PSyclone Documentation

Release 2.1.0

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

Nov 12, 2021

TABLE OF CONTENTS

1 Introduction **1**

2 Getting Going **3**
- 2.1 Download . . . 3
- 2.2 Environment . . . 3
- 2.3 Dependencies . . . 5
- 2.4 Configuration . . . 7
- 2.5 Test . . . 8
- 2.6 Run . . . 8

3 The psyclone command **11**
- 3.1 Running . . . 11
- 3.2 Basic Use . . . 12
- 3.3 Choosing the API . . . 12
- 3.4 File output . . . 13
- 3.5 Algorithm files with no invokes . . . 13
- 3.6 Kernel search directory . . . 13
- 3.7 Transformation script . . . 14
- 3.8 Fortran line length . . . 14
- 3.9 Distributed memory . . . 14
- 3.10 Automatic Profiling Instrumentation . . . 15
- 3.11 Outputting of Transformed Kernels . . . 15
- 3.12 Fortran INCLUDE Files . . . 15

4 Tutorial **17**

5 Examples **19**
- 5.1 Compilation . . . 19
- 5.2 GOcean . . . 21
- 5.3 LFRic . . . 23
- 5.4 NEMO . . . 27
- 5.5 PSyIR . . . 28

6 Libraries **29**
- 6.1 Available libraries . . . 29
- 6.2 Dependencies . . . 30
- 6.3 Compilation . . . 30

7 System-specific Set-up for Users **33**
- 7.1 Installing dependencies . . . 33
- 7.2 Installing PSyclone . . . 34

7.3 Common installation . . . 34

8 Kernel layer **35**
8.1 API . . . 35
8.2 Metadata . . . 36

9 Algorithm layer **39**
9.1 API . . . 39
9.2 Limitations . . . 41

10 PSy layer **43**
10.1 Code Generation . . . 43
10.2 API . . . 44

11 Built-ins **49**
11.1 Example . . . 49
11.2 Supported Built-in operations . . . 53
11.3 Adding new Built-in operations . . . 53

12 LFRic (Dynamo0.3) API **55**
12.1 Algorithm . . . 55
12.2 PSy-layer . . . 60
12.3 Kernel . . . 60
12.4 Built-ins . . . 82
12.5 Boundary Conditions . . . 100
12.6 Conventions . . . 101
12.7 Configuration . . . 101
12.8 Transformations . . . 103

13 GOcean1.0 API **105**
13.1 Introduction . . . 105
13.2 The GOcean Library . . . 105
13.3 Algorithm . . . 109
13.4 Kernel . . . 110
13.5 Built-ins . . . 116
13.6 Conventions . . . 116
13.7 Configuration . . . 117
13.8 Transformations . . . 119

14 NEMO API **121**
14.1 Algorithm . . . 121
14.2 Constructing the PSyIR . . . 121
14.3 Example . . . 122
14.4 Transformations . . . 123
14.5 Limitations . . . 123

15 PSyIR : The PSyclone Internal Representation **125**
15.1 PSyIR Nodes . . . 125
15.2 Node Descriptions . . . 126
15.3 Text Representation . . . 127
15.4 Tree Navigation . . . 128
15.5 DataTypes . . . 130
15.6 Symbols and Symbol Tables . . . 132
15.7 Creating PSyIR . . . 134
15.8 Modifying the PSyIR . . . 136

16 Transformations **139**
16.1 Finding . . . 139
16.2 Standard Functionality . . . 140
16.3 Available transformations . . . 140
16.4 Kernels . . . 152
16.5 Applying . . . 155
16.6 OpenMP . . . 158
16.7 OpenCL . . . 159
16.8 OpenACC . . . 161
16.9 SIR . . . 162

17 Distributed Memory **163**
17.1 Correctness . . . 163
17.2 Performance . . . 163
17.3 Implementation . . . 164
17.4 Control . . . 164
17.5 Status . . . 164

18 Stub Generation **165**
18.1 Quick Start . . . 165
18.2 Introduction . . . 165
18.3 Use . . . 166
18.4 Kernels . . . 166
18.5 Example . . . 167
18.6 Errors . . . 170

19 Line length **173**
19.1 Script . . . 173
19.2 Interactive . . . 173
19.3 Limitations . . . 174

20 Fortran Naming Conventions **175**

21 API **177**
21.1 The generator module . . . 177
21.2 The parse module . . . 178
21.3 The transformations module . . . 179
21.4 The psyGen module . . . 179
21.5 The alg_gen module . . . 180
21.6 The line_length module . . . 180

22 PSyData API **183**
22.1 Read-Only Verification . . . 183
22.2 NAN Test . . . 185

23 Profiling **187**
23.1 Interface to Third Party Profiling Tools . . . 187
23.2 Required Modifications to the Program . . . 188
23.3 Profiling Command-Line Options . . . 189
23.4 Profiling in Scripts - `ProfileTrans` . . . 191
23.5 Naming Profiling Regions . . . 192

24 PSy Kernel Extractor (PSyKE) **195**
24.1 Introduction . . . 195
24.2 Use . . . 197

24.3 NetCDF Extraction Examples 200

25 Configuration **203**
25.1 Options 203

Bibliography **209**

Python Module Index **211**

Index **213**

CHAPTER

ONE

INTRODUCTION

PSyclone, the PSy code generator, is being developed for use in finite element, finite volume and finite difference codes. PSyclone development started with the aim to support the emerging API in the GungHo project for a finite element dynamical core.

The GungHo project was initiated in 2011 to address challenges of weather and climate prediction on the next generation of supercomputers. The project ran for 5 years as a collaboration between the Met Office, NERC (via NERC funded academics) and STFC. It laid a foundation for redesign of the heart of the Met Office's Unified Model, known as the dynamical core, from the choices of numerical methods and model grids to the implementation of parallel algorithms that will scale to millions of cores.

The software infrastructure based on the GungHo project recommendations is now being developed in the LFRic project and is expected to is expected to start preparations for operational deployment in 2025. Its development is led by the requirements to support multiple meshes and element types, thus allowing for future model development.

