X`):

```
field2(:) = rscalar + field1(:)
```

inc_a_plus_X

inc_a_plus_X (*rscalar*, **field**)

Adds a `real` scalar value to all elements of a field and returns the field (`X = a + X`):

```
field(:) = rscalar + field(:)
```

aX_plus_Y

aX_plus_Y (**field3**, *rscalar*, *field1*, *field2*)

Performs `Z = aX + Y`:

```
field3(:) = rscalar*field1(:) + field2(:)
```

inc_aX_plus_Y

inc_aX_plus_Y (*rscalar*, **field1**, *field2*)

Performs `X = aX + Y` (increments the first field):

```
field1(:) = rscalar*field1(:) + field2(:)
```

inc_X_plus_bY

inc_X_plus_bY (**field1**, *rscalar*, *field2*)

Performs `X = X + bY` (increments the first field):

```
field1(:) = field1(:) + rscalar*field2(:)
```

aX_plus_bY

aX_plus_bY (**field3**, *rscalar1*, *field1*, *rscalar2*, *field2*)

Performs `Z = aX + bY`:

```
field3(:) = rscalar1*field1(:) + rscalar2*field2(:)
```

inc_aX_plus_bY

inc_aX_plus_bY (*rscalar1*, **field1**, *rscalar2*, *field2*)

Performs `X = aX + bY` (increments the first field):

```
field1(:) = rscalar1*field1(:) + rscalar2*field2(:)
```

aX_plus_aY

aX_plus_aY (**field3**, *rscalar*, *field1*, *field2*)

Performs `Z = aX + aY = a(X + Y)`:

```
field3(:) = rscalar*(field1(:) + field2(:))
```

Subtraction

Built-ins which subtract (scaled) `real`-valued fields and return the result as a `real`-valued field are denoted with the keyword **minus**.

X_minus_Y

X_minus_Y (**field3**, *field1*, *field2*)

Subtracts the second field from the first and returns the result in the third field (`Z = X - Y`):

```
field3(:) = field1(:) - field2(:)
```

inc_X_minus_Y

inc_X_minus_Y (**field1**, *field2*)

Subtracts the second field from the first and returns it (`X = X - Y`):

```
field1(:) = field1(:) - field2(:)
```

a_minus_X

a_minus_X (**field2**, *rscalar*, *field1*)

Subtracts all elements of a field from a `real` scalar value and stores the result in another field (`Y = a - X`):

```
field2(:) = rscalar - field1(:)
```

inc_a_minus_X

inc_a_minus_X (*rscalar*, **field**)

Subtracts all elements of a field from a `real` scalar value and returns the field (`X = a - X`):

```
field(:) = rscalar - field(:)
```

X_minus_a

X_minus_a (**field2**, *field1*, *rscalar*)

Subtracts a `real` scalar value from all elements of a field and stores the result in another field (`Y = X - a`):

```
field2(:) = field1(:) - rscalar
```

inc_X_minus_a

inc_X_minus_a (**field**, *rscalar*)

Subtracts a `real` scalar value from all elements of a field and returns the field (`X = X - a`):

```
field(:) = field(:) - rscalar
```

aX_minus_Y

aX_minus_Y (**field3**, *rscalar*, *field1*, *field2*)

Performs `Z = aX - Y`:

```
field3(:) = rscalar*field1(:) - field2(:)
```

X_minus_bY

X_minus_bY (**field3**, *field1*, *rscalar*, *field2*)

Performs `Z = X - bY`:

```
field3(:) = field1(:) - rscalar*field2(:)
```

inc_X_minus_bY

inc_X_minus_bY (**field1**, *rscalar*, *field2*)

Performs `X = X - bY` (decrements the first field):

```
field1(:) = field1(:) - rscalar*field2(:)
```

aX_minus_bY

aX_minus_bY (**field3**, *rscalar1*, *field1*, *rscalar2*, *field2*)

Performs `Z = aX - bY`:

```
field3(:) = rscalar1*field1(:) - rscalar2*field2(:)
```

Multiplication

Built-ins which multiply (scaled) `real`-valued fields and return the result as a `real`-valued field are denoted with the keyword **times**.

X_times_Y

X_times_Y (**field3**, *field1*, *field2*)

Multiplies two fields DoF by DoF and returns the result in a third field (`Z = X*Y`):

```
field3(:) = field1(:)*field2(:)
```

inc_X_times_Y

inc_X_times_Y (**field1**, *field2*)

Multiplies the first field by the second and returns it (`X = X*Y`):

```
field1(:) = field1(:)*field2(:)
```

inc_aX_times_Y

inc_aX_times_Y (*rscalar*, **field1**, *field2*)

Performs `X = a*X*Y` (increments the first field):

```
field1(:) = rscalar*field1(:)*field2(:)
```

Scaling

Built-ins which scale `real`-valued fields are technically cases of multiplying a `real`-valued field by a `real` scalar and are hence also denoted with the keyword **times**.

a_times_X

a_times_X (**field2**, *rscalar*, *field1*)

Multiplies a field by a `real` scalar value and stores the result in another field (`Y = a*X`):

```
field2(:) = rscalar*field1(:)
```

inc_a_times_X

inc_a_times_X (*rscalar*, **field**)

Multiplies a field by a `real` scalar value and returns the field (`X = a*X`):

```
field(:) = rscalar*field(:)
```

Division

Built-ins which divide `real`-valued fields and return the result as a `real`-valued field are denoted with the keyword **divideby**.

X_divideby_Y

X_divideby_Y (**field3**, *field1*, *field2*)

Divides the first field by the second field, DoF by DoF, and stores the result in the third field (`Z = X/Y`):

```
field3(:) = field1(:)/field2(:)
```

inc_X_divideby_Y

inc_X_divideby_Y (**field1**, *field2*)

Divides the first field by the second and returns it (`X = X/Y`):

```
field1(:) = field1(:)/field2(:)
```

X_divideby_a

X_divideby_a (**field2**, *field1*, *rscalar*)

Divides each field element by a `real` scalar value and stores the result in another field (`Y = X/a`):

```
field2(:) = field1(:)/rscalar
```

inc_X_divideby_a

inc_X_divideby_a (**field**, *rscalar*)

Divides each field element by a `real` scalar value and returns the field (`X = X/a`):

```
field(:) = field(:)/rscalar
```

Inverse scaling

Built-ins which perform inverse scaling of `real`-valued fields are also denoted with the keyword **divideby** as they divide a `real` scalar by elements of a `real`-valued field.

a_divideby_X

a_divideby_X (**field2**, *rscalar*, *field1*)

Divides a `real` scalar value by each field element and stores the result in another field (`Y = a/X`):

```
field2(:) = rscalar/field1(:)
```

inc_a_divideby_X

inc_a_divideby_X (*rscalar*, **field**)

Divides a `real` scalar value by each field element and returns the field (`X = a/X`):

```
field(:) = rscalar/field(:)
```

Setting to a value

Built-ins which set `real`-valued field elements to some `real` value are denoted with the keyword **setval**.

setval_c

setval_c (**field**, *constant*)

Sets all elements of a field *field* to a `real` scalar *constant* (`X = c`):

```
field(:) = constant
```

setval_X

setval_X (**field2**, *field1*)

Sets a field *field2* equal (DoF per DoF) to another field *field1* (`Y = X`):

```
field2(:) = field1(:)
```

setval_random

setval_random (**field**)

Fills all elements of a field *field* using a sequence of `real`, pseudo-random numbers in the interval `0 <= x < 1`:

```
do df = 1, ndofs
  field(df) = RAND()
end do
```

where `RAND()` is some function that returns a new pseudo-random number each time it is called.

Warning: This Built-in is implemented using the Fortran `random_number` intrinsic. Therefore no guarantee is made as to the quality of the sequence of pseudo-random numbers, especially when running in parallel.

Raising to power

Built-ins which raise `real`-valued field elements to an exponent are denoted with the keyword **powreal** for a `real` exponent or **powint** for an `integer` exponent.

inc_X_powreal_a

inc_X_powreal_a (**field**, *rscalar*)

Raises a field to a `real` scalar value and returns the field (`X = X**a`):

```
field(:) = field(:)**rscalar
```

inc_X_powint_n

inc_X_powint_n (**field**, *iscalar*)

Raises a field to an `integer` scalar value and returns the field (`X = X**n`):

```
field(:) = field(:)**iscalar
```

where `iscalar` is an `integer` scalar of `i_def` precision.

Inner product

Built-ins which calculate the inner product of two `real`-valued fields or of a `real`-valued field with itself and return the result as a `real` scalar are denoted with the keyword **innerproduct**.

Note: When used with distributed memory these Built-ins will trigger the addition of a global sum which may affect the performance and/or scalability of the code. Also, whilst the fields in these Built-ins can be of any supported `real` *precision*, the only currently supported precision for the global reductions in the LFRic infrastructure is `r_def`, hence the result will be converted accordingly.

X_innerproduct_Y

X_innerproduct_Y (**innprod**, *field1*, *field2*)

Computes the inner product of two fields, *field1* and *field2*, *i.e.*:

```
innprod = SUM(field1(:)*field2(:))
```

where **innprod** is a `real` scalar of `r_def` precision.

X_innerproduct_X

X_innerproduct_X (**innprod**, *field*)

Computes the inner product of the field *field1* by itself, *i.e.*:

```
innprod = SUM(field(:)*field(:))
```

where **innprod** is a `real` scalar of `r_def` precision.

Sum of elements

A Built-in which sums the elements of a `real`-valued field and returns the result as a `real` scalar is denoted with the keyword **sum**.

Note: When used with distributed memory this Built-in will trigger the addition of a global sum which may affect the performance and/or scalability of the code. Also, whilst the fields in these Built-ins can be of any supported `real` *precision*, the only currently supported precision for the global reductions in the LFRic infrastructure is `r_def`, hence the result will be converted accordingly.

---

sum_X

sum_X (**sumfld**, *field*)

Sums all of the elements of the field *field* and returns the result in the `real` scalar variable *sumfld*:

```
sumfld = SUM(field(:))
```

where **sumfld** is a `real` scalar of `r_def` precision.

Sign of elements

A Built-in which returns the sign of a `real`-valued field is denoted with the keyword **sign**.

sign_X

sign_X (**field2**, *rscalar*, *field1*)

Returns the sign of a `real`-valued field, e.g. in Fortran: `Y = sign(a, X)`. Here `a` is a `real` scalar and `Y` and `X` are `real`-valued fields. The results are `a` for `X >= 0` and `-a` for `X < 0`:

```
field2(:) = SIGN(rscalar, field1(:))
```

DoF-wise maximum of elements

Built-ins which return the DoF-wise maximum of a `real` scalar and a `real`-valued field are denoted with the keyword **max**.

max_aX

max_aX (**field2**, *rscalar*, *field1*)

Returns maximum of *rscalar* and each element of the field *field1* as the second field **field2** (`Y = max(a, X)`):

```
field2(:) = MAX(rscalar, field1(:))
```

inc_max_aX

inc_max_aX (*rscalar*, **field**)

Returns maximum of *rscalar* and each element of the field *field* in the same field (`X = max(a, X)`):

```
field(:) = MAX(rscalar, field(:))
```

DoF-wise minimum of elements

Built-ins which return the DoF-wise minimum of a `real` scalar and a `real`-valued field are denoted with the keyword **min**.

min_aX

min_aX (**field2**, *rscalar*, *field1*)

Returns minimum of *rscalar* and each element of the field *field1* as the second field **field2** (`Y = min(a, X)`):

```
field2(:) = MIN(rscalar, field1(:))
```

inc_min_aX

inc_min_aX (*rscalar*, **field**)

Returns minimum of *rscalar* and each element of the field *field* in the same field (`X = min(a, X)`):

```
field(:) = MIN(rscalar, field(:))
```

Conversion of `real` field elements

Built-ins which take a `real` field for conversion to a field of a different datatype or precision are denoted by the datatype that the input `real` field will be converted to. A Built-in that converts a `real` to an `integer` field is denoted by the phrase **to_int**. Likewise, a Built-in that converts a `real` to a `real` field is denoted by the phrase **to_real**.

real_to_int_X

real_to_int_X (**ifield2**, *field1*)

Converts `real`-valued field elements to `integer`-valued field elements, e.g. in Fortran this would be: `Y = INT(X, kind=i_<prec>)`. Here `Y` is an `integer`-valued field and `X` is the `real`-valued field being converted:

```
ifield2(:) = INT(field1(:), kind=i_<prec>)
```

where **ifield2** is currently the only supported `integer`-valued field type in LFRic (`integer_field_type` of `i_def` precision) and a `real` -valued field *field1* can be of any *supported precisions* for `GH_REAL` fields (e.g. `r_tran` for `r_tran_field_type`).

real_to_real_X

real_to_real_X (**field2**, *field1*)

Converts `real`-valued field elements from a precision `r_<prec>` to `real`-valued field elements of a differing precision `r_<prec>`, e.g. in Fortran this would be: `Y = REAL(X, kind=r_<prec>)`. Here `Y` and `X` are both `real`-valued fields, with `X` being converted to the precision of `Y`:

```
field2(:) = REAL(field1(:), kind=r_<prec>)
```

field2 and *field1* are `real`-valued fields of any *supported precisions* for `GH_REAL` fields (e.g. `r_tran` for `r_tran_field_type`).

9.6.4 Built-in operations on `integer`-valued fields

The number of supported Built-in operations on the `integer`-valued fields is not as large as for their `real` counterparts as not all mathematical operations on `integer`-valued fields make sense.

As described *above*, Built-ins that operate on `integer`-valued fields mandate `GH_INTEGER` as the kernel metadata for fields and scalars. Both `integer` scalar arguments and `integer`-valued fields can only currently have `i_def` precision, as described in the *Mixed Precision* section.

For instance, field and scalar declarations for the `X_minus_a` Built-in will be:

```
integer(kind=i_def), intent(in) :: ascalar
type(integer_field_type), intent(in) :: yfield, xfield
```

Addition

Built-ins that add `integer`-valued fields and return the result as an `integer`-valued field are denoted with the keyword **plus** and the prefix **int**.

int_X_plus_Y

int_X_plus_Y (**ifield3**, *ifield1*, *ifield2*)

Sums two fields and stores the result in the third field (`Z = X + Y`):

```
ifield3(:) = ifield1(:) + ifield2(:)
```

int_inc_X_plus_Y

int_inc_X_plus_Y (**ifield1**, *ifield2*)

Adds the second field to the first and returns it (`X = X + Y`):

```
ifield1(:) = ifield1(:) + ifield2(:)
```

int_a_plus_X

int_a_plus_X (**ifield2**, *iscalar*, *ifield1*)

Adds an `integer` scalar value to all elements of a field and stores the result in another field (`Y = a + X`):

```
ifield2(:) = iscalar + ifield1(:)
```

int_inc_a_plus_X

int_inc_a_plus_X (*iscalar*, **ifield**)

Adds an `integer` scalar value to all elements of a field and returns the field (`X = a + X`):

```
ifield(:) = iscalar + ifield(:)
```

Subtraction

Built-ins which subtract `integer`-valued fields and return the result as an `integer`-valued field are denoted with the keyword **minus** and the prefix **int**.

int_X_minus_Y

int_X_minus_Y (**ifield3**, *ifield1*, *ifield2*)

Subtracts the second field from the first and returns the result in the third field (`Z = X - Y`):

```
ifield3(:) = ifield1(:) - ifield2(:)
```

int_inc_X_minus_Y

int_inc_X_minus_Y (**ifield1**, *ifield2*)

Subtracts the second field from the first and returns it (`X = X - Y`):

```
ifield1(:) = ifield1(:) - ifield2(:)
```

int_a_minus_X

int_a_minus_X (**ifield2**, *iscalar*, *ifield1*)

Subtracts all elements of a field from an `integer` scalar value and stores the result in another field (`Y = a - X`):

```
ifield2(:) = iscalar - ifield1(:)
```

int_inc_a_minus_X

int_inc_a_minus_X (*iscalar*, **ifield**)

Subtracts all elements of a field from an `integer` scalar value and returns the field (`X = a - X`):

```
ifield(:) = iscalar - ifield(:)
```

int_X_minus_a

int_X_minus_a (**ifield2**, *ifield1*, *iscalar*)

Subtracts an `integer` scalar value from all elements of a field and stores the result in another field (`Y = X - a`):

```
ifield2(:) =  ifield1(:) - iscalar
```

int_inc_X_minus_a

int_inc_X_minus_a (**ifield**, *iscalar*)

Subtracts an `integer` scalar value from all elements of a field and returns the field (`X = X - a`):

```
ifield(:) =  ifield(:) - iscalar
```

Multiplication

Built-ins which multiply `integer`-valued fields and return the result as an `integer`-valued field are denoted with the keyword **times** and the prefix **int**.

int_X_times_Y

int_X_times_Y (**ifield3**, *ifield1*, *ifield2*)

Multiplies two fields DoF by DoF and returns the result in a third field (`Z = X*Y`):

```
ifield3(:) = ifield1(:)*ifield2(:)
```

int_inc_X_times_Y

int_inc_X_times_Y (**ifield1**, *ifield2*)

Multiplies the first field by the second and returns it (`X = X*Y`):

```
ifield1(:) = ifield1(:)*ifield2(:)
```

Scaling

Built-ins which scale `integer`-valued fields are denoted with the keyword **times** and prefixed by the keyword **int**.

int_a_times_X

int_a_times_X (**ifield2**, *iscalar*, *ifield1*)

Multiplies a field by an `integer` scalar and stores the result in another field (`Y = a*X`):

```
ifield2(:) = iscalar*ifield1(:)
```

int_inc_a_times_X

int_inc_a_times_X (*iscalar*, **ifield**)

Multiplies a field by an `integer` scalar value and returns the field (`X = a*X`):

```
ifield(:) = iscalar*ifield(:)
```

Setting to a value

Built-ins which set `integer`-valued field elements to some `integer` value are denoted with the keyword **setval** and the prefix **int**.

int_setval_c

int_setval_c (**ifield**, *constant*)

Sets all elements of a field *ifield* to an `integer` scalar *constant* (`X = c`):

```
ifield(:) = constant
```

int_setval_X

int_setval_X (**ifield2**, *ifield1*)

Sets a field *ifield2* equal (DoF per DoF) to another field *ifield1* (`Y = X`):

```
ifield2(:) = ifield1(:)
```

Sign of elements

A Built-in which returns the sign of an `integer`-valued field is denoted with the keyword **sign** and the prefix **int**.

int_sign_X

int_sign_X (**ifield2**, *iscalar*, *ifield1*)

Returns the sign of an `integer`-valued field, e.g. in Fortran: `Y = sign(a, X)`. Here `a` is an `integer` scalar and `Y` and `X` are `integer`-valued fields. The results are `a` for `X >= 0` and `-a` for `a < 0`:

```
ifield2(:) = SIGN(iscalar, ifield1(:))
```

DoF-wise maximum of elements

Built-ins which return the DoF-wise maximum of an `integer` scalar and an `integer`-valued field are denoted with the keyword **max**.

int_max_aX

int_max_aX (**ifield2**, *iscalar*, *ifield1*)

Returns maximum of *iscalar* and each element of the field *ifield1* as the second field **ifield2** (`Y = max(a, X)`):

```
ifield2(:) = MAX(iscalar, ifield1(:))
```

int_inc_max_aX

int_inc_max_aX (*iscalar*, **ifield**)

Returns maximum of *iscalar* and each element of the field *ifield* in the same field (`X = max(a, X)`):

```
ifield(:) = MAX(iscalar, ifield(:))
```

DoF-wise minimum of elements

Built-ins which return the DoF-wise minimum of an `integer` scalar and an `integer`-valued field are denoted with the keyword **min**.

int_min_aX

int_min_aX (**ifield2**, *iscalar*, *ifield1*)

Returns minimum of *iscalar* and each element of the field *ifield1* as the second field **ifield2** (`Y = min(a, X)`):

```
ifield2(:) = MIN(iscalar, ifield1(:))
```

int_inc_min_aX

int_inc_min_aX (*iscalar*, **ifield**)

Returns minimum of *iscalar* and each element of the field *ifield* in the same field (`X = min(a, X)`):

```
ifield(:) = MIN(iscalar, ifield(:))
```

Conversion of `integer` to `real` field elements

A Built-in which takes an `integer` field and converts it to a `real` field is denoted by the phrase **to_real**.

int_to_real_X

int_to_real_X (**field2**, *ifield1*)

Converts `integer`-valued field elements to `real`-valued field elements, e.g. in Fortran this would be `Y = REAL(X, kind=r_<prec>)`. Here `Y` is a `real`-valued field and `X` is the `integer`-valued field being converted:

```
field2(:) = REAL(ifield1(:), kind=r_<prec>)
```

where **ifield1** is currently the only supported `integer`-valued field type in LFRic (`integer_field_type` of `i_def` precision). The `real`-valued **field1** can be of any *supported precisions* for `GH_REAL` fields, hence `r_<prec>` is determined from the algorithm layer (e.g. `r_solver` for `r_solver_field_type`).