GungHo also proposed a novel separation of concerns for the software implementation of the dynamical core. This approach distinguishes between three layers: the Algorithm layer, the Kernel layer and the Parallelisation System (PSy) layer. Together this separation is termed PSyKAl.

The Algorithm layer specifies the algorithm that the scientist would like to run (in terms of calls to kernel routines and built-in operations) and logically operates on full fields.

The Kernel layer provides the implementation of the code kernels as subroutines. These subroutines operate on local fields (a set of elements, a vertical column, or a set of vertical columns, depending on the kernel).

The PSy layer sits in-between the algorithm and kernel layers and its primary role is to provide node-based parallel performance for the target architecture. The PSy layer can be optimised for a particular hardware architecture, such as multi-core, many-core, GPGPUs, or some combination thereof with no change to the algorithm or kernel layer code. This approach therefore offers the potential for portable performance.

Rather than writing the PSy layer manually, the PSyclone code generation system can help a user to optimise the code for a particular architecture (by providing optimisations such as blocking, loop merging, inlining etc), or alternatively, generate the PSy layer automatically.

PSyclone is also being extended to support an API being developed in the GOcean project for two finite difference ocean model benchmarks, one of which is based on the NEMO ocean model.

CHAPTER

TWO

GETTING GOING

2.1 Download

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 Installation in the Developer Guide.

PSyclone is available on the Python Package Index (PyPI) and is hosted on GitHub:

`https://github.com/stfc/PSyclone`

The latest release is 2.1.0 and the latest stable version is on the master branch.

There are two ways to install PSyclone. The first one is directly from PyPI using `pip install`, see *Installation from PyPI* for more detailed information.

Alternatively, PSyclone can be downloaded from GitHub - either see 2.1.0 in the `Releases` tab on the PSyclone page or download and extract the latest release of PSyclone directly, e.g.

```
> wget https://github.com/stfc/PSyclone/archive/2.1.0.tar.gz
> tar zxf 2.1.0.tar.gz
> ls
PSyclone-2.1.0
```

After the source package is downloaded and unpacked, it can be installed using `pip install`, albeit in a slightly different way to the PyPI installation, see *Installation from source* for more detailed information.

Hereon the location where you download or clone PSyclone (including the PSyclone directory itself) will be referred to as `<PSYCLONEHOME>`.

2.2 Environment

In order to use PSyclone (including running the test suite and building documentation) you will need to install it. Before starting the installation process, please refer to the *Dependencies* section below.

2.2.1 Installation from PyPI

The simplest, and recommended, installation process is from PyPI using `pip`:

```
> pip install psyclone
```

for the latest available release, or:

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

where `X.Y.Z` is the specific PSyclone release version (e.g. 2.1.0).

By default, `pip` will attempt a system-wide install. If you wish to do a user-local install instead then supply the `--user` flag:

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

PSyclone can also be installed to a specific location using `--install-option` (see `pip` documentation for more detailed information):

```
> pip install --install-option="--prefix=/my/install/path" psyclone==X.Y.Z
```

Depending on the installation option (e.g. system-wide, user), PSyclone will be installed in different *locations*.

2.2.2 Installation from source

PSyclone can also be installed from a *downloaded* release or repository clone. The simplest way to do this is to use `pip` with the supplied `setup.py`:

```
> cd <PSYCLONEHOME>
> pip install .
```

As above, this attempts a system-wide install. For a user-local install use:

```
> pip install --user .
```

and for a specific location use:

```
> pip install --install-option="--prefix=/my/install/path" .
```

If for some reason you would rather not use `pip` then you can run the setup manually:

```
> python setup.py install
```

or, if you do not have root access:

```
> python setup.py install --user
```

or:

```
> python setup.py install --install-option="--prefix=/my/install/path"
```

As for the *PyPI installation*, different installation options lead to different *locations* of PSyclone installation.

2.2.3 Location and structure of PSyclone installation

Location of installed Pyclone scripts, modules and other accompanying resources is similar to other Python packages:

- The `psyclone` *script* is located in `<python-base-prefix>/bin` directory (depending on your Linux distribution, you may need to add this location to your `$PATH`).
- The PSyclone Python modules are located in `<python-base-prefix>/lib/pythonX.Y/site-packages` directory (where `X.Y` is the version of Python that you are using).
- The *configuration file*, *examples*, *tutorial* and *libraries* are installed in `<python-base-prefix>/share/psyclone` directory.

For a system-wide installation on Linux, `<python-base-prefix>` will likely be `/usr` and if a user-local installation is performed it will likely be `~/.local`.

For an installation to a specific location, `<python-base-prefix>` is simply the path given to the `--install-option="--prefix=/my/install/path"`. Note that if using this method, it will be necessary to take further action to ensure PSyclone can find the *configuration file* installed as a part of this process.

2.2.4 Windows environment

PSyclone can also be installed in Python Windows environment using `pip` as described above. There are some differences in directory structure from Linux, for instance the script directory is usually called `Scripts` instead of `bin` and the modules directory `Lib` instead of `lib`.

Installation in an Anaconda Python environment on Windows also needs to be done using `pip` as `conda install` for PSyclone is currently not supported.

2.3 Dependencies

PSyclone is written in Python so needs Python to be installed on the target machine. PSyclone has been tested under Python 2.7, 3.5, 3.6 and 3.8.

PSyclone immediately relies on four external Python packages; `six`, `configparser`, `fparser` and `pyparsing`. There is also dependency on `enum34` for Python 2 support, however support for Python 2 will be dropped in future. The easiest way to satisfy the Python dependencies is to use the PyPI installation and `pip`.

If everything is working correctly then using `pip` to install PSyclone:

```
> pip install psyclone
```

will automatically install the Python dependencies.

Warning: Starting with the release 1.6.1, PSyclone will install a specific release of `fparser` (version specified in the `setup.py` script).