9.7 Boundary Conditions

In the LFRic API, boundary conditions for a field or LMA operator can be enforced by the algorithm developer by calling the Kernels `enforce_bc_type` or `enforce_operator_bc_type`, respectively. These kernels take a field or operator as input and apply boundary conditions. For example:

```
call invoke( kernel_type(field1, field2),      &
             enforce_bc_type(field1),          &
             kernel_with_op_type(field1, op1), &
             enforce_operator_bc_type(op1)     &
           )
```

The particular boundary conditions that are applied are not known by PSyclone, PSyclone simply recognises these kernels by their names and passes pre-specified dofmap and boundary_value arrays into the kernel implementations, the contents of which are set by the LFRic infrastructure.

Up to and including version 1.4.0 of PSyclone, boundary conditions were applied automatically after a call to `matrix_vector_type` if the field arguments were on a vector function space (one of `W1`, `W2`, `W2H`, `W2V` or `W2broken`). With the subsequent introduction of the ability to apply boundary conditions to operators this functionality is no longer required and has been removed.

Example `eg4` in the `examples/lfric` directory includes a call to `enforce_bc_kernel_type` so can be used to see the boundary condition code that is added by PSyclone. See the `README` in the `examples/lfric` directory for instructions on how to run this example.

An example of applying boundary conditions to an operator is the kernel `enforce_operator_bc_kernel_mod.F90` in the `<PSYCLONEHOME>/src/psyclone/tests/test_files/dynamo0p3` directory. Since operators are discontinuous quantities, updating their values can be safely performed in parallel (see Section *Kernel*). The `GH_READWRITE` access is used for updating discontinuous operators (see subsection *Valid Access Modes* for more details).

9.8 Conventions

The naming of LFRic API kernels and associated entities (types, subroutines and modules) follows the PSyclone Fortran naming conventions (see *Fortran Naming Conventions*). However, PSyclone does not need this convention to be followed apart from the stub generator (see the *Kernel-stub Generator* Section) where the name of the metadata to be parsed is determined from the module name.

The contents of the metadata is also usually declared private but this does not affect PSyclone.

Finally, the `procedure` metadata (located within the kernel metadata) usually has `nopass` specified but again this is ignored by PSyclone.

9.9 Configuration

The general and the LFRic-API-specific configuration options are described in the *Configuration* section.

9.9.1 Annexed DoFs

When a kernel operates on DoFs (rather than cell-columns) for a continuous field using distributed memory, PSyclone need only ensure that DoFs owned by a processor are computed. However, for continuous fields, shared DoFs at the boundary between processors must be replicated (as different cells share the same DoF). Only one processor can own a DoF, therefore processors will have continuous fields which contain DoFs that the processor does not own. These unowned DoFs are called *annexed* in the LFRic API and are a separate, but related, concept to field halos.

When a kernel that operates on a cell-column needs to read a continuous field then the annexed DoFs must be up-to-date on all processors. If they are not then a halo exchange must be added. Currently PSyclone defaults, for kernels which iterate over DoFs, to iterating over only owned DoFs. This behaviour can be changed by setting *COMPUTE_ANNEXED_DOFS* to `true` in the *lfric* section of the configuration file (see the *Configuration* section). PSyclone will then generate code to iterate over both owned and annexed DoFs, thereby reducing the number of halo exchanges required (at the expense of redundantly computing annexed DoFs). For more details please refer to the LFRic developers section.

9.9.2 Run-time Checks

PSyclone performs static consistency checks where possible. When this is not possible PSyclone can generate run-time checks. As there may be performance costs associated with run-time checks they may be switched on or off by the *RUN_TIME_CHECKS* option in the configuration file.

Currently run-time checks can be generated to:

1) Check that a field with a read-only function space (see section *Read-Only Function Spaces*) is not modified by a kernel. This is enforced by checking that all fields that are marked (in kernel metadata) as being updated by a kernel are not on a read-only function space. A second check that is required for fields on read-only function spaces is to ensure that the halo is clean before it is accessed. This check is currently implemented within the LFRic infrastructure halo exchange call (that the PSyclone LFRic API places at appropriate locations). If the halo is clean then the halo exchange will not be called. However, if the halo is not clean then the resulting halo exchange call will cause the infrastructure to raise an error (because the field is on a read-only space).

2) Check that the function space of a field is consistent with the kernel function space metadata that the field's data is passed into. For example, if kernel metadata specifies that a field is on the `W2` function space then a run-time check is added to ensure that the field object passed into the PSy layer is indeed on that space. For more general kernel function space metadata, such as *ANY_DISCONTINUOUS_SPACE_** then a run-time check is added to ensure that the field is on one of the discontinuous function spaces supported in the LFRic API.

9.9.3 Supported Data Types and Default Kind

The LFRic API supports three Fortran primitive (intrinsic) data types, `real`, `integer` and `logical` (listed in the *supported_fortran_datatypes* section of the *PSyclone configuration file*). All three data types are used for *scalars*. *Fields* and *field vectors* are allowed to have `real` and `integer` data. *Operators* and *column-wise operators* are only allowed to have `real` data. These supported primitive types are linked to the respective *kernel data type* metadata descriptors, `GH_REAL` and `GH_INTEGER`.

The default kind (precision) for these supported data types is set to `r_def`, `i_def` and `l_def`, respectively, in the `default_kind` dictionary in the configuration file. These default values are defined in the LFRic infrastructure code.

Note: Whilst the `logical` Fortran primitive (intrinsic) data type is supported in the LFRic API for scalar arguments, it is not yet available for fields and operators. This will be added as required in future releases.

9.9.4 Precision Map

This gives the amount of storage (in bytes) associated with a particular LFRic precision. The values for 'r_tran', 'r_solver', 'r_def', 'r_bl' and 'r_phys' are set within LFRic infrastructure according to CPP ifdefs. The values given in the configuration file are the defaults. 'l_def' is included in the dictionary so that it contains a complete record of the various precision symbols used in LFRic.

Note: Storing the precision map in the LFRic API within PSyclone is a temporary measure which will yield to the LFRic infrastructure as the single source of precisions, as discussed in PSyclone issue #1941.

9.9.5 Number of Generalised `ANY_*_SPACE` Function Spaces

As outlined in the *meta_args* and the *Supported Function Spaces* sections above, the number of generalised `ANY_SPACE_<n>` and `ANY_DISCONTINUOUS_SPACE_<n>` function spaces can be set in the *PSyclone configuration file*.

The relevant parameters are `NUM_ANY_SPACE` and `NUM_ANY_DISCONTINUOUS_SPACE`, respectively. Their default values in the configuration file are 10 and their allowed values are positive non-zero integers. PSyclone will raise a `ConfigurationError` if a supplied value is invalid.

9.10 Transformations

This section describes the LFRic API-specific transformations. In cases, excepting **Dynamo0p3RedundantComputationTrans**, **Dynamo0p3AsyncHaloExchangeTrans** and **Dynamo0p3KernelConstTrans**, these transformations are specialisations of generic transformations described in the *Transformations* section. The difference between these transformations and the generic ones is that these perform LFRic API-specific checks to make sure the transformations are valid. In practice these transformations perform the required checks then call the generic ones internally.

The use of the LFRic API-specific transformations is exactly the same as the equivalent generic ones in all cases excepting **LFRicLoopFuseTrans**. In this case an additional optional argument **same_space** can be set when applying the transformation. The reason for this is to allow loop fusion when one or more of the iteration spaces is determined by a function space that is unknown by PSyclone at compile time. This is the case when the `ANY_SPACE_<n>` function space is specified in the Kernel metadata. Adding `{"same_space": True}` as option when applying the transformation allows the user to specify that the spaces are the same (see *Standard Functionality* for using options in transformations). This option should therefore be used with caution. PSyclone will raise an error if **same_space** is used when at least one of the function spaces is not `ANY_SPACE_<n>` or both spaces are not the same. In general, PSyclone will not allow loop fusion if it does not know the spaces are the same. The exception are loops over discontinuous spaces (see *Supported Function Spaces* for list of discontinuous function spaces) for which loop fusion is allowed (unless the loop bounds become different due to a prior transformation).

The **Dynamo0p3RedundantComputationTrans** and **Dynamo0p3AsyncHaloExchange** transformations are only valid for the LFRic API. This is because this API is currently the only one that supports distributed memory. An example of redundant computation can be found in `examples/lfric/eg8` and an example of asynchronous halo exchanges can be found in `examples/lfric/eg11`.

The **Dynamo0p3KernelConstTrans** transformation is only valid for the LFRic API. This is because the properties that it makes constant are API specific.

The LFRic API-specific transformations currently available are given below. Early transformations include "Dynamo0p3" or "Dynamo" in their name to indicate that these transformations are only valid for this particular API. More

recent transformations typically include "LFRic" in their name to indicate the same restriction. However, more importantly, transformations that are specific to LFRic reside in the LFRic-specific "psyclone.domain/lfric/transformations" directory. Note, the early LFRic API-specific transformations have not yet been migrated to this directory.

Note: Only the loop-colouring and OpenMP transformations are currently supported for loops that contain inter-grid kernels. Attempting to apply other transformation types will result in PSyclone raising an error.

---

class psyclone.domain.lfric.transformations.**LFRicExtractTrans**

LFRic API application of ExtractTrans transformation to extract code into a stand-alone program. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>>
>>> API = "lfric"
>>> FILENAME = "solver_alg.x90"
>>> ast, invokeInfo = parse(FILENAME, api=API)
>>> psy = PSyFactory(API, distributed_memory=False).create(invoke_info)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>>
>>> from psyclone.domain.lfric.transformations import LFRicExtractTrans
>>> etrans = LFRicExtractTrans()
>>>
>>> # Apply LFRicExtractTrans transformation to selected Nodes
>>> etrans.apply(schedule.children[0:3])
>>> print(schedule.view())
```

apply(*nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single PSyData region. It first uses the CallTreeUtils to determine input- and output-parameters. If requested, it will then call the LFRicExtractDriverCreator to write the stand-alone driver program. Then it will call apply of the base class.

Parameters

- **nodes** (psyclone.psyir.nodes.Node or List[psyclone.psyir.nodes.Node]) – can be a single node or a list of nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (prefix_psy_data_mod) and the PSyDataType (prefix_PSyDataType) - a "_" will be added automatically. It defaults to "extract", resulting in e.g. extract_psy_data_mod.
- **options["create_driver"]** (*bool*) – whether or not to create a driver program at code-generation time. If set, the driver will be created in the current working directory with the name "driver-MODULE-REGION.f90" where MODULE and REGION will be the corresponding values for this region. Defaults to False.
- **options["region_name"]** (*Tuple[str,str]*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

validate(*node_list*, *options=None*)

Perform Dynamo0.3 API specific validation checks before applying the transformation.

Parameters

- **node_list** (List[psyclone.psyir.nodes.Node]) – the list of Node(s) we are checking.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises **TransformationError** – if transformation is applied to a Loop over cells in a colour without its parent Loop over colours.

class psyclone.domain.lfric.transformations.**LFRicLoopFuseTrans**

LFRic API specialisation of the `base class` in order to fuse two Dynamo loops after performing validity checks. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>>
>>> API = "lfric"
>>> FILENAME = "alg.x90"
>>> ast, invokeInfo = parse(FILENAME, api=API)
>>> psy = PSyFactory(API, distributed_memory=False).create(invoke_info)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>>
>>> from psyclone.domain.lfric.transformations import LFRicLoopFuseTrans
>>> ftrans =  LFRicLoopFuseTrans()
>>>
>>> ftrans.apply(schedule[0], schedule[1])
>>> print(schedule.view())
```

The optional argument *same_space* can be set as

```
>>> ftrans.apply(schedule[0], schedule[1], {"same_space": True})
```

when applying the transformation.

validate(*node1*, *node2*, *options=None*)

Performs various checks to ensure that it is valid to apply the LFRicLoopFuseTrans transformation to the supplied loops.

Parameters

- **node1** (psyclone.domain.lfric.LFRicLoop) – the first Loop to fuse.
- **node2** (psyclone.domain.lfric.LFRicLoop) – the second Loop to fuse.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["same_space"]** (*bool*) – this optional flag, set to *True*, asserts that an unknown iteration space (i.e. *ANY_SPACE*) matches the other iteration space. This is set at the user's own risk. If both iteration spaces are discontinuous the loops can be fused without having to use the *same_space* flag.

Raises

- **TransformationError** – if either of the supplied loops contains an inter-grid kernel.
- **TransformationError** – if one or both function spaces have invalid names.
- **TransformationError** – if the *same_space* flag was set, but does not apply because neither field is on *ANY_SPACE* or the spaces are not the same.

- **TransformationError** – if one or more of the iteration spaces is unknown (*ANY_SPACE*) and the *same_space* flag is not set to *True*.
- **TransformationError** – if the loops are over different spaces that are not both discontinuous and the loops both iterate over cells.
- **TransformationError** – if the loops' upper bound names are not the same.
- **TransformationError** – if the halo-depth indices of two loops are not the same.
- **TransformationError** – if each loop already contains a reduction.
- **TransformationError** – if the first loop has a reduction and the second loop reads the result of the reduction.

class psyclone.domain.lfric.transformations.**RaisePSyIR2LFRicKernTrans**

Raise a generic PSyIR representation of a kernel-layer routine and metadata to an LFRic version with specialised domain-specific nodes and symbols. This is currently limited to the specialisation of kernel metadata.

```
>>> from psyclone.configuration import Config
>>> from psyclone.domain.lfric.transformations import
↪RaisePSyIR2LFRicKernTrans
>>> from psyclone.psyir.frontend.fortran import FortranReader
>>> config = Config.get().api_conf("lfric")
>>> CODE = """
... MODULE example
... TYPE, EXTENDS(kernel_type) :: compute_cu
...   TYPE(arg_type), DIMENSION(4) :: meta_args = (/     &
...     arg_type(GH_FIELD, GH_REAL, GH_INC, W1),         &
...     arg_type(GH_FIELD, GH_REAL, GH_READ, W3),        &
...     arg_type(GH_FIELD, GH_REAL, GH_READ, W3),        &
...     arg_type(GH_FIELD, GH_REAL, GH_READ, W3)/)
...   INTEGER :: OPERATES_ON = CELL_COLUMN
... CONTAINS
...   PROCEDURE, NOPASS :: code => compute_cu_code
... END TYPE compute_cu
... contains
...   subroutine compute_cu_code()
...   end subroutine
... end module""")
>>> fortran_reader = FortranReader()
>>> kernel_container = fortran_reader.psyir_from_source(CODE)
>>> trans = RaisePSyIR2LFRicKernTrans()
>>> trans.apply(kernel_container, {"metadata_name": "compute_cu"})
```

apply(*node*, *options=None*)

Raise the supplied language-level kernel to LFRic-specific kernel PSyIR. Specialises the kernel container to an LFRic-specific subclass, populates this subclass with the kernel metadata extracted from the metadata symbol as specified in metadata_name (which is supplied via the options argument) and removes the symbol from the symbol table.

Parameters

- **node** (psyclone.psyir.node.Container) – a kernel represented in generic PSyIR.
- **options** (*Optional[Dict[str: str]]*) – a dictionary with options for transformations. This is expected to contain the metadata_name.

validate(*node*, *options=None*)
Validate the supplied PSyIR tree.

Parameters

- **node** (psyclone.psyir.node.Container) – a PSyIR node that is the root of a PSyIR tree.
- **options** (*Optional[Dict[str: str]]*) – a dictionary with options for transformations.

Raises

- **TransformationError** – if the supplied node is not a Container.
- **TransformationError** – if the supplied node argument has a parent.
- **TransformationError** – if the metadata name has not been provided in the options argument.
- **TransformationError** – if the metadata name has not been set or does not exist in the code.
- **TransformationError** – if the metadata symbol does not reside in a Container (as opposed to a FileContainer).

class psyclone.transformations.**DynamoOMPParallelLoopTrans**(*omp_directive='do'*, *omp_schedule='static'*)
Dynamo-specific OpenMP loop transformation. Adds Dynamo specific validity checks. Actual transformation is done by the base class.

Parameters

- **omp_directive** (*str*) – choose which OpenMP loop directive to use. Defaults to "do".
- **omp_schedule** (*str*) – the OpenMP schedule to use. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'. Defaults to 'static'.

validate(*node*, *options=None*)
Perform LFRic-specific loop validity checks then call the *validate* method of the base class.

Parameters

- **node** (psyclone.psyir.nodes.Node) – the Node in the Schedule to check
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises

- **TransformationError** – if the supplied Node is not a LFRicLoop.
- **TransformationError** – if the associated loop requires colouring.

class psyclone.transformations.**Dynamo0p3AsyncHaloExchangeTrans**
Splits a synchronous halo exchange into a halo exchange start and halo exchange end. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "lfric"
>>> ast, invokeInfo = parse("file.f90", api=api)
>>> psy=PSyFactory(api).create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
```

(continues on next page)

(continued from previous page)

```
>>> # print(schedule.view())
>>>
>>> from psyclone.transformations import Dynamo0p3AsyncHaloExchangeTrans
>>> trans = Dynamo0p3AsyncHaloExchangeTrans()
>>> trans.apply(schedule.children[0])
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

apply(*node*, *options=None*)
Transforms a synchronous halo exchange, represented by a HaloExchange node, into an asynchronous halo exchange, represented by HaloExchangeStart and HaloExchangeEnd nodes.

Parameters

- **node** (psyclone.psygen.HaloExchange) – a synchronous haloexchange node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

property name

Returns the name of this transformation as a string.

Return type str

validate(*node*, *options*)
Internal method to check whether the node is valid for this transformation.

Parameters

- **node** (psyclone.psygen.HaloExchange) – a synchronous Halo Exchange node
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises **TransformationError** – if the node argument is not a HaloExchange (or subclass thereof)

class psyclone.transformations.**Dynamo0p3ColourTrans**
Split a Dynamo 0.3 loop over cells into colours so that it can be parallelised. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> import transformations
>>> import os
>>> import pytest
>>>
>>> TEST_API = "lfric"
>>> _,info=parse(os.path.join(os.path.dirname(os.path.abspath(__file__)),
>>>              "tests", "test_files", "dynamo0p3",
>>>              "4.6_multikernel_invokes.f90"),
>>>              api=TEST_API)
>>> psy = PSyFactory(TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0')
>>> schedule = invoke.schedule
>>>
>>> ctrans = Dynamo0p3ColourTrans()
>>> otrans = DynamoOMPParallelLoopTrans()
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>>     ctrans.apply(child)
>>>
>>> # Then apply OpenMP to each of the colour loops
>>> for child in schedule.children:
>>>     otrans.apply(child.children[0])
>>>
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
```

Colouring in the LFRic (Dynamo 0.3) API is subject to the following rules:

- Only kernels which operate on 'CELL_COLUMN's and which increment a field on a continuous function space require colouring. Kernels that update a field on a discontinuous function space will cause this transformation to raise an exception. Kernels that only write to a field on a continuous function space also do not require colouring but are permitted.
- A kernel may have at most one field with 'GH_INC' access.
- A separate colour map will be required for each field that is coloured (if an invoke contains >1 kernel call).

apply(*node*, *options=None*)
Performs LFRic-specific error checking and then uses the parent class to convert the Loop represented by node into a nested loop where the outer loop is over colours and the inner loop is over cells of that colour.

Parameters

- **node** (psyclone.domain.lfric.LFRicLoop) – the loop to transform.
- **options** – a dictionary with options for transformations. :type options: Optional[Dict[str, Any]]

class psyclone.transformations.**Dynamo0p3KernelConstTrans**
Modifies a kernel so that the number of dofs, number of layers and number of quadrature points are fixed in the kernel rather than being passed in by argument.

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "lfric"
>>> ast, invokeInfo = parse("file.f90", api=api)
>>> psy=PSyFactory(api).create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> from psyclone.transformations import Dynamo0p3KernelConstTrans
>>> trans = Dynamo0p3KernelConstTrans()
>>> for kernel in schedule.coded_kernels():
>>>     trans.apply(kernel, number_of_layers=150)
>>>     kernel_schedule = kernel.get_kernel_schedule()
>>>     # Uncomment the following line to see a text view of the
>>>     # symbol table
>>>     # print(kernel_schedule.symbol_table.view())
```

apply(*node*, *options=None*)

Transforms a kernel so that the values for the number of degrees of freedom (if a valid value for the element_order arg is provided), the number of quadrature points (if the quadrature arg is set to True) and the number of layers (if a valid value for the number_of_layers arg is provided) are constant in a kernel rather than being passed in by argument.

The "cellshape", "element_order" and "number_of_layers" arguments are provided to mirror the namelist values that are input into an LFRic model when it is run.

Quadrature support is currently limited to XYoZ in ths transformation. In the case of XYoZ the number of quadrature points (for horizontal and vertical) are set to the element_order + 3 in the LFRic infrastructure so their value is derived.

Parameters

- **node** (psyclone.domain.lfric.LFRicKern) – a kernel node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["cellshape"]** (*str*) – the shape of the cells. This is provided as it helps determine the number of dofs a field has for a particular function space. Currently only "quadrilateral" is supported which is also the default value.
- **options["element_order"]** (*int*) – the order of the cell. In combination with cellshape, this determines the number of dofs a field has for a particular function space. If it is set to None (the default) then the dofs values are not set as constants in the kernel, otherwise they are.
- **options["number_of_layers"]** (*int*) – the number of vertical layers in the LFRic model mesh used for this particular run. If this is set to None (the default) then the nlayers value is not set as a constant in the kernel, otherwise it is.
- **options["quadrature"]** (*bool*) – whether the number of quadrature points values are set as constants in the kernel (True) or not (False). The default is False.

property name

Returns the name of this transformation as a string.

Return type str

validate(*node*, *options=None*)

This method checks whether the input arguments are valid for this transformation.

Parameters

- **node** (psyclone.domain.lfric.LFRicKern) – a dynamo 0.3 kernel node.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["cellshape"]** (*str*) – the shape of the elements/cells.
- **options["element_order"]** (*int*) – the order of the elements/cells.
- **options["number_of_layers"]** (*int*) – the number of layers to use.
- **options["quadrature"]** (*bool*) – whether quadrature dimension sizes should or shouldn't be set as constants in a kernel.

Raises **TransformationError** – if the node argument is not a dynamo 0.3 kernel, the cellshape argument is not set to "quadrilateral", the element_order argument is not a 0 or a positive integer, the number of layers argument is not a positive integer, the quadrature argument is

not a boolean, neither element order nor number of layers arguments are set (as the transformation would then do nothing), or the quadrature argument is True but the element order is not provided (as the former needs the latter).

class psyclone.transformations.**Dynamo0p3OMPLoopTrans**(*omp_schedule='static'*)
LFRic (Dynamo 0.3) specific orphan OpenMP loop transformation. Adds Dynamo-specific validity checks.

Parameters **omp_schedule** (*str*) – the OpenMP schedule to use. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'. Defaults to 'static'.

apply(*node*, *options=None*)
Apply LFRic (Dynamo 0.3) specific OMPLoopTrans.

Parameters

- **node** (psyclone.psyir.nodes.Node) – the Node in the Schedule to check.
- **options** (*Optional[dict[str, Any]]*) – a dictionary with options for transformations and validation.
- **options["reprod"]** (*bool*) – indicating whether reproducible reductions should be used. By default the value from the config file will be used.

validate(*node*, *options=None*)
Perform LFRic (Dynamo 0.3) specific loop validity checks for the OMPLoopTrans.

Parameters

- **node** (psyclone.psyir.nodes.Node) – the Node in the Schedule to check
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations and validation.
- **options["reprod"]** (*bool*) – indicating whether reproducible reductions should be used. By default the value from the config file will be used.

Raises **TransformationError** – if an OMP loop transform would create incorrect code.

class psyclone.transformations.**Dynamo0p3RedundantComputationTrans**
This transformation allows the user to modify a loop's bounds so that redundant computation will be performed. Redundant computation can result in halo exchanges being modified, new halo exchanges being added or existing halo exchanges being removed.

- This transformation should be performed before any parallelisation transformations (e.g. for OpenMP) to the loop in question and will raise an exception if this is not the case.
- This transformation can not be applied to a loop containing a reduction and will again raise an exception if this is the case.
- This transformation can only be used to add redundant computation to a loop, not to remove it.
- This transformation allows a loop that is already performing redundant computation to be modified, but only if the depth is increased.

apply(*loop*, *options=None*)
Apply the redundant computation transformation to the loop `loop`. This transformation can be applied to loops iterating over 'cells or 'dofs'. if `depth` is set to a value then the value will be the depth of the field's halo over which redundant computation will be performed. If `depth` is not set to a value then redundant computation will be performed to the full depth of the field's halo.

Parameters

- **loop** (psyclone.psyGen.LFRicLoop) – the loop that we are transforming.

- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["depth"]** (*int*) – the depth of the stencil. Defaults to None.

validate(*node*, *options=None*)

Perform various checks to ensure that it is valid to apply the RedundantComputation transformation to the supplied node

Parameters

- **node** (psyclone.psyir.nodes.Node) – the supplied node on which we are performing validity checks
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["depth"]** (*int*) – the depth of the stencil if the value is provided and None if not.

Raises

- **TransformationError** – if the parent of the loop is a psyclone.psyir.nodes.Directive.
- **TransformationError** – if the parent of the loop is not a psyclone.psyir.nodes.Loop or a psyclone.psyGen.LFRicInvokeSchedule.
- **TransformationError** – if the parent of the loop is a psyclone.psyir.nodes.Loop but the original loop does not iterate over 'colour'.
- **TransformationError** – if the parent of the loop is a psyclone.psyir.nodes.Loop but the parent does not iterate over 'colours'.
- **TransformationError** – if the parent of the loop is a psyclone.psyir.nodes.Loop but the parent's parent is not a psyclone.psyGen.LFRicInvokeSchedule.
- **TransformationError** – if this transformation is applied when distributed memory is not switched on.
- **TransformationError** – if the loop does not iterate over cells, dofs or colour.
- **TransformationError** – if the loop contains a kernel that operates on halo cells.
- **TransformationError** – if the transformation is setting the loop to the maximum halo depth but the loop already computes to the maximum halo depth.
- **TransformationError** – if the transformation is setting the loop to the maximum halo depth but the loop contains a stencil access (as this would result in the field being accessed beyond the halo depth).
- **TransformationError** – if the supplied depth value is not an integer.
- **TransformationError** – if the supplied depth value is less than 1.
- **TransformationError** – if the supplied depth value is not greater than 1 when a continuous loop is modified as this is the minimum valid value.
- **TransformationError** – if the supplied depth value is not greater than the existing depth value, as we should not need to undo existing transformations.
- **TransformationError** – if a depth value has been supplied but the loop has already been set to the maximum halo depth.

CHAPTER

TEN

GOCEAN1.0 API

10.1 Introduction

The GOcean 1.0 application programming interface (API) was originally designed to support ocean models that use the finite-difference scheme for two-dimensional domains. However, the approach is not specific to ocean models and can potentially be applied to any finite-difference code.

As with all PSyclone APIs, the GOcean 1.0 API specifies how a user must write the Algorithm Layer and the Kernel Layer to allow PSyclone to generate the PSy Layer. These Algorithm and Kernel APIs are discussed separately in the sections below. Before these we describe the functionality provided by the GOcean Library.

10.2 The GOcean Infrastructure Library - dl_esm_inf

The use of PSyclone and the GOcean 1.0 API implies the use of a standard set of data types and associated infrastructure. This is provided by the GOcean infrastructure library - dl_esm_inf. Currently this library is distributed separately from PSyclone and is available from https://github.com/stfc/dl_esm_inf.

10.2.1 Grid

The dl_esm_inf library contains a `grid_mod` module which defines a `grid_type` and associated constructor:

```
use grid_mod
...
!> The grid on which our fields are defined
type(grid_type), target :: model_grid
...
! Create the model grid
model_grid = grid_type(GO_ARAKAWA_C,                                    &
                       (/GO_BC_EXTERNAL,GO_BC_EXTERNAL,GO_BC_NONE/), &
                       GO_OFFSET_NE)
```

Note: The grid object itself must be declared with the `target` attribute. This is because each field object will contain a pointer to it.

The `grid_type` constructor takes three arguments:

1. The type of grid (only GO_ARAKAWA_C is currently supported)

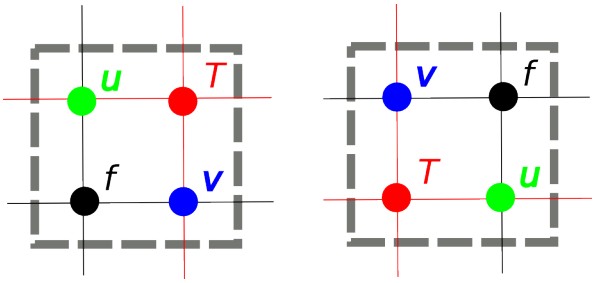
2. The boundary conditions on the domain for the x, y and z dimensions (see below). The value for the z dimension is currently ignored.
3. The 'index offset' - the convention used for indexing into offset fields.

Three types of boundary condition are currently supported:

Name	Description
GO_BC_NONE	No boundary conditions are applied.
GO_BC_EXTERNAL	Some external forcing is applied. This must be implemented by a kernel. The domain must be defined with a T-point mask (see *The grid_init Routine*).
GO_BC_PERIODIC	Periodic boundary conditions are applied.

The infrastructure requires this information in order to determine the extent of the model grid.

The index offset is required because a model (kernel) developer has choice in how they actually implement the staggering of variables on a grid. This comes down to a choice of which grid points in the vicinity of a given T point have the same array (i, j) indices. In the diagram below, the image on the left corresponds to choosing those points to the South and West of a T point to have the same (i, j) index. That on the right corresponds to choosing those points to the North and East of the T point (this is the offset scheme used in the NEMO ocean model):



The GOcean 1.0 API supports these two different offset schemes, which we term `GO_OFFSET_SW` and `GO_OFFSET_NE`.

Note that the constructor does not specify the extent of the model grid. This is because this information is normally obtained by reading a file (a namelist file, a netcdf file etc.) which is specific to an application. Once this information has been obtained, a second routine, `grid_init`, is provided with which to 'load' a grid object with state. This is discussed below.

The `grid_init` Routine

Once an application has determined the details of the model configuration, it must use this information to populate the grid object. This is done via a call to the `grid_init` subroutine:

```
subroutine grid_init(grid, m, n, dxarg, dyarg, tmask)
  !> The grid object to configure
  type(grid_type), intent(inout) :: grid
  !> Dimensions of the model grid
  integer,         intent(in)    :: m, n
  !> The (constant) grid spacing in x and y (m)
  real(wp),        intent(in)    :: dxarg, dyarg
  !> Optional T-point mask specifying whether each grid point is
  !! wet (1), dry (0) or external (-1).
  integer, dimension(m,n), intent(in), optional :: tmask
```

If no T-mask is supplied then this routine configures the grid appropriately for an all-wet domain with periodic boundary conditions in both the *x*- and *y*-dimensions. It should also be noted that currently only grids with constant resolution in *x* and *y* are supported by this routine.

10.2.2 Fields

Once a model has a grid defined it will require one or more fields. The dl_esm_inf library contains a `field_mod` module which defines an `r2d_field` type (real, 2-dimensional field) and associated constructor:

```
use field_mod
...
!> Current ('now') sea-surface height at different grid points
type(r2d_field) :: sshn_u_fld, sshn_v_fld, sshn_t_fld
...

! Sea-surface height now (current time step)
sshn_u = r2d_field(model_grid, GO_U_POINTS)
sshn_v = r2d_field(model_grid, GO_V_POINTS)
sshn_t = r2d_field(model_grid, GO_T_POINTS)
```

The constructor takes two arguments:

1. The grid on which the field exists
2. The type of grid point at which the field is defined (`GO_U_POINTS`, `GO_V_POINTS`, `GO_T_POINTS` or `GO_F_POINTS`)

Note that the grid object need not have been fully configured (by a call to `grid_init` for instance) before it is passed into this constructor.

10.2.3 Example

PSyclone is distributed with a full example of the use of the GOcean Library. See `<PSYCLONEHOME>/examples/gocean/shallow_alg.f90`. In what follows we will walk through a slightly cut-down example for a different program.

The following code illustrates the use of dl_esm_inf for constructing an application:

```
program gocean2d
  use grid_mod  ! From dl_esm_inf
  use field_mod ! From dl_esm_inf
  use model_mod
  use boundary_conditions_mod

  !> The grid on which our fields are defined. Must have the 'target'
  !! attribute because each field object contains a pointer to it.
  type(grid_type), target :: model_grid

  !> Current ('now') velocity component fields
  type(r2d_field) :: un_fld, vn_fld
  !> 'After' velocity component fields
  type(r2d_field) :: ua_fld, va_fld
  ...

  ! time stepping index
```

(continues on next page)

(continued from previous page)

```
  integer :: istp

  ! Create the model grid. We use a NE offset (i.e. the U, V and F
  ! points immediately to the North and East of a T point all have the
  ! same i,j index).  This is the same offset scheme as used by NEMO.
  model_grid = grid_type(GO_ARAKAWA_C,                                  &
                         (/GO_BC_EXTERNAL,GO_BC_EXTERNAL,GO_BC_NONE/), &
                          GO_OFFSET_NE)

  !! read in model parameters and configure the model grid
  CALL model_init(model_grid)

  ! Create fields on this grid

  ! Velocity components now (current time step)
  un_fld = r2d_field(model_grid, GO_U_POINTS)
  vn_fld = r2d_field(model_grid, GO_V_POINTS)

  ! Velocity components 'after' (next time step)
  ua_fld = r2d_field(model_grid, GO_U_POINTS)
  va_fld = r2d_field(model_grid, GO_V_POINTS)

  ...

  !! time stepping
  do istp = nit000, nitend, 1

    call step(istp,                                 &
              ua_fld, va_fld, un_fld, vn_fld,       &
              ...)
  end do
  ...
end program gocean2d
```

The `model_init` routine is application specific since it must determine details of the model configuration being run, *e.g.* by reading a namelist file. An example might look something like:

```
subroutine model_init(grid)
  type(grid_type), intent(inout) :: grid

  !> Problem size, read from namelist
  integer :: jpiglo, jpjglo
  real(wp) :: dx, dy
  integer, dimension(:,:), allocatable :: tmask

  ! Read model configuration from namelist
  call read_namelist(jpiglo, jpjglo, dx, dy, &
                     nit000, nitend, irecord, &
                     jphgr_msh, dep_const, rdt, cbfr, visc)

  ! Set-up the T mask. This defines the model domain.
  allocate(tmask(jpiglo,jpjglo))
```

(continues on next page)

(continued from previous page)

```
  call setup_tpoints_mask(jpiglo, jpjglo, tmask)

  ! Having specified the T points mask, we can set up mesh parameters
  call grid_init(grid, jpiglo, jpjglo, dx, dy, tmask)

  ! Clean-up. T-mask has been copied into the grid object.
  deallocate(tmask)

end subroutine model_init
```

Here, only `grid_type` and the `grid_init` routine come from dl_esm_inf. The remaining code is all application specific.

Once the grid object is fully configured and all fields have been constructed, a simulation will proceed by performing calculations with those fields. In the example program given above, this calculation is performed in the time-stepping loop within the `step` subroutine. The way in which this routine uses Invoke calls is described in the *Invokes* Section.

10.3 Algorithm

The Algorithm is the top-level specification of the natural science implemented in the software. Essentially it consists of mesh setup, field declarations, initialisation of fields and (a series of) Kernel calls. Infrastructure to support these tasks is provided in version 1.0 of the GOcean library (see *The GOcean Infrastructure Library - dl_esm_inf*).

10.3.1 Invokes

The Kernels to call are specified through the use of Invokes, e.g.:

```
call invoke( kernel1(field1, field2),                    &
             kernel2(field1, field3)                     &
           )
```

The location and number of these `call invoke(...)` statements within the source code is entirely up to the user. The only requirement is that PSyclone must be run on every source file that contains one or more Invokes. The body of each Invoke specifies the kernels to be called, the order in which they are to be applied and the fields (and scalars) that they work with.

Note that the kernel names specified in an Invoke are the names of the corresponding kernel *types* defined in the kernel metadata (see the *Kernel* Section). These are not the same as the names of the Fortran subroutines which contain the actual kernel code. The kernel arguments are typically field objects, as described in the *Fields* Section, but they may also be scalar quantities (real or integer).

In the example `gocean2d` program shown earlier, there is only one Invoke call and it is contained within the `step` subroutine:

```
subroutine step(istp,                    &
                ua, va, un, vn,          &
                sshn_t, sshn_u, sshn_v, &
                ssha_t, ssha_u, ssha_v, &
                hu, hv, ht)
  use kind_params_mod  ! From dl_esm_inf
```

(continues on next page)

(continued from previous page)

```
  use grid_mod         ! From dl_esm_inf
  use field_mod        ! From dl_esm_inf
  use model_mod, only: rdt ! The model time-step
  use continuity_mod,  only: continuity
  use momentum_mod,    only: momentum_u, momentum_v
  use boundary_conditions_mod, only: bc_ssh, bc_solid_u
  !> The current time step
  integer,         intent(inout) :: istp
  type(r2d_field), intent(inout) :: un, vn, sshn_t, sshn_u, sshn_v
  type(r2d_field), intent(inout) :: ua, va, ssha_t, ssha_u, ssha_v
  type(r2d_field), intent(inout) :: hu, hv, ht

  call invoke(                                              &
              continuity(ssha_t, sshn_t, sshn_u, sshn_v,    &
                         hu, hv, un, vn, rdt),              &
              momentum_u(ua, un, vn, hu, hv, ht,            &
                         ssha_u, sshn_t, sshn_u, sshn_v),   &
              momentum_v(va, un, vn, hu, hv, ht,            &
                         ssha_v, sshn_t, sshn_u, sshn_v),   &
              bc_ssh(istp, ssha_t),                         &
              bc_solid_u(ua),                               &
              ...
             )
end subroutine step
```

Note that in this example the grid was constructed for a model with 'external' boundary conditions. These boundary conditions are applied through several user-supplied kernels, two of which (`bc_ssh` and `bc_solid_u`) are include in the above code fragment.

10.4 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. This section explains the metadata and subroutine arguments that are specific to the GOcean 1.0 API.

10.4.1 Metadata

The metadata for a GOcean 1.0 API kernel has four components:

1) 'meta_args',
2) 'iterates_over',
3) 'index_offset' and
4) 'procedure':

These are illustrated in the code below:

```
type, extends(kernel_type) :: my_kernel_type
   type(go_arg), dimension(...) :: meta_args = (/ ... /)
   integer :: iterates_over = ...
   integer :: index_offset = ...
```

(continues on next page)

(continued from previous page)

```
contains
  procedure, nopass :: code => my_kernel_code
end type my_kernel_type
```

These four metadata elements are discussed in order in the following sections.

Argument Metadata: meta_args

The `meta_args` array specifies information about data that the kernel code expects to be passed to it via its argument list. There is one entry in the `meta_args` array for each **scalar**, **field**, or **grid-property** passed into the Kernel. Their ordering in the `meta_args` array must be the same as that in the kernel code argument list. The entry must be of type `go_arg` which itself contains metadata about the associated argument. The size of the meta_args array must correspond to the total number of **scalars**, **fields** and **grid properties** passed into the Kernel.

For example, if there are a total of two **field** entities being passed to the Kernel then the meta_args array will be of size 2 and there will be two entries of type `GO_arg`:

```
type(GO_arg) :: meta_args(2) = (/                                    &
     go_arg( ... ),                                                  &
     go_arg( ... )                                                   &
     /)
```

Argument-metadata (metadata contained within the brackets of an `go_arg` entry), describes either a **scalar**, a **field** or a **grid property**.