In addition to the mandatory dependencies just described, PSyclone also has optional dependencies on both `graphviz` and `termcolor`. PSyclone can use `graphviz` to produce a visualisation of a schedule's dependency graph. If this is desired then the Python package `graphviz` (for the Python bindings) as well as the `graphviz` package itself must be installed. If the `graphviz` package is not available then the associated PSyclone routines will return silently and no visualisations will be produced. The Python package `termcolor` is used for pretty-printing a schedule in terminals that support coloured text. If the package is not available then the schedule is simply printed in plain text without colour highlighting.

2.3.1 System-specific set-up

System-specific Set-up for Users instructions are available for Ubuntu 14.04.2 and OpenSUSE 42.2.

2.3.2 fparser

The `fparser` package (https://github.com/stfc/fparser) is a Fortran parser originally developed as a part of the f2py project.

`fparser` is available from the Python Package Index and thus may be installed using `pip` (https://packaging.python.org/installing/#requirements-for-installing-packages):

```
> pip install fparser
```

If you do not have sufficient permissions to perform a system-wide install then you can instruct `pip` to do a user-local install:

```
> pip install --user fparser
```

Should you wish to remove `fparser` then simply do:

```
> pip uninstall fparser
```

If you have already installed `fparser` and want to upgrade to the latest version simply do:

```
> pip install fparser --upgrade
```

Warning: Due to the above-mentioned reliance of PSyclone on a specific `fparser` release, it is not advisable to install `fparser` independently unless it is not to be used with PSyclone. An exception is installation of PSyclone from source for development purposes, see Installation in the Developer Guide.

2.3.3 pyparsing

PSyclone requires `pyparsing`, a library designed to allow parsers to be built in Python. PSyclone uses `pyparsing` to parse Fortran regular expressions as `fparser` does not fully parse these (see here for more information).

PSyclone has been tested with `pyparsing` versions 1.5.2, 2.0.1 and 2.2.0.

You can test whether `pyparsing` is already installed on your machine by typing `import pyparsing` from the Python command line. If `pyparsing` is installed, this command will complete successfully. If `pyparsing` is installed you can check its version by typing `pyparsing.__version__` after successfully importing it.

If `pyparsing` is not installed on your system then it may be installed from the Python Package Index using `pip`:

```
> pip install pyparsing
```

Should you wish to, uninstalling is simply performed by doing:

```
> pip uninstall pyparsing
```

If you do not have sufficient privileges for a system-wide install then you can instruct `pip` to do a user-local install:

```
> pip install --user pyparsing
```

Alternatively, you could follow these instructions.

2.3.4 graphviz

The data dependencies of a PSyIR schedule determine the validity of changes to this schedule. PSyclone supports the visualisation of these dependencies as a graph using `graphviz`. This visualisation is not needed to use PSyclone.

If the Python bindings to `graphviz` are not installed on your system then it may be installed from the Python Package Index using `pip`:

```
> sudo pip install graphviz
```

Should you wish to, uninstalling is simply performed by doing:

```
> sudo pip uninstall graphviz
```

If you do not have sufficient privileges for a system-wide install then you can instruct `pip` to do a user-local install:

```
> pip install --user graphviz
```

If `graphviz` itself is not installed on your system and your system supports the `apt` package manager then see below, otherwise please refer to the download and install instructions which are available here.

If your system supports the `apt` package manager then it can be installed and removed in the following way:

```
> sudo apt install graphviz
> sudo apt remove graphviz
```

2.3.5 termcolor

By default, the `view()` method available on any PSyIR (PSyclone Internal Representation) object prints a plain-text representation to standard-out. However, if the `termcolor` package is available then PSyclone uses this to add colour highlighting to the output text.

Installation (and uninstallation) of this package can be done via `pip` in exactly the same way as for `graphviz`, as described above.

2.4 Configuration

Various aspects of PSyclone are configured through a configuration file, `psyclone.cfg`. The default version of this file is installed to `<python-base-prefix>/shared/psyclone/` during the installation process. Similar to what is described *above*, if a system-wide installation is being performed then this will likely be `/usr/share/psyclone/`. If a user-local installation is performed (`--user` flag to `pip install`) then the location will be something like `~/.local/share/psyclone/`.

Warning: If PSyclone is installed to a non-standard location (e.g. by specifying the `--install-option="--prefix=...` option to `pip install`) then PSyclone will not be able to find the configuration file at execution time. There are two solutions to this: 1. copy the configuration file to a location

where PSyclone will find it (see *Configuration*) or 2. set the `PSYCLONE_CONFIG` environment variable to the full-path to the configuration file, e.g.:

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

Warning: When installing in 'editable' mode (`-e` flag to `pip`), `pip` does *not* install the configuration file. You will have to take one of the two actions described above.

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

2.5 Test

PSyclone contains an extensive test suite, but this test suite is not part of a standard installation. If you want to run the full test suite, you need to install PSyclone from source, see *above* or Installation in the Developer Guide.

2.6 Run

You are now ready to try running PSyclone on the *examples*. One way of doing this is to use the `psyclone` driver script. Assuming it is on your `PATH`:

```
> psyclone
usage: psyclone [-h] [-oalg OALG] [-opsy OPSY] [-okern OKERN] [-api API]
                [-s SCRIPT] [-d DIRECTORY] [-I INCLUDE] [-l {off,all,output}]
                [-dm] [-nodm] [--kernel-renaming {multiple,single}]
                [--profile {invokes,kernels}] [--config CONFIG] [-v]
                filename
psyclone: error: the following arguments are required: filename
```

As indicated above, the `psyclone` script takes the name of 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 *PSy, middle 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 `-oalg` and `-opsy`). If those arguments are not supplied then the script writes the generated/re-written Fortran to the terminal. For details of the other command-line arguments please see the *The psyclone command* Section.

Examples are provided in the `examples` directory of the PSyclone Git repository - if you have cloned the repository then `EGS_HOME` in what follows is the root `PSyclone` directory. Alternatively, if you have installed PSyclone using `pip` then they may be found in the `share/psyclone` directory under your Python installation (see *above* for location of PSyclone installation. In this case you should copy the whole `examples` directory to some convenient location (hereafter called `EGS_HOME`) before attempting to carry out the following instructions. Depending on your precise setup, you may also need to set `PSYCLONE_CONFIG` to the full-path to the PSyclone configuration file (see *Configuration*).