The first argument-metadata entry describes how the kernel will access the corresponding argument. As an example, the following `meta_args` metadata describes four entries, the first one is written to by the kernel while the remaining three are only read:

```
type(go_arg) :: meta_args(4) = (/                                    &
     go_arg(GO_WRITE, ... ),                                         &
     go_arg(GO_READ, ... ),                                          &
     go_arg(GO_READ, ... ),                                          &
     go_arg(GO_READ, ...)                                            &
     /)
```

The second entry to argument-metadata (information contained within the brackets of an `go_arg` type) describes the type of data represented by the argument. This type falls into three categories; field data, scalar data and grid properties. For field data the metadata entry consists of the type of grid-point that field values are defined on. Since the GOcean API supports fields on an Arakawa C grid, the possible grid-point types are `GO_CU`, `GO_CV`, `GO_CF` and `GO_CT`. GOcean Kernels can also take scalar quantities as arguments. Since these do not live on grid-points they are specified as either `GO_R_SCALAR` or `GO_I_SCALAR` depending on whether the corresponding Fortran variable is a real or integer quantity. Finally, grid-property entries are used to specify any properties of the grid required by the kernel (*e.g.* the area of cells at U points or whether T points are wet or dry).

For example:

```
type(go_arg) :: meta_args(4) = (/                                    &
     go_arg(GO_WRITE, GO_CT, ... ),                                  &
     go_arg(GO_READ,  GO_CU, ... ),                                  &
     go_arg(GO_READ,  GO_R_SCALAR, ... ),                            &
     go_arg(GO_READ,  GO_GRID_AREA_U)                                &
     /)
```

Here, the first argument is a field on T points, the second is a field on U points, the fourth is a real scalar and the fifth is a property of the grid (cell area at U points).

The full list of supported grid properties in the GOcean 1.0 API is:

Table 10.1: Grid Properties Table

Name	Description	Type
go_grid_area_t	Cell area at T point	Real array, rank=2
go_grid_area_u	Cell area at U point	Real array, rank=2
go_grid_area_v	Cell area at V point	Real array, rank=2
go_grid_mask_t	T-point mask (1=wet, 0=dry)	Integer array, rank=2
go_grid_dx_t	Grid spacing in x at T points	Real array, rank=2
go_grid_dx_u	Grid spacing in x at U points	Real array, rank=2
go_grid_dx_v	Grid spacing in x at V points	Real array, rank=2
go_grid_dy_t	Grid spacing in y at T points	Real array, rank=2
go_grid_dy_u	Grid spacing in y at U points	Real array, rank=2
go_grid_dy_v	Grid spacing in y at V points	Real array, rank=2
go_grid_lat_u	Latitude of U points (gphiu)	Real array, rank=2
go_grid_lat_v	Latitude of V points (gphiv)	Real array, rank=2
go_grid_dx_const	Grid spacing in x if constant	Real, scalar
go_grid_dy_const	Grid spacing in y if constant	Real, scalar
go_grid_x_min_index	Minimum X index	Integer, scalar
go_grid_x_max_index	Maximum X index	Integer, scalar
go_grid_y_min_index	Minimum Y index	Integer, scalar
go_grid_y_max_index	Maximum Y index	Integer, scalar

These are defined in the psyclone config file (see *Configuration*), and the user or infrastructure library developer can provide additional entries if required. PSyclone will query PSyclone's Configuration class to get the properties required. All of the rank-two arrays have the first rank as longitude (x) and the second as latitude (y).

Scalars and fields contain a third argument-metadata entry which describes whether the kernel accesses the corresponding argument with a stencil. The value `GO_POINTWISE` indicates that there is no stencil access. Metadata for a scalar field is limited to this value. Grid-property arguments have no third metadata argument. If there are no stencil accesses then the full argument metadata for our previous example will be:

```
type(go_arg) :: meta_args(4) = (/                              &
     go_arg(GO_WRITE, GO_CT,       GO_POINTWISE),              &
     go_arg(GO_READ,  GO_CU,       GO_POINTWISE),              &
     go_arg(GO_READ,  GO_R_SCALAR, GO_POINTWISE),              &
     go_arg(GO_READ,  GO_GRID_AREA_U)                          &
     /)
```

If a kernel accesses a field using a stencil then the third argument metadata entry should take the form `go_stencil(...)`. Note, a stencil access is only allowed for a field that is `READ` by a kernel.

In the GOcean API, fields are implemented as two-dimensional arrays. In Fortran, a standard 5-point stencil would look something like the following:

```
a(i,j) + a(i+1,j) + a(i-1,j) + a(i,j+1) + a(i,j-1)
```

If we view the above accesses as co-ordinates relative to the `a(i,j)` access we get `(0,0), (1,0), (-1,0), (0,1), (0,-1)`. If we then view these accesses in graphical form with `i` being in the horizontal direction and `j` in the vertical and with a `1` indicating a (depth-1) access and a `0` indicating there is no access we get the following:

```
010
111
010
```

In the GOcean API a stencil access is captured as a triplet of integers (one row at a time from top to bottom) using the above view i.e.

```
go_stencil(010,111,010)
```

So far we have only considered depth-1 stencils. In our notation the depth of access is captured by the integer value (`0` for no access, `1` for depth 1, `2` for depth 2 etc). For example:

```
a(i,j) + a(i,j+1) + a(i,j+2)
```

would be captured as:

```
go_stencil(020,010,000)
```

All forms of stencil can be **summarised** using this triplet notation up to a depth of `9` apart from the central `a(i,j)` value which can either be `0` (not accessed) or `1` (accessed). Note, the central value is not currently used by PSyclone. The notation is a **summary** in two ways

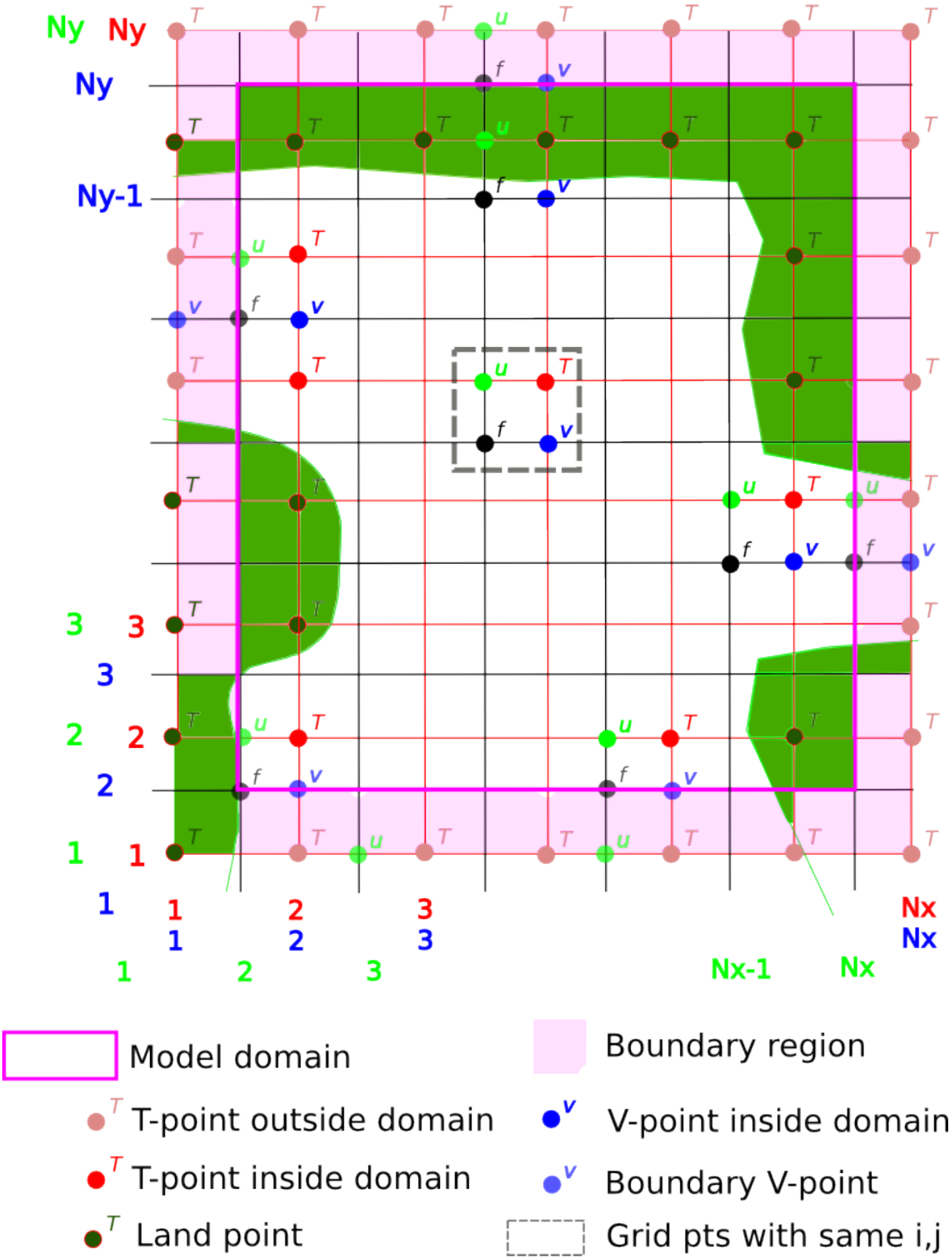
1) it only captures the depth of the stencil in a particular direction, not the actual accesses. Therefore, there is no way to distinguish between the stencil `a(i+2,j)` and the stencil `a(i+1,j) + a(i+2,j)`.

2) when there are offsets for both `i` and `j` e.g. `a(i+1,j+1)` it only captures whether there is an access in that direction at a particular depth, not the details of the access. For example, there is no way to distinguish between `a(i+2,j+2)` and `a(i+2,j+2) + a(i+1,j+2) + a(i+2,j+1)`.

Whilst the description is a summary, it is accurate enough for PSyclone as this information is primarily used to determine which grid partitions must communicate with which for the purposes of placing halo exchange calls. In this case, it is the depth and direction information that is most important.

Iterates Over

The second element of kernel metadata is `ITERATES_OVER`. This specifies that the Kernel has been written with the assumption that it is iterating over grid points of the specified type. By default the supported values are: `GO_INTERNAL_PTS`, `GO_EXTERNAL_PTS` and `GO_ALL_PTS`. These may be understood by considering the following diagram of an example model configuration:



GO_INTERNAL_PTS are then those points that are within the Model domain (fuscia box), GO_EXTERNAL_PTS are those

outside the domain and GO_ALL_PTS encompasses all grid points in the model. The chosen value is specified in the kernel-meta data like so:

```
integer :: iterates_over = GO_INTERNAL_PTS
```

A user can use a config file (see *Configuration*) to add additional iteration spaces to PSyclone.

Index Offset

The third element of kernel metadata, INDEX_OFFSET, specifies the index-offset that the kernel uses. This is the same quantity as supplied to the grid constructor (see the *Grid* Section for a description).

The GOcean 1.0 API supports two different offset schemes; GO_OFFSET_NE, GO_OFFSET_SW. The scheme used by a kernel is specified in the metadata as, e.g.:

```
integer :: index_offset = GO_OFFSET_NE
```

Currently all kernels used in an application must use the same offset scheme which must also be the same as passed to the grid constructor.

Procedure

The fourth and final type of metadata is procedure metadata. This specifies the name of the Kernel Fortran subroutine that this metadata describes.

For example:

```
procedure :: my_kernel_code
```

10.4.2 Subroutine

Rules

Kernel arguments follow a set of rules which have been specified for the GOcean 1.0 API. These rules are encoded in the gen_code() method of the GOKern class in the gocean1p0.py file. The rules, along with PSyclone's naming conventions, are:

1) Every kernel has the indices of the current grid point as the first two arguments, i and j. These are integers and have intent in.
2) For each field/scalar/grid property in the order specified by the meta_args metadata:
    1) For a field; the field array itself. A field array is a real array of kind go_wp and rank two. The first rank is longitude (x) and the second latitude (y).
    2) For a scalar; the variable itself. A real scalar is of kind go_wp.
    3) For a grid property; the array or variable (see the earlier table) containing the specified property.

Note: Grid properties are not passed from the Algorithm Layer. PSyclone generates the necessary lookups in the PSy Layer and includes the resulting references in the arguments passed to the kernel.

As an example, consider the bc_solid_u kernel that is used in the gocean2d program shown earlier. The metadata for this kernel is:

```
 type, extends(kernel_type) :: bc_solid_u
   type(go_arg), dimension(2) :: meta_args =  &
        (/ go_arg(GO_WRITE, GO_CU, GO_POINTWISE),      &
           go_arg(GO_READ,      GO_GRID_MASK_T)     &
         /)

   !> This is a boundary-conditions kernel and therefore
   !! acts on all points of the domain rather than just
   !! those that are internal
   integer :: ITERATES_OVER = GO_ALL_PTS

   integer :: index_offset = GO_OFFSET_NE

contains
  procedure, nopass :: code => bc_solid_u_code
end type bc_solid_u
```

The interface to the subroutine containing the implementation of this kernel is:

```
subroutine bc_solid_u_code(ji, jj, ua, tmask)
  integer,                  intent(in)    :: ji, jj
  integer,  dimension(:,:), intent(in)    :: tmask
  real(wp), dimension(:,:), intent(inout) :: ua
```

As described above, the first two arguments to this subroutine specify the grid-point at which the computation is to be performed. The third argument is the field that this kernel updates and the fourth argument is the T-point mask. The latter is a property of the grid and is provided to the kernel call from the PSy Layer.

Comparing this interface definition with the use of the kernel in the Invoke call:

```
call invoke ( ...,            &
              bc_solid_u(ua), &
              ... )
```

we see that in the Algorithm Layer the user need only provide the field(s) (and possibly scalars) that a kernel operates on. The index of the grid point and any grid properties are provided in the (generated) PSy Layer where the kernel subroutine proper is called.

10.5 Built-ins

The GOcean 1.0 API does not support any built-in operations.

10.6 Conventions

The GOcean 1.0 API kernel code conforms to the PSyclone Fortran naming conventions (see *Fortran Naming Conventions*). However, PSyclone's support for the GOcean 1.0 API does not rely on this convention.

The contents of the kernel metadata is usually declared private but this does not affect PSyclone.

Finally, the `procedure` metadata (located within the kernel metadata) usually has `nopass` specified but again this is ignored by PSyclone.

10.7 Configuration

The configuration file (see *Configuration*) used by PSyclone can contain GOcean 1.0 specific options. For example, after the default section the GOcean 1.0 specific section looks like this:

```
[gocean]
iteration-spaces=offset_sw:ct:test_only:1:2:3:4
                 offset_sw:ct:internal_ns_halo:{start}-1:{stop}+1:{start}:{stop}
```

The supported keys are listed in the next section.

10.7.1 Iteration-spaces

This section lists additional iteration spaces that can be used in a kernel metadata declaration to allow PSyclone to create a loop with different loop boundaries. Each line of the `iteration-spaces` declaration contains 7 values, separated by ':'. The fields are:

Field	Description	Details
1	Index Offset	See *Index Offset*.
2	grid-point types	See *Grid point types*.
3	Iterates Over	See *Iterates Over*.
4	Start index of outer loop	Start index of North-South loop.
5	End index of outer loop	End index of North-South loop.
6	Start index of inner loop	Start index of East-West loop.
7	End index of inner loop	End index of East-West loop.

Two special variables can be used in an iteration space: `{start}` and `{stop}`. These values will be replaced by PSyclone with the correct loop boundaries for the inner points of a grid (i.e. the non-halo area). This means that the depth-1 halo region can be specified using `{start}-1` and `{stop}+1`.

For example, given the iteration-spaces declaration above, a kernel declared with `iterates_over=internal_ns_halo` for a field type `ct` and index offset `offset_sw` would create the following loop boundaries:

```
DO j=2-1,jstop+1
  DO i=2,istop
    CALL (i, j, ...)
  END DO
END DO
```

Warning: With user defined iteration spaces it is possible that PSyclone will create code that does not compile: if you specify syntactically correct, but semantically incorrect boundary definitions, the PSyclone internal tests will accept the new iteration space, but the compiler will not. For example if one of the loop boundaries contains the name of a variable that is not defined, compilation will fail. It is the responsibility of the user to make sure that valid loop boundaries are specified in a new iteration space definition.

10.7.2 Grid Properties

Various grid properties can be specified as parameters to a kernel. The actual names and meaning of these properties depend on the infrastructure library used. By default PSyclone provides settings for the dl_esm_inf infrastructure library. But the user or a library developer can change or add definitions to the configuration file as required.

The grid properties are specified as values for the key `grid-properties`. They consist of three entries, separated by ":".

- The first entry is the name of the property as used in kernel metadata.
- The next entry is the way of dereferencing the corresponding value in Fortran. The expression `{0}` is replaced with the field name that is used. Note that any `%` must be replaced with `%%` (due to the way Python reads in configuration files).
- The last entry specifies whether the value is an `array` or a `scalar`.

Below an excerpt from the configuration file that is distributed with PSyclone:

```
grid-properties = go_grid_xstop: {0}%%grid%%subdomain%%internal%%xstop: scalar,
                  go_grid_ystop: {0}%%grid%%subdomain%%internal%%ystop: scalar,
                  go_grid_data: {0}%%data: array,
                  go_grid_internal_inner_stop: {0}%%internal%%xstop: scalar,
                  go_grid_internal_outer_stop: {0}%%internal%%ystop: scalar,
                  go_grid_whole_inner_stop: {0}%%whole%%xstop: scalar,
                  go_grid_whole_outer_stop: {0}%%whole%%ystop: scalar,
...
```

Most of the property names can be set arbitrarily by the user (to match whatever infrastructure library is being used), but PSyclone relies on a small number of properties that must be defined with the right name:

Key	Description
go_grid_data	This property gives access to the raw 2d-field.
go_grid_xstop, go_grid_ystop	These values specify the upper loop boundary when computing the constant loop boundaries.
go_grid_{internal,whole} _{inner,outer}_{start,stop}	These eight values are required to specify the loop boundaries depending on the field space.
go_grid_nx, go_grid_ny	These properties are only required when OpenCL is enabled. They specify the overall array size (including any padding that the infrastructure library might implement).

10.7.3 Debug Mode

The GOcean configuration also includes a boolean parameter to enable or disable the generation of additional code which may impact performance but is useful for debugging the application. By default it is set to False, but it can be changed by updating the following line in the configuration file:

```
[gocean]
DEBUG_MODE = true
```

Currently, only the OpenCL Invokes generate additional debugging code.

10.8 Transformations

In this section we describe the transformations that are specific to the GOcean 1.0 API. For an overview of transformations in general see *Transformations*.

class psyclone.domain.gocean.transformations.**GOceanExtractTrans**

GOcean1.0 API application of ExtractTrans transformation to extract code into a stand-alone program. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>>
>>> API = "gocean"
>>> FILENAME = "shallow_alg.f90"
>>> ast, invokeInfo = parse(FILENAME, api=API)
>>> psy = PSyFactory(API, distributed_memory=False).create(invoke_info)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>>
>>> from psyclone.domain.gocean.transformations import GOceanExtractTrans
>>> etrans = GOceanExtractTrans()
>>>
>>> # Apply GOceanExtractTrans transformation to selected Nodes
>>> etrans.apply(schedule.children[0])
>>> print(schedule.view())
```

apply(*nodes*, *options=None*)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single PSyData region. Note that this implementation just calls the base class, it is only added here to provide the documentation for this function, since it accepts different options to the base class (e.g. create_driver, which is passed to the ExtractNode instance that will be inserted.).

Parameters

- **nodes** (`psyclone.psyir.nodes.Node` or list of `psyclone.psyir.nodes.Node`) – can be a single node or a list of nodes.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (`prefix_psy_data_mod`) and the PSyDataType (`prefix_PSyDataType`) - a "_" will be added automatically. It defaults to "extract", resulting in e.g. `extract_psy_data_mod`.
- **options["create_driver"]** (*bool*) – whether or not to create a driver program at code-generation time. If set, the driver will be created in the current working directory with the name "driver-MODULE-REGION.f90" where MODULE and REGION will be the corresponding values for this region. Defaults to False.
- **options["region_name"]** (*(str,str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

validate(*node_list*, *options=None*)

Perform GOcean1.0 API specific validation checks before applying the transformation.

Parameters

- **node_list** (list of psyclone.psyir.nodes.Node) – the list of Node(s) we are checking.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.
- **options["create_driver"]** (*bool*) – whether or not to create a driver program at code-generation time. If set, the driver will be created in the current working directory with the name "driver-MODULE-REGION.f90" where MODULE and REGION will be the corresponding values for this region. This flag is forwarded to the ExtractNode. Its default value is False.
- **options["region_name"]** (*(str,str)*) – an optional name to use for this data-extraction region, provided as a 2-tuple containing a module name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library). This option is forwarded to the PSyDataNode (where it changes the region names) and to the ExtractNode (where it changes the name of the created output files and the name of the driver program).

Raises **TransformationError** – if transformation is applied to an inner Loop without its parent outer Loop.

---

class psyclone.domain.gocean.transformations.**GOceanLoopFuseTrans**

GOcean API specialisation of the base class in order to fuse two GOcean loops after performing validity checks (e.g. that the loops are over the same grid-point type). For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> ast, invokeInfo = parse("shallow_alg.f90")
>>> psy = PSyFactory("gocean").create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> print(schedule.view())
>>>
>>> from psyclone.transformations import GOceanLoopFuseTrans
>>> ftrans = GOceanLoopFuseTrans()
>>> ftrans.apply(schedule[0], schedule[1])
>>> print(schedule.view())
```

validate(*node1*, *node2*, *options=None*)

Checks if it is valid to apply the GOceanLoopFuseTrans transform. It ensures that the fused loops are over the same grid-point types, before calling the normal LoopFuseTrans validation function.

Parameters

- **node1** (psyclone.gocean1p0.GOLoop) – the first Node representing a GOLoop.
- **node2** (psyclone.gocean1p0.GOLoop) – the second Node representing a GOLoop.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises

- **TransformationError** – if the supplied loops are over different grid-point types.
- **TransformationError** – if invalid parameters are passed in.

---

class psyclone.transformations.**GOceanOMPParallelLoopTrans**(*omp_directive='do'*, *omp_schedule='static'*)

GOcean specific OpenMP Do loop transformation. Adds GOcean specific validity checks (that supplied Loop is an inner or outer loop). Actual transformation is done by `base class`.

param str omp_directive choose which OpenMP loop directive to use. Defaults to "do".

param str omp_schedule the OpenMP schedule to use. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'. Defaults to 'static'.

apply(*node*, *options=None*)

Perform GOcean-specific loop validity checks then call `OMPParallelLoopTrans.apply()`.

Parameters

- **node** (`psyclone.psyir.nodes.Loop`) – a Loop node from an AST.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations and validation.

Raises **TransformationError** – if the supplied node is not an inner or outer loop.

---

class psyclone.transformations.**GOceanOMPLoopTrans**(*omp_directive='do'*, *omp_schedule='static'*)

GOcean-specific orphan OpenMP loop transformation. Adds GOcean specific validity checks (that the node is either an inner or outer Loop).

Parameters

- **omp_directive** (*str*) – choose which OpenMP loop directive to use. Defaults to "do".
- **omp_schedule** (*str*) – the OpenMP schedule to use. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'. Defaults to 'static'.

validate(*node*, *options=None*)

Checks that the supplied node is a valid target for parallelisation using OMP directives.

Parameters

- **node** (`psyclone.psyir.nodes.Loop`) – the candidate loop for parallelising using OMP Do.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises **TransformationError** – if the loop_type of the supplied Loop is not "inner" or "outer".

---

class psyclone.domain.gocean.transformations.**GOConstLoopBoundsTrans**

Use of a common constant variable for each loop bound within a GOInvokeSchedule. By deafault, PSyclone generates loops where the bounds are obtained by de-referencing a field object, e.g.:

```
DO j = my_field%grid%internal%ystart, my_field%grid%internal%ystop
```

Some compilers are able to produce more efficient code if they are provided with information on the relative trip-counts of the loops within an Invoke. With constant loop bounds, PSyclone generates code like:

```
ny = my_field%grid%subdomain%internal%ystop
...
DO j = 1, ny-1
```

In practice, the application of the constant loop bounds transformation looks something like, e.g.:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> import os
>>> TEST_API = "gocean"
>>> _, info = parse(os.path.join("tests", "test_files", "gocean1p0",
...                              "single_invoke.f90"),
...                 api=TEST_API)
>>> psy = PSyFactory(TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0_compute_cu')
>>> schedule = invoke.schedule
>>>
>>> from psyclone.transformations import GOConstLoopBoundsTrans
>>> clbtrans = GOConstLoopBoundsTrans()
>>>
>>> clbtrans.apply(schedule)
>>> print(schedule.view())
```

apply(*node*, *options=None*)

Modify the GOcean kernel loops in a GOInvokeSchedule to use common constant loop bound variables.

Parameters

- **node** (psyclone.gocean1p0.GOInvokeSchedule) – the GOInvokeSchedule of which all loops will get the constant loop bounds.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

property name

Returns the name of the Transformation as a string.

Return type str

validate(*node*, *options=None*)

Checks if it is valid to apply the GOConstLoopBoundsTrans transform.

Parameters

- **node** (psyclone.gocean1p0.GOInvokeSchedule) – the GOInvokeSchedule to transform.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises

- **TransformationError** – if the supplied node is not a GOInvokeSchedule.
- **TransformationError** – if the supplied schedule has loops with a loop with loop_type different than 'inner' or 'outer'.
- **TransformationError** – if the supplied schedule has loops with attributes for index_offsets, field_space, iteration_space and loop_type that don't appear in the GOLoop.bounds_lookup table.
- **TransformationError** – if the supplied schedule doesn't have a field argument.

class psyclone.domain.gocean.transformations.**GOMoveIterationBoundariesInsideKernelTrans**

Provides a transformation that moves iteration boundaries that are encoded in the Loops lower_bound() and upper_bound() methods to a mask inside the kernel with the boundaries passed as kernel arguments.

For example the following kernel call:

```
do i = 2, N - 1
    do j = 2, N - 1
        kernel(i, j, field)
    end do
end do
```

will be transformed to:

```
startx = 2
stopx = N - 1
starty = 2
stopy = N - 1
do i = 1, size(field, 1)
    do j = 1, size(field, 2)
        kernel(i, j, field, startx, stopx, starty, stopy)
    end do
end do
```

additionally a mask like the following one will be introduced in the kernel code:

```
if (i < startx .or. i > stopx .or. j < starty .or. j > stopy) then
    return
end if
```

apply(*node*, *options=None*)

Apply this transformation to the supplied node.

Parameters

- **node** (psyclone.gocean1p0.GOKern) – the node to transform.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

property name

Returns the name of this transformation as a string.

validate(*node*, *options=None*)

Ensure that it is valid to apply this transformation to the supplied node.

Parameters

- **node** (psyclone.gocean1p0.GOKern) – the node to validate.
- **options** (*Optional[Dict[str, Any]]*) – a dictionary with options for transformations.

Raises **TransformationError** – if the node is not a GOKern.

CHAPTER

ELEVEN

PSYCLONE KERNEL TOOLS

In addition to the `psyclone` command, the PSyclone package also provides tools related to generating code purely from kernel metadata. Currently there are two such tools:

1. *Kernel-stub Generator*
2. *Algorithm Generator*

The kernel-stub generator takes a file containing kernel metadata as input and outputs the (Fortran) kernel subroutine arguments and declarations. The word "stub" is used to indicate that it is only the subroutine arguments and their declarations that are generated; the subroutine has no content.

The algorithm generator also takes a file containing a kernel implementation but this time generates an appropriate algorithm layer subroutine. This algorithm layer plus the associated kernel metadata may then be processed with PSyclone in the usual way to generate code which executes the supplied kernel.

This functionality is provided to the user via the `psyclone-kern` command, described in more detail below.

11.1 The psyclone-kern Command

Before using the `psyclone-kern` tool, PSyclone must be installed. If you have not already done so, please follow the instructions for setting up PSyclone in Section *Getting Going*.

PSyclone will be installed in a particular location on your machine, which will be referred to as the `<PSYCLONEINSTALL>` directory. The `psyclone-kern` script comes with the PSyclone installation. A quick check `> which psyclone-kern` should return the location of the `<PSYCLONEINSTALL>/bin` directory.

The `psyclone-kern` command has the following arguments:

```
> psyclone-kern -h
usage: psyclone-kern [-h] [-gen {alg,stub}] [-o OUT_FILE] [-api API]
                     [-I INCLUDE] [-l {off,all,output}]
                     [--config CONFIG] [-v]
                     filename

Run the PSyclone kernel generator on a particular file

positional arguments:
  filename              file containing Kernel metadata

optional arguments:
  -h, --help            show this help message and exit
  -gen {alg,stub}       what to generate for the supplied kernel
```

(continues on next page)

(continued from previous page)

```
                      (alg=algorithm layer, stub=kernel-stub subroutine).
                      Defaults to stub.
-o OUT_FILE           filename for created code.
-api API              choose a particular API from ['lfric',
                      'gocean'].
-I INCLUDE, --include INCLUDE
                      path to Fortran INCLUDE or module files
-l {off,all,output}, --limit {off,all,output}
                      limit the Fortran line length to 132
                      characters (default 'off'). Use 'all' to
                      apply limit to both input and output
                      Fortran. Use 'output' to apply line-length
                      limit to output Fortran only.
--config CONFIG, -c CONFIG
                      config file with PSyclone specific options.
-v, --version         display version information (\ |release|\ )
```

The `-o` option allows the user to specify that the output should be written to a particular file. If this is not specified then the Python `print` statement is used to write to stdout. Typically this results in the output being printed to the terminal.

The `-l`, or `--limit` option utilises the PSyclone support for wrapping of lines within the 132 character limit in the generated Fortran code (please see the *Line Length* chapter for more details).

11.2 Kernel-stub Generator

11.2.1 Quick Start

1) Use an existing Kernel file or create a Kernel file containing a Kernel module with the required metadata and an empty Kernel subroutine with no arguments.

2) Run the following command

```
> psyclone-kern -api lfric -gen stub <PATH>/my_file.f90
```

3) To have the generated code written to file rather than stdout use the *-o* flag

```
> psyclone-kern -api lfric -gen stub -o my_stub_file.f90 ./my_kernel_mod.f90
```

(Since stub generation is the default, the `-gen stub` may be omitted if desired.)

11.2.2 Introduction

PSyclone provides a kernel stub generator for the LFRic API. The kernel stub generator takes a kernel file as input and outputs the kernel subroutine arguments and declarations. The word "stub" is used to indicate that it is only the subroutine arguments and their declarations that are generated; the subroutine has no content.

The primary reason the stub generator is useful is that it generates the correct Kernel subroutine arguments and declarations for the LFRic API as specified by the Kernel metadata. As the number of arguments to Kernel subroutines can become large and the arguments have to follow a particular order, it can become burdensome, and potentially error prone, for the user to have to work out the appropriate argument list if written by hand.

The stub generator can be used when creating a new Kernel. A Kernel can first be written to specify the required metadata and then the generator can be used to create the appropriate (empty) Kernel subroutine. The user can then fill in the content of the subroutine.

The stub generator can also be used to check whether the arguments for an existing Kernel are correct i.e. whether the Kernel subroutine and Kernel metadata are consistent. One example would be where a Kernel is updated resulting in a change to the metadata and subroutine arguments.

The LFRic API requires Kernels to conform to a set of rules which determine the required arguments and types for a particular Kernel. These rules are required as the generated PSy layer needs to know exactly how to call a Kernel. These rules are outlined in Section *Rules*.

Therefore PSyclone has been coded with the LFRic API rules which are then applied when reading the Kernel metadata to produce the required Kernel call and its arguments in the generated PSy layer. These same rules are used by the Kernel stub generator to produce Kernel subroutine stubs, thereby guaranteeing that Kernel calls from the PSy layer and the associated Kernel subroutines are consistent.

11.2.3 Kernels

Any LFRic kernel can be used as input to the stub generator. Example Kernels can be found in the `examples/lfric` repository or, for more simple cases, in the `tests/test_files/dynamo0p3` directory. These directories are located in the <PSYCLONEHOME>`/src/psyclone` directory where <PSYCLONEHOME> refers to the location where you download or clone PSyclone (*Getting Going*).

In the `tests/test_files/dynamo0p3` directory the majority of examples start with `testkern`. Amongst the exceptions are: `testkern_simple_mod.f90`, `ru_kernel_mod.f90` and `matrix_vector_kernel_mod.F90`. The following test kernels can be used to generate kernel stub code (running stub generation from the <PSYCLONEHOME>`/src/psyclone` directory):

```
tests/test_files/dynamo0p3/testkern_chi_read_mod.F90
tests/test_files/dynamo0p3/testkern_coord_w0_mod.F90
tests/test_files/dynamo0p3/testkern_operator_mod.f90
tests/test_files/dynamo0p3/testkern_operator_nofield_mod.f90
tests/test_files/dynamo0p3/ru_kernel_mod.f90
tests/test_files/dynamo0p3/testkern_simple_mod.f90
```

11.2.4 Example

A simple, single field example of a kernel that can be used as input for the stub generator is found in `tests/test_files/dynamo0p3/testkern_simple_mod.f90` and is shown below:

```
module simple_mod

  use argument_mod
  use fs_continuity_mod
  use kernel_mod
  use constants_mod

  implicit none

  type, extends(kernel_type) :: simple_type
    type(arg_type), dimension(1) :: meta_args = &
         (/ arg_type(gh_field, gh_real, gh_inc, w1) /)
```

(continues on next page)

(continued from previous page)

```
    integer :: operates_on = cell_column
  contains
    procedure, nopass :: code => simple_code
  end type simple_type

contains

  subroutine simple_code()
  end subroutine

end module simple_mod
```

Note: The module name `simple_mod` and the type name `simple_type` share the same root `simple` and have the extensions `_mod` and `_type` respectively. This is a convention in LFRic API and is required by the kernel stub generator as it needs to determine the name of the type containing the metadata and infers this by reading the module name. If this rule is not followed the kernel stub generator will return with an error message (see Section *Errors*).

Note: Whilst strictly the kernel stub generator only requires the Kernel metadata to generate the appropriate stub code, the parser that the generator relies on currently requires a dummy kernel subroutine to exist.

If we run the kernel stub generator on the `testkern_simple_mod.f90` example:

```
> psyclone-kern -api lfric -gen stub tests/test_files/dynamo0p3/testkern_simple_mod.f90
```

we get the following kernel stub output:

```
MODULE simple_mod
  IMPLICIT NONE
  CONTAINS
  SUBROUTINE simple_code(nlayers, field_1_w1, ndf_w1, undf_w1, map_w1)
    USE constants_mod, ONLY: r_def, i_def
    IMPLICIT NONE
    INTEGER(KIND=i_def), intent(in) :: nlayers
    INTEGER(KIND=i_def), intent(in) :: ndf_w1
    INTEGER(KIND=i_def), intent(in), dimension(ndf_w1) :: map_w1
    INTEGER(KIND=i_def), intent(in) :: undf_w1
    REAL(KIND=r_def), intent(inout), dimension(undf_w1) :: field_1_w1
  END SUBROUTINE simple_code
END MODULE simple_mod
```

The subroutine content can then be copied into the required module, used as the basis for a new module, or checked with an existing subroutine for correctness.

Note: The output does not currently conform to Met Office coding standards so must be modified accordingly.

Note: The code will not compile without a) providing the `constants_mod`, `argument_mod` and `kernel_mod` modules in the compiler include path and b) adding in code that writes to any arguments declared as intent `out` or `inout`. For

a quick check, the USE declaration and KIND declarations can be removed and the field_1_w1 array can be initialised with some value in the subroutine. At this point the Kernel should compile successfully.

Note: Whilst there is only one field declared in the metadata there are 5 arguments to the Kernel. The first argument nlayers specifies the number of layers in a column for a field. The second argument is the array associated with the field. The field array is dimensioned as the *number of unique degrees of freedom* (hereafter undf) which is also passed into the kernel (the fourth argument). The naming convention is to call each field a field, followed by its position in the (algorithm) argument list (which is reflected in the metadata ordering). The third argument is the number of degrees of freedom for the particular column and is used to dimension the final argument which is the *degrees of freedom map* (dofmap) which indicates the location of the required values in the field array. The naming convention for the dofmap, undf and ndf is to append the name with the space that it is associated with.

We now take a look at a more complicated example. The metadata in this example is the same as an actual LFRic kernel, however the subroutine content and various comments have been removed. The metadata specifies that there are four fields passed by the algorithm layer, the fourth of which is a vector field of size three. All three of the spaces require a basis function and the W0 and W2 function spaces additionally require a differential basis function. The content of the Kernel, excluding the subroutine body, is given below:

```
module ru_kernel_mod

use argument_mod
use fs_continuity_mod
use kernel_mod
use constants_mod

implicit none

private

type, public, extends(kernel_type) :: ru_kernel_type
  private
  type(arg_type) :: meta_args(6) = (/                                    &
       arg_type(GH_FIELD,   GH_REAL,    GH_INC,  W2),                    &
       arg_type(GH_FIELD,   GH_REAL,    GH_READ, W3),                    &
       arg_type(GH_SCALAR,  GH_INTEGER, GH_READ),                        &
       arg_type(GH_SCALAR,  GH_REAL,    GH_READ),                        &
       arg_type(GH_FIELD,   GH_REAL,    GH_READ, W0),                    &
       arg_type(GH_FIELD*3, GH_REAL,    GH_READ, W0)                     &
       /)
  type(func_type) :: meta_funcs(3) = (/                                  &
       func_type(W2, GH_BASIS, GH_DIFF_BASIS),                           &
       func_type(W3, GH_BASIS),                                          &
       func_type(W0, GH_BASIS, GH_DIFF_BASIS)                            &
       /)
  integer :: operates_on = CELL_COLUMN
  integer :: gh_shape = gh_quadrature_XYoZ
contains
  procedure, nopass :: ru_code
end type

public ru_code
```

(continues on next page)

(continued from previous page)

```
contains

  subroutine ru_code()
  end subroutine ru_code

end module ru_kernel_mod
```

If we run the kernel stub generator on this example:

```
> psyclone-kern -api lfric -gen stub tests/test_files/dynamo0p3/ru_kernel_mod.f90
```

we obtain the following output:

```
MODULE ru_mod
  IMPLICIT NONE
  CONTAINS
  SUBROUTINE ru_code(nlayers, field_1_w2, field_2_w3, iscalar_3, rscalar_4, &
                     field_5_w0, field_6_w0_v1, field_6_w0_v2, field_6_w0_v3, &
                     ndf_w2, undf_w2, map_w2, basis_w2_qr_xyoz, &
                     diff_basis_w2_qr_xyoz, ndf_w3, undf_w3, map_w3, &
                     basis_w3_qr_xyoz, ndf_w0, undf_w0, map_w0, &
                     basis_w0_qr_xyoz, diff_basis_w0_qr_xyoz, &
                     np_xy_qr_xyoz, np_z_qr_xyoz, weights_xy_qr_xyoz, weights_
↪z_qr_xyoz)
    USE constants_mod, ONLY: r_def, i_def
    IMPLICIT NONE
    INTEGER(KIND=i_def), intent(in) :: nlayers
    INTEGER(KIND=i_def), intent(in) :: ndf_w0
    INTEGER(KIND=i_def), intent(in), dimension(ndf_w0) :: map_w0
    INTEGER(KIND=i_def), intent(in) :: ndf_w2
    INTEGER(KIND=i_def), intent(in), dimension(ndf_w2) :: map_w2
    INTEGER(KIND=i_def), intent(in) :: ndf_w3
    INTEGER(KIND=i_def), intent(in), dimension(ndf_w3) :: map_w3
    INTEGER(KIND=i_def), intent(in) :: undf_w2, undf_w3, undf_w0
    REAL(KIND=r_def), intent(in) :: rscalar_4
    INTEGER(KIND=i_def), intent(in) :: iscalar_3
    REAL(KIND=r_def), intent(inout), dimension(undf_w2) :: field_1_w2
    REAL(KIND=r_def), intent(in), dimension(undf_w3) :: field_2_w3
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_5_w0
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_6_w0_v1
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_6_w0_v2
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_6_w0_v3
    INTEGER(KIND=i_def), intent(in) :: np_xy_qr_xyoz, np_z_qr_xyoz
    REAL(KIND=r_def), intent(in), dimension(3,ndf_w2,np_xy_qr_xyoz,np_z_qr_
↪xyoz) :: basis_w2_qr_xyoz
    REAL(KIND=r_def), intent(in), dimension(1,ndf_w2,np_xy_qr_xyoz,np_z_qr_
↪xyoz) :: diff_basis_w2_qr_xyoz
    REAL(KIND=r_def), intent(in), dimension(1,ndf_w3,np_xy_qr_xyoz,np_z_qr_
↪xyoz) :: basis_w3_qr_xyoz
    REAL(KIND=r_def), intent(in), dimension(1,ndf_w0,np_xy_qr_xyoz,np_z_qr_
↪xyoz) :: basis_w0_qr_xyoz
```

(continues on next page)

(continued from previous page)