There are seven subdirectories, three of which (`lfric`, `gocean` and `nemo`) correspond to the different APIs/domains that are supported by PSyclone. (Note, that we are currently in the process of renaming the `dynamo0.3` API to `lfric`.) In this case we are going to use one of the LFRic examples:

```
> cd <EGS_HOME>/examples/lfric/eg1
> psyclone -api dynamo0.3 -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 -api dynamo0.3 -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 exection: `import pdb; pdb.set_trace()`. This then enables interactive exploration of the PSyIR if you are interested. Alternatively, you can play with some interactive examples on Binder.

CHAPTER

THREE

THE PSYCLONE COMMAND

The simplest way to run PSyclone is to use the `psyclone` command. If you installed PSyclone using `pip` then this command should be available on your PATH (see *Environment* for more details). Alternatively it can be found in the `<PSYCLONEHOME>/bin` directory. The command takes an algorithm file as input and outputs modified algorithm code and generated PSy code. This section walks through its functionality.

3.1 Running

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] [-oalg OALG] [-opsy OPSY] [-okern OKERN] [-api API]
                [-s SCRIPT] [-d DIRECTORY] [-I INCLUDE] [-l {off,all,output}]
                [-dm] [-nodm] [--kernel-renaming {multiple,single}]
                [--profile {invokes,kernels}] [--config CONFIG] [-v]
                filename

Run the PSyclone code generator on a particular file

positional arguments:
  filename              algorithm-layer source code

optional arguments:
  -h, --help            show this help message and exit
  -oalg OALG            filename of transformed algorithm code
  -opsy OPSY            filename of generated PSy code
  -okern OKERN          directory in which to put transformed kernels
  -api API              choose a particular api from ['dynamo0.1',
                        'dynamo0.3', 'gocean0.1', 'gocean1.0', 'nemo'],
                        default 'dynamo0.3'.
  -s SCRIPT, --script SCRIPT
                        filename of a PSyclone optimisation script
  -d DIRECTORY, --directory DIRECTORY
                        path to a root directory structure containing kernel
                        source code. Multiple roots can be specified by using
                        multiple -d arguments.
```

```
-I INCLUDE, --include INCLUDE
                      path to Fortran INCLUDE files (nemo API only)
-l {off,all,output}, --limit {off,all,output}
                      limit the Fortran line length to 132 characters
                      (default 'off'). Use 'on' to apply limit to both input
                      and output Fortran. Use 'output' to apply line-length
                      limit to output Fortran only.
-dm, --dist_mem       generate distributed memory code
-nodm, --no_dist_mem  do not generate distributed memory code
--kernel-renaming {single,multiple}
                      Naming scheme to use when re-naming transformed
                      kernels.
--profile {invokes,kernels}, -p {invokes,kernels}
                      Add profiling hooks for either 'kernels' or 'invokes'
--config CONFIG       Config file with PSyclone specific options.
-v, --version         Display version information (2.1.0)
```

3.2 Basic Use

The simplest way to use `psyclone` is to provide it with an algorithm file:

```
> psyclone alg.f90
```

If the algorithm file is invalid for some reason, the command should return with an appropriate error. For example, if we use the Python `genkernelstub` script as an algorithm file we get the following:

```
> psyclone <PSYCLONEHOME>/bin/genkernelstub
...
    1:#!/usr/bin/env python <== no parse pattern found for "#" in 'BeginSource' block.
'Parse Error: Fatal error in external fparser tool'
```

If the algorithm file is valid then the modified algorithm code and the generated PSy code will be output to the terminal screen.

3.3 Choosing the API

In the previous section we relied on PSyclone using the default API. The default API, along with the supported APIs can be seen by running the `psyclone` command with the `-h` option.

If you use a particular API frequently and it is not the default then you can change the default by creating a copy of the default `psyclone.cfg` file and editing it. See *Configuration* for more details.

If your code uses an API that is different to the default then you can specify this as an argument to the `psyclone` command.

```
> psyclone -api dynamo0.1 alg.f90
```

3.4 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 alg.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.