```
    REAL(KIND=r_def), intent(in), dimension(3,ndf_w0,np_xy_qr_xyoz,np_z_qr_
↪xyoz) :: diff_basis_w0_qr_xyoz
    REAL(KIND=r_def), intent(in), dimension(np_xy_qr_xyoz) :: weights_xy_qr_
↪xyoz
    REAL(KIND=r_def), intent(in), dimension(np_z_qr_xyoz) :: weights_z_qr_xyoz
  END SUBROUTINE ru_code
END MODULE ru_mod
```

The above example demonstrates that the argument list can get quite complex. Rather than going through an explanation of each argument you are referred to Section *Rules* for more details on the rules for argument types and argument ordering. Regarding naming conventions for arguments you can see that the arrays associated with the fields are labelled as 1-6 depending on their position in the metadata. For a vector field, each vector results in a different array. These are distinguished by appending `_vx` where `x` is the number of the vector.

The introduction of stencil operations on field arguments further complicates the argument list of a kernel. An example of the use of the stub generator for a kernel that performs stencil operations is provided in `examples/lfric/eg5`:

```
> psyclone-kern -api lfric -gen stub ../../examples/lfric/eg5/conservative_flux_kernel_
↪mod.F90
```

11.2.5 Errors

The stub generator has been written to provide useful errors if mistakes are found. If you run the generator and it does not produce a useful error - and in particular if it produces a stack trace - please contact the PSyclone developers.

The following tests do not produce stub kernel code either because they are invalid or because they contain functionality that is not supported in the stub generator:

```
tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90
tests/test_files/dynamo0p3/testkern_any_space_4_mod.f90
tests/test_files/dynamo0p3/testkern_any_discontinuous_space_op_2_mod.f90
tests/test_files/dynamo0p3/testkern_dofs_mod.f90
tests/test_files/dynamo0p3/testkern_invalid_fortran_mod.f90
tests/test_files/dynamo0p3/testkern_short_name_mod.f90
tests/test_files/dynamo0p3/testkern_no_datatype_mod.f90
tests/test_files/dynamo0p3/testkern_wrong_file_name.F90
```

`testkern_invalid_fortran_mod.f90`, `testkern_no_datatype_mod.f90`, `testkern_short_name_mod.f90` and `testkern_wrong_file_name.F90` are designed to be invalid for PSyclone stub generation testing purposes and should produce appropriate errors. Two examples are below:

```
> psyclone-kern -api lfric -gen stub tests/test_files/dynamo0p3/testkern_invalid_fortran_
↪mod.f90
Error: 'Parse Error: Code appears to be invalid Fortran'

> psyclone-kern -api lfric -gen stub tests/test_files/dynamo0p3/testkern_no_datatype_mod.
↪f90
Error: 'Parse Error: Kernel type testkern_type does not exist'
```

`testkern_dofs_mod.f90` is an example with an unsupported feature, as the `operates_on` metadata specifies `dof`. Currently only kernels with `operates_on=CELL_COLUMN` are supported by the stub generator.

Generic function space metadata `any_space` and `any_discontinuous_space` (see Section *Supported Function Spaces* for function-space identifiers) are currently only supported for *LFRic fields* in the stub genera-

tor. Basis and differential basis functions on these generic function spaces, required for *quadrature* and *evaluators*, are not supported. Hence, `testkern_any_space_1_mod.f90`, `testkern_any_space_4_mod.f90` and `testkern_any_discontinuous_space_op_2_mod.f90` should fail with appropriate warnings because of that. For example:

```
> psyclone-kern -api lfric -gen stub tests/test_files/dynamo0p3/testkern_any_space_1_mod.
↪f90
Error: "Generation Error: Unsupported space for basis function, expecting
one of ['w3', 'wtheta', 'w2v', 'w2vtrace', 'w2broken', 'w0', 'w1', 'w2',
'w2trace', 'w2h', 'w2htrace', 'any_w2', 'wchi'] but found 'any_space_1'"
```

As noted above, if the LFRic API naming convention for module and type names is not followed, the stub generator will return with an error message. For example:

```
> psyclone-kern -api lfric -gen stub tests/test_files/dynamo0p3/testkern_wrong_file_name.
↪F90
Error: "Parse Error: Error, module name 'testkern_wrong_file_name' does not have
'_mod' as an extension. This convention is assumed."
```

11.3 Algorithm Generator

11.3.1 Quick Start

1) Use an existing Kernel file containing a full LFRic kernel implementation.
2) Run the following command

```
> psyclone-kern -api lfric -gen alg <PATH>/my_kern_file_mod.f90
```

3) The generated Algorithm code will be output to stdout by default. To have it written to a file use the *-o* flag.

11.3.2 Introduction

The ability to generate a valid LFRic Algorithm layer that calls a given kernel is useful for a number of reasons:

1) Starting point for creating a test for a kernel;
2) Benchmarking an individual kernel;
3) Constructing a test harness for the adjoint of a kernel produced by PSyAD.

Currently algorithm generation is only supported for the LFRic API but it could be extended to the GOcean API if desired.

Mapping of Function Spaces

Every field or operator argument to an LFRic kernel must have its function space(s) specified in the metadata of the kernel. This information is used by the algorithm generation to ensure that each kernel argument is correctly constructed. However, the metadata permits the use of certain 'generic' function-space specifiers (see *Supported Function Spaces*). If an argument is specified as being on one of these spaces then the algorithm generator chooses an appropriate, specific function space for that argument. e.g. an argument that is specified as being on `ANY_SPACE_<n>` will be constructed on `W0` while one on `ANY_DISCONTINUOUS_SPACE_<n>` will be constructed on `W3`.

11.3.3 Example

If we take the same kernel used in the stub-generation *example* then running

```
> psyclone-kern -api lfric -gen alg tests/test_files/dynamo0p3/testkern_simple_mod.f90
```

gives the following algorithm layer code:

```
module test_alg_mod
  implicit none
  public

contains
  subroutine test_alg(mesh, chi, panel_id)
    use field_mod, only : field_type
    use function_space_mod, only : function_space_type
    use fs_continuity_mod, only : w1
    use function_space_collection_mod, only : function_space_collection
    use mesh_mod, only : mesh_type
    use simple_mod, only : simple_type
    use constants_mod, only : i_def, r_def
    integer(kind=i_def), parameter :: element_order = 1_i_def
    type(mesh_type), pointer, intent(in) :: mesh
    type(field_type), dimension(3), intent(in), optional :: chi
    type(field_type), intent(in), optional :: panel_id
    TYPE(function_space_type), POINTER :: vector_space_w1_ptr
    type(field_type) :: field_1

    vector_space_w1_ptr => function_space_collection % get_fs(mesh, element_
↪order, w1)
    call field_1 % initialise(vector_space=vector_space_w1_ptr, name='field_1')
    call invoke(setval_c(field_1, 1.0_r_def), simple_type(field_1))

  end subroutine test_alg

end module test_alg_mod
```

Note that the generated code implements an Algorithm subroutine that is intended to be called from within an LFRic application that has already setup data structures for the mesh (and, optionally, the *chi* coordinate field and panel ID mapping). Since the *metadata* for the *simple_type* kernel specifies that the field argument is on *W1*, the generated code must ensure that the appropriate function space is set up and used to initialise the field. Once that's done, the interesting part is the *invoke* call:

```
call invoke(setval_c(field_1, 1.0_r_def), &
            simple_type(field_1))
```

(where a line-break has been added for clarity). In this example the *invoke* is for two kernels: the first is a *Built-in* that gives *field_1* the value *1.0* everywhere and the second is the 'simple' kernel itself which is passed the now initialised *field_1*.

This Algorithm code can now be processed by PSyclone in the normal way in order to generate a transformed version plus an associated PSy-layer routine. See *Example 20: Algorithm Generation* for a full example of doing this.

11.3.4 Limitations

- Algorithm generation is only currently supported for the LFRic API.
- All fields are currently set to unity. Obviously the generated algorithm code may be edited to change this.
- The generator does not currently recognise 'special' fields that hold geometry information (such as Chi or the face IDs) and these too will all be initialised to unity. This is the subject of Issue #1708 (although note that the generated code already permits the caller to supply Chi and/or face IDs).
- Kernels with operator arguments are not yet supported.
- Kernels with stencil accesses are not yet supported.

CHAPTER

TWELVE

LINE LENGTH

By default PSyclone will generate Fortran code with no consideration of Fortran line-length limits. As the line-length limit for free-form Fortran is 132 characters, the code that is output may be non-conformant.

Line length is not an issue for many compilers as they allow compiler flags to be set which allow lines longer than the Fortran standard. However this is not the case for all compilers.

PSyclone therefore supports the wrapping of lines within the 132 character limit. The next two sections discuss how this is done when scripting and when working interactively respectively.

12.1 Script

The *psyclone* script provides the -l option to wrap lines. Please see the *Fortran line length* section for more details.

12.2 Interactive

When using PSyclone interactively the line lengths of the input algorithm and Kernel files can be checked by setting the `psyclone.parse.algorithm.parse()` function's `line_length` argument to `True`.

```
>>> from psyclone.parse.algorithm import parse
>>> ast, info = parse("argspec.F90", line_length=True)
```

Similarly the `line_length` argument can be set to `True` if calling the `generator.generate()` function. This function simply passes this argument on to the `psyclone.parse.algorithm.parse()` function.

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("argspec.F90", line_length=True)
```

Line wrapping is performed as a post-processing step, i.e. after the code has been generated. This is done by an instance of the `line_length.FortLineLength` class. For example:

```
>>> from psyclone.generator import generate
>>> from psyclone.line_length import FortLineLength
>>> psy, alg = generate("algspec.f90", line_length=True)
>>> line_length = FortLineLength()
>>> psy_str = line_length.process(str(psy))
>>> print psy_str
>>> alg_str = line_length.process(str(alg))
>>> print alg_str
```

12.3 Limitations

The `line_length.FortLineLength` class is only partially aware of Fortran syntax. This awareness is required so that appropriate continuation characters can be used (for example & at the end of a line and `!$omp&` at the start of a line for OpenMP directives, & at the end of a line for statements and & at the end of a line and & at the beginning of a line for strings).

Whilst statements only require an & at the end of the line when line wrapping with free-form fortran they may optionally also have an & at the beginning of the subsequent line. In contrast, when splitting a string over multiple lines an & is required at both locations. Therefore an instance of the `line_length.FortLineLength` class will always add & at the beginning of a continuation line for a statement, in case the line is split within a string.

One known situation that could cause an instance of the `line_length.FortLineLength` class to fail is when an inline comment is used at the end of a line to make it longer than the 132 character limit. Whilst PSyclone does not generate such code for the PSy-layer, this might occur in Algorithm-layer code, even if the Algorithm-layer code conforms to the 132 line length limit. The reason for this is that PSyclone's internal parser concatenates lines together, thus a long line correctly split with continuation characters in the Algorithm-layer becomes a line that needs to be split by an instance of the `line_length.FortLineLength` class.

CHAPTER

THIRTEEN

FORTRAN NAMING CONVENTIONS

There is a convention in the kernel code for the Dynamo0.3 and GOcean1.0 APIs that if the name of the operation being performed is `<name>` then a kernel file is `<name>_mod.[fF90]`, the name of the module inside the kernel file is `<name>_mod`, the name of the kernel metadata in the module is `<name>_type` and the name of the kernel subroutine in the module is `<name>_code`.

PSyclone itself does not rely on this convention apart from in the stub generator (see the *Kernel-stub Generator* Section) where the name of the metadata to be parsed is determined from the module name.

CHAPTER

FOURTEEN

PSYDATA API

PSyclone provides transformations that will insert callbacks to an external library at runtime. These callbacks allow third-party libraries to access data structures at specified locations in the code. The PSyclone *wrappers* to external libraries are provided in `share/psyclone/lib` in PSyclone *Installation location*. Some example use cases are:

Profiling: By inserting callbacks before and after a region of code, performance measurements can be added. PSyclone provides wrapper libraries for some common performance profiling tools, see *Profiling* for details.

Kernel Data Extraction: PSyclone provides the ability to add callbacks that provide access to all input variables before, and output variables after a kernel invocation. This can be used to automatically create tests for a kernel, or to write a stand-alone driver that just calls one kernel, which can be used for performance tuning. Two example libraries that extract input and output data into either a Fortran binary or a NetCDF file are included with PSyclone (see *Extraction Libraries*).

Access Verification: The callbacks can be used to make sure a field declared as read-only is not modified during a kernel call (either because of an incorrect declaration, or because memory is overwritten). The implementation included in PSyclone uses a simple 64-bit checksum to detect changes to a field (and scalar values). See *Read-Only Verification* for details.

Value Range Check: The callbacks can be used to make sure that all floating point input and output parameters of a kernel are within a user-specified range. Additionally, it will also verify that the values are not a `NaN` (not-a-number) or infinite. See *Value Range Check* for the full description.

In-situ Visualisation: By giving access to output fields of a kernel, an in-situ visualisation library can be used to plot fields while a (PSyclone-processed) application is running. There is no example library available at this stage, but the API has been designed with this application in mind.

The PSyData API should be general enough to allow these and other applications to be developed and used.

PSyclone provides transformations that will insert callbacks to the PSyData API, for example `ProfileTrans`, `GOceanExtractTrans` and `LFRicExtractTrans`. A user can develop additional transformations and corresponding runtime libraries for additional functionality. Refer to psy_data for full details about the PSyData API.

14.1 Read-Only Verification

The PSyData interface is being used to verify that read-only variables in a kernel are not overwritten. The `ReadOnlyVerifyTrans` (in `psyir.transformations.read_only_verify_trans`, or the Transformation Reference Guide) uses the dependency analysis to determine all read-only variables (i.e. arguments declared to be read-only in metadata, most implicit arguments in LFRic, grid properties in GOcean). A simple 64-bit checksum is then computed for all these arguments before a kernel call, and compared with the checksum after the kernel call. Any change in the checksum causes a message to be printed at runtime, e.g.:

```
-------------------------------------
Double precision field b_fld has been modified in main : update
Original checksum:   4611686018427387904
New checksum:        4638355772470722560
-------------------------------------
```

The transformation that adds read-only-verification to an application can be applied for both the *LFRic* and *GOcean API* - no API-specific transformations are required. Below is an example that searches for each loop in a PSyKAl invoke code (which will always surround kernel calls) and applies the transformation to each one. This code has been successfully used as a global transformation with the LFRic Gravity Wave application (the executable is named `gravity_wave`)

```
def trans(psyir):
    from psyclone.psyir.transformations import ReadOnlyVerifyTrans
    from psyclone.psyir.nodes import Loop
    read_only_verify = ReadOnlyVerifyTrans()

    for loop in psyir.walk(Loop):
        read_only_verify.apply(loop)
```

Besides the transformation, a library is required to do the actual verification at runtime. There are two implementations of the read-only-verification library included in PSyclone: one for LFRic, and one for GOcean. Both libraries support the environment variable `PSYDATA_VERBOSE`. This can be used to control how much output is generated by the read-only-verification library at runtime. If the variable is not specified or has the value '0', warnings will only be printed if checksums change. If it is set to '1', a message will be printed before and after each kernel call that is checked. If the variable is set to '2', it will additionally print the name of each variable that is checked.

14.1.1 Read-Only Verification Library for LFRic

This library is contained in `lib/read_only/lfric` and it must be compiled before compiling any LFRic-based application that uses read-only verification. Compiling this library requires access to the LFRic infrastructure library (since it must implement a generic interface for e.g. the LFRic *field* class).

The `Makefile` uses the variable `LFRIC_INF_DIR` to point to the location where LFRic's `field_mod` and `integer_field_mod` have been compiled. It defaults to the path to location of the pared-down LFRic infrastructure located in a clone of PSyclone repository, `<PSYCLONEHOME>/src/psyclone/tests/test_files/dynamo0p3/infrastructure`, but this will certainly need to be changed for any user (for instance with PSyclone installation). The LFRic infrastructure library is not used in linking the verification library. The application which uses the read-only-verification library needs to link in the infrastructure library anyway.

Note: It is the responsibility of the user to make sure that the infrastructure files used during compilation of the read-only-verification library are also used when linking the application. Otherwise strange and non-reproducible crashes might happen.

Compilation of the library is done by invoking `make` and setting the required variables:

```
make LFRIC_INF_DIR=some_path F90=ifort F90FLAGS="--some-flag"
```

This will create a library called `lib_read_only.a`.

An executable example for using the LFRic read-only-verification library is included in `tutorial/practicals/LFRic/building_code/4_psydata` directory, see this link for more information.

14.1.2 Read-Only-Verification Library for GOcean

This library is contained in the `lib/read_only/dl_esm_inf` directory and it must be compiled before linking any GOcean-based application that uses read-only verification. Compiling this library requires access to the GOcean infrastructure library (since it must implement a generic interface for e.g. the dl_esm_inf `r2d_field` class).

The `Makefile` uses the variable `GOCEAN_INF_DIR` to point to the location where dl_esm_inf's `field_mod` has been compiled. It defaults to the relative path to location of the dl_esm_inf version included in PSyclone repository as a Git submodule, `<PSYCLONEHOME>/external/dl_esm_inf/finite_difference/src`. It can be changed to a user-specified location if required (for instance with the PSyclone installation).

The dl_esm_inf library is not used in linking the verification library. The application which uses the read-only-verification library needs to link in the infrastructure library anyway.

Compilation of the library is done by invoking `make` and setting the required variables:

```
make GOCEAN_INF_DIR=some_path F90=ifort F90FLAGS="--some-flag"
```

This will create a library called `lib_read_only.a`. An executable example for using the GOcean read-only-verification library is included in `examples/gocean/eg5/readonly`, see *Example 5.3: Read-only-verification*.

14.2 Value Range Check

This transformation can be used for both LFRic and GOcean APIs. It will test all input and output parameters of a kernel to make sure they are within a user-specified range. Additionally, it will also verify that floating point values are not `NaN` or infinite.

At runtime, environment variables must be specified to indicate which variables are within what expected range, and optionally also at which location. The range is specified as a `:` separated tuple:

```
1.1:3.3   A value between 1.1 and 3.3 (inclusive).
:3.3      A value less than or equal to 3.3
1.1:      A value greater than or equal to 1.1
```

The syntax for the environment variable is one of:

`PSYVERIFY__module__kernel__variable` The specified variable is tested when calling the specified kernel in the specified module.

`PSYVERIFY__module__variable` The specified variable name is tested in all kernel calls of the specified module that are instrumented with the ValueRangeCheckTrans transformation.

`PSYVERIFY__variable` The specified variable name is tested in any instrumented code region.

If the module name or kernel name contains a - (which can be inserted by PSyclone, e.g. *invoke_compute-r1*), it needs to be replaced with an underscore character in the environment variable (_)

An example taken from the LFric tutorial (note that values greater than 4000 are actually valid, the upper limit was just chosen to show a few warnings raised by the value range checker):

```
PSYVERIFY__time_evolution__invoke_initialise_perturbation__perturbation_data=0.0:4000
PSYVERIFY__time_evolution__perturbation_data=0.0:4000
PSYVERIFY__perturbation_data=0.0:4000
```

Warning: Note that while the field variable is called *perturbation*, PSyclone will append *_data* when the LFRic domain is used, so the name becomes *perturbation_data*. You have to use this name in LFRic in order to trigger the value range check. To verify that the tests are done as expected, set the environment variable *PSYDATA_VERBOSE* to 1, which will print which data is taken from the environment variables:

```
PSyData: checking 'time_evolution' region 'invoke_initialise_perturbation' :   0.
↪0000000000000000       <= perturbation_data <=    4000.0000000000000
```

If values outside the specified range are found, appropriate warnings are printed, but the program is not aborted:

```
PSyData: Variable 'perturbation_data' has the value 4227.3587826606408 at index/indices␣
↪27051 in module 'time_evolution', region 'invoke_initialise_perturbation', which is␣
↪not between '0.0000000000000000' and '4000.0000000000000'.
```

The library uses the function `IEEE_IS_FINITE` from the ieee_arithmetic module for additionally verifying that values are not `NAN` or `infinity` for any floating point variable, even if no `PSY_VERIFY...` environment variable is set for this variable. Integer numbers do not have a bit pattern for 'infinity' or `NaN`, so they will only be tested for valid range if a corresponding environment variable is specified.