3.5 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'
```

3.6 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 gocean1.0 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 dynamo0.3 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.

3.7 Transformation script

By default the `psyclone` command will generate 'vanilla' PSy layer code. The -s option allows a Python script to be specified which can transform the PSy layer. This option is discussed in more detail in the *Script* section.

3.8 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 allow compiler flags to be set which allow lines longer than the Fortran standard. 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 parsed algorithm and kernel 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.

3.9 Distributed memory

By default the `psyclone` command will generate distributed memory (DM) code (i.e. parallelised using MPI). As with the choice of API, this default may be configured by editing `psyclone.cfg` - see *Configuration*. 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 details of PSyclone's support for generating DM code see *Distributed Memory*.

3.10 Automatic Profiling Instrumentation

The `--profile` option allows the user to instruct PSyclone to automatically insert profiling calls within the generated PSy code. Two options are provided, `invokes` and `kernels`. The first of these causes PSyclone to insert profiling-start and -stop calls at the beginning and end of every generated invoke routine. The second puts profiling calls around every kernel call (including the associated loops). 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. For full details on the use of this profiling functionality please see the *Profiling* section.

3.11 Outputting of Transformed 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.

3.12 Fortran INCLUDE Files

For the NEMO API, if the source code to be processed by PSyclone contains INCLUDE statements (other than those for libraries such as MPI) 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 api "nemo" -I /some/path -I /some/other/path alg.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 ideally the INCLUDE line would be left unchanged. However, fparser currently treats any such INCLUDE lines as comments which results in them being lost (fparser issue #138). The workaround for this is to ensure that the location of *all* INCLUDE files is supplied to PSyclone.

Attempting to specify `-I`/`--include` for any API other than NEMO will be rejected by PSyclone.

CHAPTER

FOUR

TUTORIAL

PSyclone provides a tutorial, part of which uses Jupyter notebooks. This can be launched from a browser using binder: https://mybinder.org/v2/gh/stfc/psyclone/master?filepath=tutorial%2Fnotebooks%2Fintroduction.ipynb/.

If PSyclone is installed on your system then you can run the tutorial locally. First find the tutorial. If you have installed PSyclone using `pip` then the examples may be found in `share/psyclone/tutorial/notebooks` under your Python installation (see *here* for possible locations). Next, copy the tutorial to a local writable space (as the notebooks will be modified as you go through the tutorial) and change directory to this tutorial. Lastly, start up the tutorial's introduction `jupyter-notebook introduction.ipynb`.

Another part of the tutorial are practicals that provide hands-on introductions to various functionality contained in the *LFRic* and *NEMO* APIs. The sections in the included directories are worked through using PSyclone in a "normal" Linux environment.

Note: Hands-on practicals are not currently included in a PSyclone installation.

CHAPTER

FIVE

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` under your Python installation (see *here* for possible locations).

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.

Alternatively, some of the examples have associated Jupyter notebooks that may be launched with Binder on MyBinder. This is most easily done by following the links from the top-level README.

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`.

5.1 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 `Makefiles` 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.

5.1.1 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 compiler	9.3
Intel Fortran compiler	17, 21

5.1.2 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 `Makefiles` 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 details in the hands-on practicals documentation.

5.2 GOcean

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.2.4 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).

5.2.5 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. It uses the dl_esm_inf-specific extraction library `netcdf` (`lib/extract/netcdf/dl_esm_inf`), and needs NetCDF to be available (including `nf-config` to detect installation-specific paths). You need to compile the NetCDF extraction library (see *NetCDF Extraction Examples*). The `Makefile` in this example will link with the compiled NetCDF extraction library and NetCDF. You can execute the created binary and it will create two output NetCDF files, one for each of the two invokes.

It will also create two stand-alone driver programs (one for each invoke), that will read the corresponding NetCDF file, and then executes the original code.

Note: At this stage the driver program will not compile (see issue #644).

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, 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: Valid Number Verification (NaN Test)

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 NaN-verification code. It uses the dl_esm_inf-specific nan_test library (`lib/nan_test/dl_esm_inf/`).

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

The `Makefile` in this example will link with the compiled nan_test library. You can execute the created binary and it will print five warnings about invalid numbers at the indices 1 1, . . . , 5 5:

```
PSyData: Variable a_fld has the invalid value
                 Infinity  at index/indices            1           1
mainupdate
...
```

5.2.6 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.

5.3 LFRic

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

5.3.1 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).

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 Example 5: Stencils

Example of kernels which require stencil information.

5.3.6 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.)

5.3.7 Example 7: Column-Matrix Assembly Operators

Example of kernels requiring Column-Matrix Assembly operators.

5.3.8 Example 8: Redundant Computation

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

5.3.9 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.

5.3.10 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.

5.3.11 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.

5.3.12 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
```

5.3.13 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.

5.3.14 Example 14: OpenACC

Example of adding OpenACC directives in the dynamo0.3 API. This is a work in progress so the generated code may not work as expected. However it is never-the-less useful as a starting point. Three scripts are provided.

The first script (`acc_kernels.py`) shows how to add OpenACC Kernels directives to the PSy-layer. This example only works with distributed memory switched off as the OpenACC Kernels transformation does not yet support halo exchanges within an OpenACC Kernels region.

The second script (`acc_parallel.py`)shows how to add OpenACC Loop, Parallel and Enter Data directives to the PSy-layer. Again this example only works with distributed memory switched off as the OpenACC Parallel transformation does not support halo exchanges within an OpenACC Parallel region.

The third script (`acc_parallel_dm.py`) is the same as the second except that it does support distributed memory being switched on by placing an OpenACC Parallel directive around each OpenACC Loop directive, rather than having one for the whole invoke. This approach avoids having halo exchanges within an OpenACC Parallel region.

The generated code has a number of problems including 1) it does not modify the kernels to include the OpenACC Routine directive, 2) a loop's upper bound is computed via a derived type (this should be computed beforehand) 3) set_dirty and set_clean calls are placed within an OpenACC Parallel directive and 4) there are no checks on whether loops are parallel or not, it is just assumed they are - i.e. support for colouring or locking is not yet implemented.

5.3.15 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).

5.3.16 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.

5.3.17 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*. It requires the installation of a NetCDF development environment (see here for installing NetCDF). The code can be compiled with `make compile`, and the binary executed with either `make run` or `./extract` Running the compiled binary will create one NetCDF file `main-update.nc` containing the input and output parameters for the `testkern_w0` kernel call. For example:

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

5.3.18 Example 18: Incrementing a Continuous Field After Reading It

Example of a `GH_READINC` access. 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.

5.4 NEMO

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

5.4.1 Example 1: OpenMP parallelisation of tra_adv

Demonstrates the use of PSyclone to parallelise the loops over vertical levels in a NEMO tracer-advection benchmark using OpenMP.

5.4.2 Example 2: OpenMP parallelisation of traldf_iso

Demonstrates the use of PSyclone to parallelise the loops over vertical levels in some NEMO tracer-diffusion code using OpenMP.

5.4.3 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.

5.4.4 Example 4: Transforming Fortran code to the SIR

Demonstrates that simple Fortran code examples which conform to the NEMO API 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 these simple Fortran examples can be transformed to optimised cuda and/or gridtools code by using PSyclone and then DAWN.

5.4.5 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 the NEMO API in PSyclone. They are *not* intended as 'turn-key' solutions but as a starting point.

5.5 PSyIR

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

5.5.1 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.

5.5.2 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.

CHAPTER

SIX

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` under your Python (or PSyclone, depending on the `pip install` options) installation (see *here* for possible locations).

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.

6.1 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.

6.1.1 Profiling

PSyclone provides wrapper libraries for some common performance profiling tools, such as dl_timer 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.

6.1.2 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 (Dynamo0.3)* and *GOcean1.0* 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.

6.1.3 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 (Dynamo0.3)* and *GOcean1.0* 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.