The runtime libraries for GOcean and LFRic are based on a jinja-template contained in the directory `<PSYCLONEHOME>/lib/value_range_check`. The respective API-specific libraries map the internal field structures to Fortran basic types and call the functions from the base class to handle those.

The relevant libraries for the LFRic and GOcean APIs are contained in the `lib/value_range_check/lfric` and `lib/value_range_check/dl_esm_inf` subdirectories, respectively. For more information on how to build and link these libraries, please refer to the relevant `README.md` files.

14.3 Integrating PSyData Libraries into the LFRic Build Environment

The easiest way of integrating any PSyData-based library into the LFRic build environment is:

- In the LFRic source tree create a new directory under `infrastructure/source`, e.g. `infrastructure/source/psydata`.
- Build the PSyData wrapper stand-alone in `lib/extract/netcdf/lfric` (which will use NetCDF as output format) or `lib/extract/standalone/lfric` (which uses standard Fortran binary output format) by executing `make`. The compiled files will actually not be used, but this step will create all source files (some of which are created by jinja). Do not copy the compiled files into your LFRic build tree, since these files might be compiled with an outdated version of the infrastructure files and be incompatible with files in a current LFRic version.
- Copy all processed source files (`extract_netcdf_base.f90`, `kernel_data_netcdf.f90`, `psy_data_base.f90`, `read_kernel_data_mod.f90`) into `infrastructure/source/psydata`
- Start the LFRic build process as normal. The LFRic build environment will copy the PSyData source files into the working directory and compile them.
- If the PSyData library needs additional include paths (e.g. when using an external profiling tool), add the required paths to `$FFLAGS`.
- If additional libraries are required at link time, add the paths and libraries to `$LDFLAGS`. Alternatively, when a compiler wrapper script is provided by a third-party tool (e.g. the profiling tool TAU provides a script `tau_f90.sh`), either set the environment variable `$FC`, or if this is only required at link time, the variable `$LDMPI` to this compiler wrapper.

Warning: Only one PSyData library can be integrated at a time. Otherwise there will be potentially several modules with the same name (e.g. `psy_data_base`), resulting in errors at compile time.

Note: With the new build system FAB this process might change.

CHAPTER

FIFTEEN

PROFILING

PSyclone has the ability to define regions that can be profiled with various performance measurement tools. The profiling can be enabled automatically using command line parameters like:

```
psyclone --profile kernels ...
```

Or, for finer-grained control, it may be applied via a profiling transformation within a transformation script.

PSyclone can be used with a variety of existing profiling tools. It currently supports dl_timer, TAU, Dr Hook, the NVIDIA GPU profiling tools and it comes with a simple stand-alone timer library. The *PSyData API* (see also the Developer Guide) is utilised to implement wrapper libraries that connect the PSyclone application to the profiling libraries. Certain adjustments to the application's build environment are required:

- The compiler needs to be able to find the module files for the wrapper of the selected profiling library.
- The application needs to be linked with the wrapper library that interfaces between the PSyclone API and the tool-specific API.
- The tool-specific library also needs to be linked in.

It is the responsibility of the user to supply the corresponding compiler command line options when building the application that incorporates the PSyclone-generated code.

15.1 Interface to Third Party Profiling Tools

PSyclone comes with *wrapper libraries* to support usage of TAU, Dr Hook, dl_timer, NVTX (NVIDIA Tools Extension library), and a simple non-thread-safe timing library. Support for further profiling libraries will be added in the future. To compile the wrapper libraries, change into the directory `lib/profiling` of PSyclone and type `make` to compile all wrappers. If only some of the wrappers are required, you can either use `make wrapper-name` (e.g. `make drhook`), or change into the corresponding directory and use `make`. The corresponding `README.md` files contain additional parameters that can be set in order to find third party profiling tools.

Below are short descriptions of each of the various wrapper libraries that come with PSyclone:

`lib/profiling/template` This is a simple library that just prints out the name as regions are entered and exited. It could act as a template to develop new wrapper libraries, hence its name.

`lib/profiling/simple_timing` This is a simple, stand-alone library that uses Fortran system calls to measure the execution time, and reports average, minimum and maximum execution time for all regions. It is not MPI aware (i.e. it will just report independently for each MPI process), and not thread-safe.

`lib/profiling/dl_timer` This wrapper uses the apeg-dl_timer library. In order to use this wrapper, you must download and install the dl_timer library from `https://bitbucket.org/apeg/dl_timer`. This library has various compile-time options and may be built with MPI or OpenMP support. Additional link options might therefore be required (e.g. enabling OpenMP, or linking with MPI).

lib/profiling/tau This wrapper uses TAU profiling and tracing toolkit. It can be downloaded from `https://www.cs.uoregon.edu/research/tau`.

lib/profiling/drhook This wrapper uses the Dr Hook library. You need to contact ECMWF to obtain a copy of Dr Hook.

lib/profiling/nvidia This is a wrapper library that maps the PSyclone profiling API to the NVIDIA Tools Extension library (NVTX). This library is available from `https://developer.nvidia.com/cuda-toolkit`.

lib/profiling/lfric_timer This profile wrapper uses the timer functionality provided by LFRic, and it comes in two different versions:

- `libpsy_lfric_timer.a` This library just contains the PSyData wrapper, but not the actual timer code. It must therefore be linked with the LFRic infrastructure library. It is meant to be used by LFRic only.
- `libpsy_lfric_timer_standalone.a` This library contains the LFRic timer object and its dependencies. It can be used standalone (i.e. without LFRic) with any program. A runnable example using a GOcean code is included in `examples/gocean/eg5/profile`.

The LFRic timer writes its output to a file called `timer.txt` in the current directory, and will overwrite this file if it should already exist.

Any user can create similar wrapper libraries for other profiling tools by providing a corresponding Fortran module. The functions that need to be implemented are described in the developer's guide (psy_data).

Most libraries in `lib/profiling` need to be linked in with the corresponding 3rd party profiling tool, or use a compiler wrapper provided by the tool which will provide the required additional compiler parameters. The exceptions are the template and simple_timing libraries, which are stand alone. The profiling example in `examples/gocean/eg5/profile` can be used with any of the wrapper libraries (except `nvidia`) to see how they work.

15.2 Required Modifications to the Program

In order to guarantee that any profiling library is properly initialised, PSyclone's profiling wrappers utilise two additional function calls that the user must manually insert into the program:

15.2.1 profile_PSyDataInit()

This method needs to be called once to initialise the profiling tool. At this stage this call is not automatically inserted by PSyclone, so it is the responsibility of the user to add the call to an appropriate location in the application:

Listing 15.1: Adding profile_PSyDataInit.

```
use profile_psy_data_mod, only : profile_PSyDataInit
...
call profile_PSyDataInit()
```

The "appropriate" location might depend on the profiling library used. For example, it might be necessary to invoke this before or after a call to `MPI_Init()`.

15.2.2 profile_PSyDataShutdown()

At the end of the program the function `profile_PSyDataShutdown()` must be called. It will make sure that the measurements are printed, files are flushed, and that the profiling tool is closed correctly. Again at this stage it is necessary to manually insert the call at an appropriate location:

Listing 15.2: Adding profile_PSyDataShutdown.

```
use profile_psy_data_mod, only : profile_PSyDataShutdown
...
call profile_PSyDataShutdown()
```

And again the appropriate location might depend on the profiling library used (e.g. before or after a call to `MPI_Finalize()`).

15.3 Profiling Command-Line Options

PSyclone offers two command-line options to automatically instrument code with profiling regions. It can create profile regions around a full invoke routine (including all kernel calls in this invoke), and/or around each individual kernel (for the PSyKAl APIs 'lfric' and 'gocean').

The option `--profile invokes` will automatically add calls to start and end a profile region at the beginning and end of every invoke subroutine created by PSyclone. All kernels called within this invoke subroutine will be included in the profiled region.

The option `--profile routines` is a synonym for 'invokes' but is provided as it is more intuitive for users who are transforming existing code. (In this case, PSyclone will put a profiling region around every routine that it processes.)

The option `--profile kernels` will surround each outer loop created by PSyclone with start and end profiling calls. Note that this option is only available if PSyclone was invoked with a `-api` parameter. If you are only transforming existing code, this option cannot be used as there is no concept of *kernels*.

Note: In some APIs (for example *LFRic* when using distributed memory) additional minor code might get included in a profiled kernel section, for example `setDirty()` calls (expensive calls like `HaloExchange` are excluded).

Note: If the `kernels` option is used in combination with an optimisation script that introduces OpenACC then profiling calls are automatically excluded from within OpenACC regions (since the PSyData wrappers are not compiled for GPU execution).

Note: It is still the responsibility of the user to manually add the calls to `profile_PSyDataInit` and `profile_PSyDataShutdown` to the code base (see *Required Modifications to the Program*).

PSyclone will modify the schedule of each invoke to insert the profiling regions. Below we show an example of a schedule created when instrumenting invokes - all children of a Profile-Node will be part of the profiling region, including all loops created by PSyclone and all kernel calls (note that for brevity, the nodes holding the loop bounds have been omitted for all but the first loop):

Listing 15.3: Instrumenting invokes.

```
GOInvokeSchedule[invoke='invoke_1']
    0: [Profile]
        Schedule[]
            0: Loop[type='outer',field_space='go_cu',it_space='go_internal_pts']
                Literal[value:'2']
                Literal[value:'jstop']
                Literal[value:'1']
                Schedule[]
                    0: Loop[type='inner',field_space='go_cu',
                            it_space='go_internal_pts']
                        ...
                        Schedule[]
                            0: CodedKern compute_unew_code(unew_fld,uold_fld,z_fld,
                                       cv_fld,h_fld,tdt,dy) [module_inline=False]
            1: Loop[type='outer',field_space='cv',it_space='internal_pts']
                ...
                Schedule[]
                    0: Loop[type='inner',field_space='cv',it_space='internal_pts']
                        ...
                        Schedule[]
                            0: CodedKern compute_vnew_code(vnew_fld,vold_fld,z_fld,
                                       cu_fld,h_fld,tdt,dy) [module_inline=False]
            2: Loop[type='outer',field_space='ct',it_space='internal_pts']
                ...
                Schedule[]
                    0: Loop[type='inner',field_space='ct',it_space='internal_pts']
                        ...
                        Schedule[]
                            0: CodedKern compute_pnew_code(pnew_fld,pold_fld,cu_fld,
                                       cv_fld,tdt,dx,dy) [module_inline=False]
```

And now the same schedule when instrumenting kernels. In this case each loop nest and kernel call will be contained in a separate region:

Listing 15.4: Instrumenting kernels.

```
GOInvokeSchedule[invoke='invoke_1']
    0: [Profile]
        Schedule[]
            0: Loop[type='outer',field_space='go_cu',it_space='go_internal_pts']
                ...
                Schedule[]
```

(continues on next page)

(continued from previous page)

```
                    0: Loop[type='inner',field_space='go_cu',
                            it_space='go_internal_pts']
                        ...
                        Schedule[]
                            0: CodedKern compute_unew_code(unew_fld,uold_fld,z_fld,
                                    cv_fld,h_fld,tdt,dy) [module_inline=False]
    1: [Profile]
        Schedule[]
            0: Loop[type='outer',field_space='go_cv',it_space='go_internal_pts']
                ...
                Schedule[]
                        0: Loop[type='inner',field_space='go_cv',
                            it_space='go_internal_pts']
                            ...
                            Schedule[]
                                0: CodedKern compute_vnew_code(vnew_fld,vold_fld,z_fld,
                                    cu_fld,h_fld,tdt,dy) [module_inline=False]
    2: [Profile]
        Schedule[]
            0: Loop[type='outer',field_space='go_ct',it_space='go_internal_pts']
                ...
                Schedule[]
                    0: Loop[type='inner',field_space='go_ct',
                            it_space='go_internal_pts']
                        ...
                        Schedule[]
                            0: CodedKern compute_pnew_code(pnew_fld,pold_fld,
                                    cu_fld,cv_fld,tdt,dx,dy) [module_inline=False]
```

Both options can be specified at the same time:

Listing 15.5: Instrumenting kernels and invokes.

```
GOInvokeSchedule[invoke='invoke_1']
    0: [Profile]
        Schedule[]
            0: [Profile]
                Schedule[]
                    0: Loop[type='outer',field_space='go_cu',
                            it_space='go_internal_pts']
                        ...
                        Schedule[]
                            0: Loop[type='inner',field_space='go_cu',
                                    it_space='go_internal_pts']
                                ...
                                Schedule[]
                                    0: CodedKern compute_unew_code(unew_fld,uold_fld,
                                            ...) [module_inline=False]
            1: [Profile]
                Schedule[]
                    0: Loop[type='outer',field_space='go_cv',
                            it_space='go_internal_pts']
```

(continues on next page)

(continued from previous page)

```
                    ...
                    Schedule[]
                            0: Loop[type='inner',field_space='go_cv',
                                it_space='go_internal_pts']
                                ...
                                Schedule[]
                                    0: CodedKern compute_vnew_code(vnew_fld,vold_fld,
                                        ...) [module_inline=False]
            2: [Profile]
                Schedule[]
                    0: Loop[type='outer',field_space='go_ct',
                            it_space='go_internal_pts']
                    ...
                    Schedule[]
                        0: Loop[type='inner',field_space='go_ct',
                                it_space='go_internal_pts']
                            ...
                            Schedule[]
                                0: CodedKern compute_pnew_code(pnew_fld,pold_fld,
                                        ...) [module_inline=False]
```

15.4 Profiling in Scripts - `ProfileTrans`

The greatest flexibility is achieved by using the profiler transformation explicitly in a transformation script. The script takes either a single PSyIR Node or a list of PSyIR Nodes as argument, and will insert a Profile Node into the PSyIR, with the specified nodes as children. At code creation time the listed children will all be enclosed in one profile region. As an example:

Listing 15.6: Explicitly adding profiling regions.

```
from psyclone.psyir.transformations import ProfileTrans

p_trans = ProfileTrans()
schedule = psy.invokes.get('invoke_0').schedule
print(schedule.view())

# Enclose some children within a single profile region
p_trans.apply(schedule.children[1:3])
print(schedule.view())
```

The profiler transformation also allows the profile name to be set explicitly, rather than being automatically created (see *Naming Profiling Regions* for details). This allows for potentially more intuitive names or finer grain control over profiling (as particular regions could be provided with the same profile names). For example:

Listing 15.7: Setting profile region names.

```
invoke = psy.invokes.invoke_list[0]
schedule = invoke.schedule
profile_trans = ProfileTrans()
# Use the actual PSy-layer module and subroutine names.
options = {"region_name": (psy.name, invoke.name)}
```

(continues on next page)

(continued from previous page)

```
profile_trans.apply(schedule.children, options=options)
# Use own names and repeat for different regions to aggregate profile.
options = {"region_name": ("my_location", "my_region")}
profile_trans.apply(schedule[0].children[1:2], options=options)
profile_trans.apply(schedule[0].children[5:7], options=options)
```

Warning: If "region_name" is misspelt in the options dictionary then the option will be silently ignored. This is true for all options. Issue #613 captures this problem.

Warning: It is the responsibility of the user to make sure that a profile region is only created inside a multi-threaded region if the profiling library used is thread-safe!

15.5 Naming Profiling Regions

A profile region derives its name from two components:

module_name A string identifying the PSy-layer (PSyKAl DSL) or module (existing code) containing this profile node.

region_name A string identifying the invoke (PSyKAl DSL) or routine (existing code) containing this profile node and its location within the invoke/routine (where necessary).

By default PSyclone will generate appropriate names to uniquely determine a particular region. Since those names can be somewhat cryptic, alternative names can be specified by the user when adding profiling via a transformation script, see Passing Parameters From the User to the Node Constructor.

The automatic name generation depends on whether you are using a PSyKAl DSL or only the transformation capabilities of PSyclone. If you are transforming existing code:

- the `module_name` string is set to the module which contains the current code. If there is no module (e.g. a stand-alone subroutine), the subroutine name is used instead. This name is unique as Fortran requires these names to be unique within a program.
- the `region_name` is set to the name of the subroutine, followed by an `r` (standing for region) followed by an integer which uniquely identifies the profile within the parent function/subroutine/program (based on the profile node's position in the PSyIR representation relative to any other profile nodes). If there is no module name (which means the `module_name` is already set to the subroutine name), only the `r` followed by an integer number is specified.

  Example:

Listing 15.8: Profiling names used when transforming existing code.

```
! If the subroutine tra_adv is contained in module tra_adv_mod:
CALL profile_psy_data % PreStart("tra_adv_mod", "tra_adv-r0", 0, 0)
! If the subroutinetra_adv is not contained in a module:
CALL profile_psy_data % PreStart("tra_adv", "r0", 0, 0)
```

For the *LFRic* and *GOcean* APIs:

- the `module_name` string is set to the module name of the generated PSy-layer. This name should be unique by design (otherwise module names would clash when compiling).

- the `region_name` is set to the name of the invoke in which it resides, followed by a `-` and a kernel name if the profile region contains a single kernel, and is completed by `-r` (standing for region) followed by an integer which uniquely identifies the profile within the invoke (based on the profile node's position in the PSyIR representation relative to any other profile nodes). For example:

Listing 15.9: PSyIR with profiling nodes.

```
InvokeSchedule[invoke='invoke_0', dm=True]
  0: Profile[]
      Schedule[]
          0: Profile[]
              Schedule[]
                  0: HaloExchange[field='f2', type='region', depth=1,
                                  check_dirty=True]
                  1: HaloExchange[field='m1', type='region', depth=1,
                                  check_dirty=True]
                  2: HaloExchange[field='m2', type='region', depth=1,
                                  check_dirty=True]
          1: Profile[]
              Schedule[]
                  0: Loop[type='', field_space='w1', it_space='cells',
                          upper_bound='cell_halo(1)']
                      Literal[value:'1', DataType.INTEGER]
                      Literal[value:'mesh%get_last_halo_cell(1)',
                              DataType.INTEGER]
                      Literal[value:'1', DataType.INTEGER]
                      Schedule[]
                          0: CodedKern testkern_code(a,f1,f2,m1,m2)
                             [module_inline=False]
```

This is the code created for this example:

Listing 15.10: Created Fortran source code with profiling regions.

```
 MODULE container
  CONTAINS
  SUBROUTINE invoke_0(a, f1, f2, m1, m2, istp, qr)
    ...
    CALL psy_data_2%PreStart("multi_functions_multi_invokes_psy", "invoke_0-
↪r0", &
                             0, 0)
    CALL psy_data%PreStart("multi_functions_multi_invokes_psy", "invoke_0-r1
↪", 0, 0)
    IF (f2_proxy%is_dirty(depth=1)) THEN
      CALL f2_proxy%halo_exchange(depth=1)
    END IF
    IF (m1_proxy%is_dirty(depth=1)) THEN
      CALL m1_proxy%halo_exchange(depth=1)
    END IF
    IF (m2_proxy%is_dirty(depth=1)) THEN
      CALL m2_proxy%halo_exchange(depth=1)
    END IF
    CALL psy_data%PreEnd()
    CALL psy_data_1%PreStart("multi_functions_multi_invokes_psy", "invoke_0-
↪r2", &
```

(continues on next page)

(continued from previous page)

```
                                0, 0)
    DO cell=1,mesh%get_last_halo_cell(1)
      CALL testkern_code(...)
    END DO

    CALL psy_data_1%PostEnd()
    ...
    DO cell=1,mesh%get_last_halo_cell(1)
      CALL testkern_qr_code(...)
    END DO
    ...
    CALL psy_data_2%PostEnd()
    ...
  END SUBROUTINE invoke_0
END MODULE container
```

CHAPTER

SIXTEEN

PSY KERNEL EXTRACTOR (PSYKE)

16.1 Introduction

PSyclone has the ability to define regions of a PSyclone-conformant code to be extracted and run as a stand-alone application. This ability, called PSyKE (PSy Kernel Extractor), can be useful for benchmarking parts of a model, such as LFRic, without the need for using its infrastructure.

16.1.1 Mechanism

The code marked for extraction can be (subject to *Restrictions*):

- One or more Nodes in an Invoke (e.g. Loops containing Kernel or Built-In calls, a Directive enclosing one or more Loops) or
- The entire Invoke (extraction applied to all Nodes).

The basic mechanism of code extraction is through applying the `ExtractTrans` transformation to selected Nodes. This transformation is further sub-classed into API-specific implementations, `LFRicExtractTrans` and `GOceanExtractTrans`. Both sub-classed transformations insert an instance of the `ExtractNode` object into the Schedule of a specific Invoke. All Nodes marked for extraction become children of the `ExtractNode`.

The `ExtractNode` class uses the dependency analysis to detect which variables are input-, and which ones are output-parameters. The lists of variables are then passed to the `PSyDataNode`, which is the base class of any `ExtractNode` (details of the `PSyDataNode` can be found in psy_data). This node then creates the actual code, as in the following LFRic example:

```
! ExtractStart
!
CALL extract_psy_data%PreStart("testkern_mod", "testkern_code", 4, 2)
CALL extract_psy_data%PreDeclareVariable("a", a)
CALL extract_psy_data%PreDeclareVariable("f2", f2)
CALL extract_psy_data%PreDeclareVariable("m1", m1)
CALL extract_psy_data%PreDeclareVariable("m2", m2)
CALL extract_psy_data%PreDeclareVariable("map_w1", map_w1)
...
CALL extract_psy_data%PreDeclareVariable("undf_w3", undf_w3)
CALL extract_psy_data%PreDeclareVariable("f1_post", f1)
CALL extract_psy_data%PreDeclareVariable("cell_post", cell)
CALL extract_psy_data%PreEndDeclaration
CALL extract_psy_data%ProvideVariable("a", a)
CALL extract_psy_data%ProvideVariable("f2", f2)
CALL extract_psy_data%ProvideVariable("m1", m1)
```

(continues on next page)

(continued from previous page)

```
CALL extract_psy_data%ProvideVariable("m2", m2)
CALL extract_psy_data%ProvideVariable("map_w1", map_w1)
...
CALL extract_psy_data%ProvideVariable("undf_w3", undf_w3)
CALL extract_psy_data%PreEnd
DO cell=1,f1_proxy%vspace%get_ncell()
  !
  CALL testkern_code(nlayers, a, f1_proxy%data, f2_proxy%data,  &
       m1_proxy%data, m2_proxy%data, ndf_w1, undf_w1,           &
       map_w1(:,cell), ndf_w2, undf_w2, map_w2(:,cell), ndf_w3, &
       undf_w3, map_w3(:,cell))
END DO
CALL extract_psy_data%PostStart
CALL extract_psy_data%ProvideVariable("cell_post", cell)
CALL extract_psy_data%ProvideVariable("f1_post", f1)
CALL extract_psy_data%PostEnd
!
! ExtractEnd
```

The *PSyData API* relies on generic Fortran interfaces to provide the field-type-specific implementations of the `ProvideVariable` for different types. This means that a different version of the external PSyData library that PSyKE uses must be supplied for each PSyclone API.

16.1.2 Restrictions

Code extraction can be applied to unoptimised or optimised code. There are restrictions that check for correctness of optimising transformations when extraction is applied, as well as restrictions that eliminate dependence on the specific model infrastructure.

General

This group of restrictions is enforced irrespective of whether optimisations are used or not.

- Extraction can be applied to a single Node or a list of Nodes in a Schedule. For the latter, Nodes in the list must be consecutive children of the same parent Schedule.
- Extraction cannot be applied to an `ExtractNode` or a Node list that already contains one (otherwise we would have an extract region within another extract region).
- A Kernel or a Built-In call cannot be extracted without its parent Loop.

Distributed memory

Kernel extraction for distributed memory is supported in as much as each process will write its own output file by adding its rank to the output file name. So each kernel and each rank will produce one file. It is possible to extract several consecutive kernels, but there must be no halo exchange calls between the kernels. The extraction transformation will test for this and raise an exception if this should happen. The compiled driver program accepts the name of the extracted kernel file as a command line parameter. If this is not specified, it will use the default name (`module-region` without a rank).

Shared memory and API-specific

The `ExtractTrans` transformation cannot be applied to:

- A Loop without its parent Directive,
- An orphaned Directive (e.g. `OMPDoDirective`, `ACCLoopDirective`) without its parent Directive (e.g. ACC or OMP Parallel Directive),
- A Loop over cells in a colour without its parent Loop over colours in the LFRic API,
- An inner Loop without its parent outer Loop in the GOcean API.
- Kernels that have a halo exchange call between them.

16.2 Use

The code extraction is currently enabled by utilising a transformation script (see *Script* section for more details).

For example, the transformation script which extracts the first Kernel call in LFRic API test example `15.1.2_builtin_and_normal_kernel_invoke.f90` would be written as:

```
from psyclone.domain.lfric.transformations import LFRicExtractTrans

# Get instance of the ExtractRegionTrans transformation
etrans = LFRicExtractTrans()

# Get Invoke and its Schedule
invoke = psy.invokes.get("invoke_0")
schedule = invoke.schedule

# Apply extract transformation to the selected Node
etrans.apply(schedule.children[2])
print(schedule.view())
```

and called as:

```
> psyclone -nodm -s ./extract_single_node.py \
    <path-to-example>/15.1.2_builtin_and_normal_kernel_invoke.f90
```

PSyclone modifies the Schedule of the selected `invoke_0`:

```
Schedule[invoke='invoke_0' dm=False]
    0: Loop[type='dofs',field_space='any_space_1',it_space='dofs',
            upper_bound='ndofs']
        Literal[value:'NOT_INITIALISED']
        Literal[value:'NOT_INITIALISED']
        Literal[value:'1']
        Schedule[]
            0: BuiltIn setval_c(f5,0.0)
    1: Loop[type='dofs',field_space='any_space_1',it_space='dofs',
            upper_bound='ndofs']
        ...
        Schedule[]
            0: BuiltIn setval_c(f2,0.0)
```

(continues on next page)

(continued from previous page)

```
    2: Loop[type='',field_space='w2',it_space='cells', upper_bound='ncells']
        ...
        Schedule[]
            0: CodedKern testkern_code_w2_only(f3,f2) [module_inline=False]
    3: Loop[type='',field_space='wtheta',it_space='cells', upper_bound='ncells']
        ...
        Schedule[]
            0: CodedKern testkern_wtheta_code(f4,f5) [module_inline=False]
    4: Loop[type='',field_space='w1',it_space='cells', upper_bound='ncells']
        ...
        Schedule[]
            0: CodedKern testkern_code(scalar,f1,f2,f3,f4) [module_inline=False]
```

to insert the extract region. As shown below, all children of an `ExtractNode` will be part of the region:

```
Schedule[invoke='invoke_0' dm=False]
    0: Loop[type='dofs',field_space='any_space_1',it_space='dofs',
            upper_bound='ndofs']
        ...
        Schedule[]
            0: BuiltIn setval_c(f5,0.0)
    1: Loop[type='dofs',field_space='any_space_1',it_space='dofs',
            upper_bound='ndofs']
        ...
        Schedule[]
            0: BuiltIn setval_c(f2,0.0)
    2: Extract
        Schedule[]
            0: Loop[type='',field_space='w2',it_space='cells', upper_bound='ncells']
                ...
                Schedule[]
                    0: CodedKern testkern_code_w2_only(f3,f2) [module_inline=False]
    3: Loop[type='',field_space='wtheta',it_space='cells', upper_bound='ncells']
        ...
        Schedule[]
            0: CodedKern testkern_wtheta_code(f4,f5) [module_inline=False]
    4: Loop[type='',field_space='w1',it_space='cells', upper_bound='ncells']
        ...
        Schedule[]
            0: CodedKern testkern_code(scalar,f1,f2,f3,f4) [module_inline=False]
```

To extract multiple Nodes, `ExtractTrans` can be applied to the list of Nodes (subject to *General* restrictions above):

```
# Apply extract transformation to the selected Nodes
etrans.apply(schedule.children[1:3])
```

This modifies the above Schedule as:

```
...
    Extract
        Schedule[]
            0: Loop[type='dofs',field_space='any_space_1',it_space='dofs',
                    upper_bound='ndofs']
```

(continues on next page)

(continued from previous page)

```
                ...
                Schedule[]
                    0: BuiltIn setval_c(f2,0.0)
            1: Loop[type='',field_space='w2',it_space='cells', upper_bound='ncells']
                ...
                Schedule[]
                    0: CodedKern testkern_code_w2_only(f3,f2) [module_inline=False]
...
```

As said above, extraction can be performed on optimised code. For example, the following example transformation script first adds `!$OMP PARALLEL DO` directive and then extracts the optimised code in LFRic API test example `15.1.2_builtin_and_normal_kernel_invoke.f90`:

```
from psyclone.domain.lfric.transformations import LFRicExtractTrans
from psyclone.transformations import DynamoOMPParallelLoopTrans

# Get instances of the transformations
etrans = LFRicExtractTrans()
otrans = DynamoOMPParallelLoopTrans()

# Get Invoke and its Schedule
invoke = psy.invokes.get("invoke_0")
schedule = invoke.schedule

# Add OMP PARALLEL DO directives
otrans.apply(schedule.children[1])
otrans.apply(schedule.children[2])
# Apply extract transformation to the selected Nodes
etrans.apply(schedule.children[1:3])
print(schedule.view())
```

The generated code is now:

```
! ExtractStart
CALL extract_psy_data%PreStart("unknown-module", "setval_c", 0, 4)
CALL extract_psy_data%PreDeclareVariable("cell_post", cell)
CALL extract_psy_data%PreDeclareVariable("df_post", df)
CALL extract_psy_data%PreDeclareVariable("f2_post", f2)
CALL extract_psy_data%PreDeclareVariable("f3_post", f3)
...
CALL extract_psy_data%PreEndDeclaration
...
CALL extract_psy_data%PreEnd
!
!$omp parallel do default(shared), private(df), schedule(static)
DO df=1,undf_aspc1_f2
  f2_proxy%data(df) = 0.0
END DO
!$omp end parallel do
!$omp parallel do default(shared), private(cell), schedule(static)
DO cell=1,f3_proxy%vspace%get_ncell()
  !
```

(continues on next page)

(continued from previous page)

```
  CALL testkern_code_w2_only(nlayers, f3_proxy%data, f2_proxy%data, ndf_w2, undf_w2, map_
↪w2(:,cell))
END DO
!$omp end parallel do
CALL extract_psy_data%PostStart
CALL extract_psy_data%ProvideVariable("cell_post", cell)
CALL extract_psy_data%ProvideVariable("df_post", df)
CALL extract_psy_data%ProvideVariable("f2_post", f2)
CALL extract_psy_data%ProvideVariable("f3_post", f3)
CALL extract_psy_data%PostEnd
!
! ExtractEnd
```

Examples in `examples/lfric/eg12` directory demonstrate how to apply code extraction by utilising PSyclone transformation scripts (see *Examples* section for more information). The code in `examples/lfric/eg17/full_example_extract` can be compiled and run, and it will create two kernel data files.

16.3 Extraction Libraries

PSyclone comes with two extraction libraries: one is based on NetCDF and will create NetCDF files to contain all input- and output-parameters. The second one is a stand-alone library which uses only standard Fortran IO to write and read kernel data. The binary files produced using this library may not be portable between machines and compilers. If you require such portability then please use the NetCDF extraction library.

The two extraction *libraries* are in lib/extract/standalone. and in lib/extract/netcdf.

All versions of the extraction libraries can be compiled with MPI support by setting the variable `MPI=yes`:

```
make MPI=yes ...
```

The only difference is that the output files will now have the process rank in the name. The compiled driver program accepts the name of the extracted kernel file as a command line parameter. If this is not specified, it will use the default name (`module-region` without a rank).

16.3.1 Extraction for GOcean

The extraction libraries in lib/extract/standalone/dl_esm_inf and lib/extract/netcdf/dl_esm_inf implement the full PSyData API for use with the *GOcean* dl_esm_inf infrastructure library. When running the instrumented executable, it will create either a binary or a NetCDF file for each instrumented code region. It includes all variables that are read before the code is executed, and all variables that have been modified. The output variables have the postfix `_post` attached to the names, e.g. a variable `xyz` that is read and written will be stored with the name `xyz` containing the input values, and the name `xyz_post` containing the output values. Arrays have their size explicitly stored (in case of NetCDF as dimensions): again the variable `xyz` will have its sizes stored as `xyzdim1`, `xyzdim2` for the input values, and output arrays use the name `xyz_postdim1`, `xyz_postdim2`.

Note: The stand-alone library does not store the names of the variables in the output file, but these names will be used as variable names in the created driver.

The output file contains the values of all variables used in the subroutine. The `GOceanExtractTrans` transformation can automatically create a driver program which will read the corresponding output file, call the instrumented region,

and compare the results. In order to create this driver program, the options parameter `create_driver` must be set to true:

```
extract = GOceanExtractTrans()
extract.apply(schedule.children,
              {"create_driver": True,
               "region_name": ("main", "init")})
```

This will create a Fortran file called `driver-main-init.f90`, which can then be compiled and executed. This stand-alone program will read the output file created during an execution of the actual program, call the kernel with all required input parameter, and compare the output variables with the original output variables. This can be used to create stand-alone test cases to reproduce a bug, or for performance optimisation of a stand-alone kernel.

Warning: Care has to be taken that the driver matches the version of the code that was used to create the output file, otherwise the driver will likely crash. The stand-alone driver relies on a strict ordering of variable values in the output file and e.g. even renaming one variable can affect this. The NetCDF version stores the variable names and will not be able to find a variable if its name has changed.

16.3.2 Extraction for LFRic

The libraries in lib/extract/standalone/lfric and lib/extract/netcdf/lfric implement the full PSyData API for use with the *LFRic* infrastructure library. When running the code, it will create an output file for each instrumented code region. The same logic for naming variables (using `_post` for output variables) used in *Extraction for GOcean* is used here.

Check *Integrating PSyData Libraries into the LFRic Build Environment* for the recommended way of linking an extraction library to LFRic.

The output file contains the values of all variables used in the subroutine. The `LFRicExtractTrans` transformation can automatically create a driver program which will read the corresponding output file, call the instrumented region, and compare the results. In order to create this driver program, the options parameter `create_driver` must be set to true:

```
extract = LFRicExtractTrans()
extract.apply(schedule.children,
              {"create_driver": True,
               "region_name": ("main", "init")})
```

This will create a Fortran file called `driver-main-init.F90`, which can then be compiled and executed. This stand-alone program will read the output file created during an execution of the actual program, call the kernel with all required input parameter, and compare the output variables with the original output variables. This can be used to create stand-alone test cases to reproduce a bug, or for performance optimisation of a stand-alone kernel.

Warning: Care has to be taken that the driver matches the version of the code that was used to create the output file, otherwise the driver will likely crash. The stand-alone driver relies on a strict ordering of variable values in the output file and e.g. even renaming one variable can affect this. The NetCDF version stores the variable names and will not be able to find a variable if its name has changed.

The LFRic kernel driver will inline all required external modules into the driver. It uses a `ModuleManager` to find the required modules, based on the assumption that a file `my_special_mod.f90` will define exactly one module called `my_special_mod` (the `_mod` is required to be part of the filename). The driver creator will sort the modules in the appropriate order and add the source code directly into the driver. As a result, the driver program is truly stand-alone and does not need any external dependency (the only exception being NetCDF if the NetCDF-based extraction library is

used). The `ModuleManager` uses all kernel search paths specified on the command line (see `-d` option in *The psyclone command*), and it will recursively search for all files under each path specified on the command line.

Therefore, compilation for a created driver, e.g. the one created in `examples/lfric/eg17/full_example_extract`, is simple:

```
$ gfortran -g -O0 driver-main-update.F90 -o driver-main-update
$ ./driver-main-update
    Variable      max_abs      max_rel      l2_diff       l2_cos    identical    #rel<1E-
↪9    #rel<1E-6    #rel<1E-3
        cell .0000000E+00 .0000000E+00 .0000000E+00 .1000000E+01 .1000000E+01 .
↪0000000E+00 .0000000E+00 .0000000E+00
 field1_data .0000000E+00 .0000000E+00 .0000000E+00 .1000000E+01 .5390000E+03 .
↪0000000E+00 .0000000E+00 .0000000E+00
  dummy_var1 .0000000E+00 .0000000E+00 .0000000E+00 .1000000E+01 .1000000E+01 .
↪0000000E+00 .0000000E+00 .0000000E+00
```

(see *Driver Summary Statistics* for details about the statistics`). Note that the Makefile in the example will actually provide additional include paths (infrastructure files and extraction library) for the compiler, but these flags are actually only required for compiling the example program, not for the driver.

Restrictions of Kernel Extraction and Driver Creation

A few restrictions still apply to the current implementation of the driver creation code:

- Distributed memory is not yet supported. See #1992.
- The extraction code will now write variables that are used from other modules to the kernel data file, and the driver will read these values in. Unfortunately, if a variable is used that is defined as private, the value cannot be written to the file, and compilation will abort. The only solution is to modify this file and make all variables public. This mostly affects `log_mod.F90`, but a few other modules as well.
- The new build system FAB will be able to remove `private` and `protected` declarations in any source files, meaning no manual modification of files is required anymore (TODO #2536).

16.3.3 Extraction for generic Fortran

The libraries in lib/extract/standalone/generic and lib/extract/netcdf/generic implement the full PSyData API for use with generic code transformation. When running the code, it will create an output file for each instrumented code region. The same logic for naming variables used in *Extraction for GOcean* is used here.

Note: Driver creation for generic Fortran is not yet supported, and is tracked in issue #2058.

16.4 Driver Summary Statistics

When a driver is executed, it will print summary statistics at the end for each variable that was modified, indicating the difference between the *original* values from when the data file was created, and the *new* ones computed when executing the kernel. These differences can be caused by changing the compilation options, or compiler version. Example output:

```
   Variable      max_abs      max_rel      l2_diff       l2_cos    identical    #rel<1E-
↪9    #rel<1E-6    #rel<1E-3
       cell .0000000E+00 .0000000E+00 .0000000E+00 .1000000E+01 .1000000E+01 .
↪0000000E+00 .0000000E+00 .0000000E+00
field1_data .0000000E+00 .0000000E+00 .0000000E+00 .1000000E+01 .5390000E+03 .
↪0000000E+00 .0000000E+00 .0000000E+00
 dummy_var1 .0000000E+00 .0000000E+00 .0000000E+00 .1000000E+01 .1000000E+01 .
↪0000000E+00 .0000000E+00 .0000000E+00
```

The columns from left to right are:

- The variable name.
- The maximum absolute error of all elements.
- The maximum relative error of all elements. If an element has the value 0, the relative error for this element is considered to be 1.0.
- The L2 difference: $\sqrt{\sum (original - new)^2}$.
- The cosine of the angle between the two vectors: $\frac{\sum original*new}{\sqrt{\sum original*original}*\sqrt{\sum new*new}}$.
- How many values are identical.
- How many values have a relative error of less than 10^{-9} but are not identical.
- How many values have a relative error of less than 10^{-6} but more than 10^{-9}.
- How many values have a relative error of less than 10^{-3} but more than 10^{-6}.

Note: The usefulness of the columns printed is still being evaluated. Early indications are that the cosine of the angle between the two vectors, which is commonly used in AI, might not be sensitive enough to give a good indication of the differences.

BIBLIOGRAPHY

[AFH+19] S. V. Adams, R. W. Ford, M. Hambley, J. M. Hobson, I. Kavčič, C. M. Maynard, T. Melvin, E. H. Müller, S. Mullerworth, A. R. Porter, M. Rezny, B. J. Shipway, and R. Wong. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. *Journal of Parallel and Distributed Computing*, 132:383–396, 2019. doi:https://doi.org/10.1016/j.jpdc.2019.02.007.

INDEX

T

U

W