6.1.4 NAN Test

These libraries test all input and output parameters of a kernel to make sure they are not `NaN` or infinite. More information can be found in the *NAN Test* section.

The libraries for *LFRic (Dynamo0.3)* and *GOcean1.0* APIs are included with PSyclone in the `lib/nan_test` directory. For detailed instructions on how to build and use these libraries please refer to their specific `README.md` documentation.

6.2 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 PSyData API and Jinja Support in the Base Class for full details about the PSyData API).

Compilation of `extract`, `nan_test`, `read_only` and some of the profiling wrapper libraries depends on infrastructure libraries relevant to the API they are used for. *LFRic API* uses the LFRic infrastructure and *GOcean1.0* 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.

6.3 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 GOcean1.0 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

SEVEN

SYSTEM-SPECIFIC SET-UP FOR USERS

This chapter describes the setup for a user of PSyclone. It includes all steps necessary to be able to use PSyclone. And while you could obviously do some development, none of the required tools for testing or documentation creation will be installed.

The System-specific Developer Set-up in the Developers' Guide describes the additional installation of all required tools to run tests and create documentation.

Detailed instructions are provided for Ubuntu 16.04.2 and OpenSUSE 42.2 - if you are working with a different Linux distribution some adjustments will be necessary. You will need a terminal window open in which to enter the commands.

7.1 Installing dependencies

Most required dependencies are installed from the Python Package Index (https://packaging.python.org/installing/) using the program pip ("PIP Installs Packages"). Besides `pip` it is also recommended to install the graphviz package to be able to visualise dependency graphs. This is optional and the associated routine will silently return if the graphviz bindings are not installed.

7.1.1 Installing dependencies on Ubuntu

On Ubuntu `pip` and `graphviz` are installed using `apt-get`. Remember that graphviz is optional and that you'll need to install the graphviz package in addition to the Python bindings.

```
> sudo apt-get install python-pip graphviz
```

7.1.2 Installing dependencies on OpenSUSE

The vanilla OpenSUSE installation includes `pip` for Python 3. Note that the graphviz package is installed by default.

```
> sudo zypper install python-pip
```

7.2 Installing PSyclone

Change your working directory to where you would like to place the code and download the latest stable release of PSyclone.

```
> cd <PSYCLONEHOME>
> wget https://github.com/stfc/PSyclone/archive/2.1.0.tar.gz
> gunzip 2.1.0.tar.gz
> tar xf 2.1.0.tar
> rm 2.1.0.tar
> cd PSyclone-2.1.0
> export PYTHONPATH=`pwd`/src:${PYTHONPATH}
> export PATH=`pwd`/bin:${PATH}
```

This sets up your python path and path appropriately. You may want to set these paths permanently (e.g. by editing your ${HOME}/.bashrc file if you run the BASH shell). You can also use the latest version using git, as described in Installing PSyclone From GitHub.

7.3 Common installation

To avoid warnings during the dependency installation, it is recommended to update `pip` to the latest version:

```
> sudo pip install --upgrade pip
```

Next you need to install the `fparser` and `pyparsing` packages:

```
> sudo pip install fparser pyparsing
```

Tip: With `pip` it is possible to install packages either system-wide (which requires root privileges) as above, or for a single user only (in ~/.local). While the latter is only useful for one particular user, it means that PSyclone can be installed using `pip` without needing root privileges. In order to install a package for a user, add the –user command line option to all pip commands. This flag requests that the packages be installed locally for the current user rather than requiring root access:

```
> pip install --user fparser pyparsing
```

You may remove the use of `sudo` and add the `--user` option to all pip commands described in this document.

Uninstalling is simply a matter of doing:

```
> sudo pip uninstall fparser pyparsing
```

PSyclone supports the ability to output a schedule dependency graph using the graphviz package. This is optional and the associated routine will silently return if the graphviz bindings are not installed. If you have the graphviz package installed (see especially section *Installing dependencies on Ubuntu* if you are on Ubuntu), you also need to install the python bindings to the graphviz package:

```
> sudo pip install graphviz
```

If you just want to use PSyclone then you've installed all you need and you are ready to go to the getting-going *Run* section.

CHAPTER

EIGHT

KERNEL LAYER

In the PSyKAl separation of concerns, Kernel code (code which is created to run within the Kernel layer), operates on a subset of a field (such as a column of cells). The reason for doing this is that it gives the PSy layer the responsibility of calling the Kernel over the spatial domain which is where parallelism is typically exploited in finite element and finite difference codes. The PSy layer is therefore able to call the kernel layer in a flexible way (blocked and/or in parallel for example). Kernel code in the kernel layer is not allowed to include any parallelisation calls or directives 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.

8.1 API

Kernels in the kernel layer are implemented as subroutines within Fortran modules. One or more kernel modules are allowed, each of which can contain one or more kernel subroutines. In the example below there is one module `integrate_one_module` which contains one kernel subroutine `integrate_one_code`. The kernel subroutines contain the code that operates over a subset of the field (such as a column).

Metadata describing the kernel subroutines is required by the PSyclone system to generate appropriate PSy layer code. The metadata is written by the kernel developer and is kept with the kernel code in the same module using a sub-type of the `kernel_type` type. In the example below the `w3_solver_kernel_type` type specifies the appropriate metadata information describing the kernel code for the `dynamo0.3` api:

```
module w3_solver_kernel_mod

  use kernel_mod,              only : kernel_type
  use constants_mod,           only : r_def, i_def
  use fs_continuity_mod,       only : W3, Wchi
  use argument_mod,            only : arg_type, func_type,        &
                                      GH_FIELD, GH_SCALAR,        &
                                      GH_REAL, GH_READ, GH_WRITE, &
                                      GH_BASIS, GH_DIFF_BASIS,    &
                                      GH_QUADRATURE_XYoZ, CELLS

  implicit none

  private

  type, public, extends(kernel_type) :: w3_solver_kernel_type
    private
    type(arg_type) :: meta_args(4) = (/                 &
```

(continues on next page)

(continued from previous page)

```
         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_w0, undf_w0, map_w0, w0_diff_basis,   &
                            nqp_h, nqp_v, wqp_h, wqp_v)
    ...
  end subroutine solver_w3_code

end module w3_solver_kernel_mod
```

8.2 Metadata

Kernel metadata is not required if the PSy layer is going to be written manually - its sole purpose is to let PSyclone know how to generate the PSy layer. The content of Kernel metadata differs depending on the particular API and this information can be found in the API-specific sections of this document.

In all APIs the kernel metadata is implemented as an extension of the *kernel_type* type. The reason for using a type to specify metadata is that it allows the metadata to be kept with the code and for it to be compilable. In addition, currently all APIs will contain information about the arguments in an array called `meta_args`, a specification of what data the kernel code expects in a variable called `operates_on` and a reference to the kernel code itself as a type-bound procedure:

```
type, extends(kernel_type) :: integrate_one_kernel
  ...
  type(...) :: meta_args(...) = (/ ... /)
  ...
  integer :: operates_on = ...
  ...
  contains
  ...
  procedure ...
  ...
end type integrate_one_kernel
```

If no type-bound procedure is declared then a named interface with module procedures must be included in the module:

```
type, extends(kernel_type) :: integrate_one_kernel
  ...
  type(...) :: meta_args(...) = (/ ... /)
  ...
  integer :: operates_on = ...
  ...
end type integrate_one_kernel

interface ...
  module procedure ...
end interface
```

These module procedures provide alternative implementations (using different precisions) of the kernel code. They are selected as appropriate by the Fortran compiler, depending on the precision of the fields being passed to them.

CHAPTER

NINE

ALGORITHM LAYER

In the PSyKAl separation of concerns, the Algorithm layer specifies the algorithm that the scientist would like to run (in terms of calls to kernel routines and Built-in operations) and logically operates on full fields. Algorithm code in the algorithm layer is not allowed to include any parallelisation calls or directives and passes datatypes specified by the particular API.

9.1 API

The Algorithm layer is forbidden from calling the Kernel layer directly. In PSyclone, if the programmer would like to call a Kernel routine or a Built-in operation from the algorithm layer they must use the `invoke` call (which is common to all APIs). The `invoke` call is not necessary (and indeed will not work) if the PSy layer is written manually.

To make an `invoke` call, the algorithm layer developer adds one or more `call invoke()` statements to their code and within the content of the `invoke` call they add a reference to the required Kernel/Built-in and the data to pass to it. For example,

```
...
call invoke(integrate_one_kernel(arg1,arg2))
...
```

For more information on the concept of Built-in operations see the *Built-ins* Section. Details of which operations are supported for a specific API are given in the documentation of that API.

The algorithm layer can consist of an arbitrary number of files containing fortran code, any of which may contain as many `invoke()` calls as is required. PSyclone is applied to an individual algorithm layer file and must therefore be run multiple times if multiple files containing `invoke()` calls exist in the algorithm layer.

The algorithm developer is also able to reference more than one Kernel/Built-in within an invoke. In fact this feature is encouraged for performance reasons. **As a general guideline the developer should aim to use as few invokes as possible with as many Kernel references within them as is possible**. The reason for this is that it allows for greater freedom for optimisation in the PSy layer as PSy layer optimisations are limited to the contents of individual invoke calls - 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 so that the algorithm code is compilable and linkable to the PSy layer and adding in the appropriate `use` statement. For example, the above `integrate_one_kernel` invoke is translated into something like the following:

```
...
use psy, only : invoke_0_integrate_one_kernel
...
```

(continues on next page)

(continued from previous page)

```
call invoke_0_integrate_one_kernel(arg1,arg2)
...
```

You may have noticed from other examples in this guide that an algorithm specification in an invoke call references the metadata `type` in an invoke call, not the `code` directly; this is by design.

For example, in the invoke call below, `integrate_one_kernel` is used.

```
...
call invoke(integrate_one_kernel(arg1,arg2))
...
```

`integrate_one_kernel` is the name of the metadata type in the module, not the name of the subroutine in the Kernel ...

```
module integrate_one_module
  ...
  type, extends(kernel_type) :: integrate_one_kernel
    ...
  end type
  ...
contains
  ...
  subroutine integrate_one_code(...)
  ...
  end subroutine integrate_one_code
  ...
end module integrate_one_module
```

9.1.1 Named Invokes

PSyclone permits the user to optionally specify a label for an invoke call like so:

```
...
call invoke(integrate_one_kernel(arg1,arg2), &
            name="compute something")
...
```

The `name` argument to the invoke call is optional. If supplied it must be a string literal. The content of this string (with any spaces replaced by '_' characters) is used in naming the corresponding PSy-layer routine generated by PSyclone. So, for the above example, the generated PSy-layer subroutine will be named "invoke_compute_something." Each invoke label must currently be unique within an Algorithm source file. Note that, in keeping with the Fortran language, labels are not case sensitive and, after having any spaces replaced by underscores, must be valid Fortran names (e.g. `name="compute(1)"` is invalid). In the future it is intended that the labelling of invokes will help to support invoke-specific optimisations to be applied as well as enabling more readable profiling output. It may also be used to instruct PSyclone to just generate a single subroutine to implement all invokes that share the same label.

9.2 Limitations

In order to re-write the Algorithm layer, as just described, PSyclone must obviously be able to parse the invoke calls. Since the Fortran expression parser used by PSyclone is relatively simple, this means there are limitations on what Fortran may be used when specifying kernel arguments in an invoke call. Since these limitations can have a direct impact on the natural science code, the PSyclone developers endeavour to keep them to a minimum.

The current list of known limitations/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.

CHAPTER

TEN

PSY LAYER

In the PSyKAl separation of concerns, the PSy layer is responsible for linking together the Algorithm and Kernel layers and for providing the implementation of any Built-in operations used. Its functional responsibilities are to

1. map the arguments supplied by an Algorithm `invoke` call to the arguments required by a Built-in or Kernel call (as these will not have a one-to-one correspondance).
2. call any Kernel routines such that they cover the required iteration space and
3. perform any Built-in operations (either by including the necessary code directly in the PSy layer or by e.g. calling a maths library) and
4. include any required distributed memory operations such as halo swaps and reductions.

Its other role is to allow the optimisation expert to optimise any required distributed memory operations, include and optimise any shared memory parallelism and optimise for single node (e.g. cache and vectorisation) performance.

10.1 Code Generation

The PSy layer can be written manually but this is error prone and potentially complex to optimise. The PSyclone code generation system generates the PSy layer so there is no need to write the code manually.

To generate correct PSy layer code, PSyclone needs to understand the arguments and datatypes passed by the algorithm layer and the arguments and datatypes expected by the Kernel layer; it needs to know the name of the Kernel subroutine(s); it needs to know the iteration space that the Kernel(s) is/are written to iterate over; it also needs to know the ordering of Kernels and Built-ins as specified in the algorithm layer. Finally, it needs to know where to place any distributed memory operations.

PSyclone determines the above information by being told the API in question (by the user), by reading the appropriate Kernel and Built-in metadata and by reading the order of Kernels and Built-ins in an invoke call (as specified in the algorithm layer).

PSyclone has an API-specific parsing stage which reads the algorithm layer and all associated Kernel metadata. This information is passed to a PSy-generation stage which creates a high level view of the PSy layer. From this high level view the PSy-generation stage can generate the required PSy code.

For example, the following Python code shows a code being parsed, a PSy-generation object being created using the output from the parser and the PSy layer code being generated by the PSy-generation object.

```
from psyclone.parse.algorithm import parse
from psyclone.psyGen import PSyFactory

# This example uses version 0.1 of the Dynamo API
api = "dynamo0.1"
```

(continues on next page)

(continued from previous page)

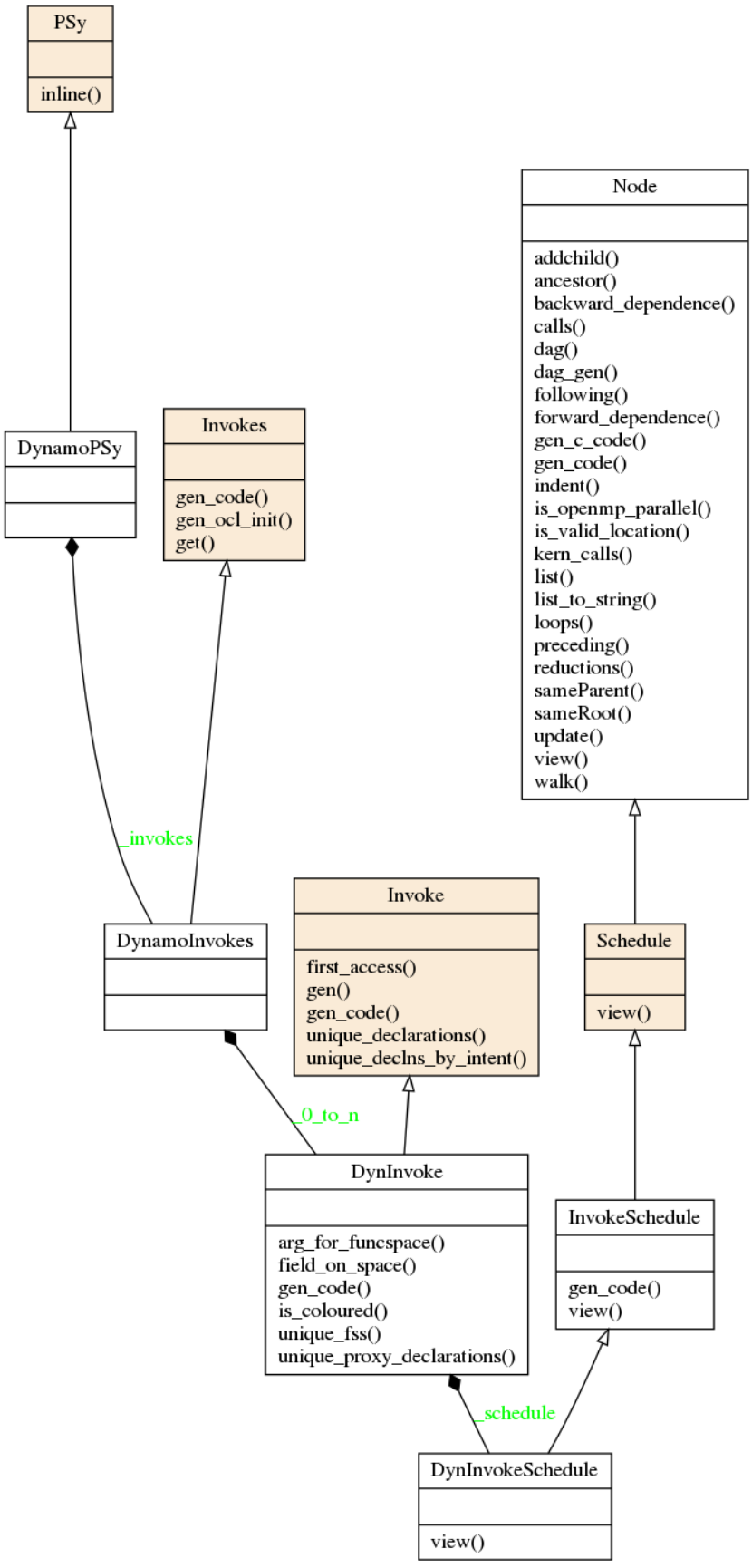
```
# Parse the file containing the algorithm specification and
# return the Abstract Syntax Tree and invokeInfo objects
ast, invokeInfo = parse("dynamo.F90", api=api)

# Create the PSy-layer object using the invokeInfo
psy = PSyFactory(api).create(invokeInfo)
# Generate the Fortran code for the PSy layer
print psy.gen
```

10.2 API

The PSy-layer of a single algorithm file is represented by the **PSy** class. The PSy class has an **Invokes** object which contain one or more **Invoke** instances (one for each invoke in the algorithm layer). Each **Invoke** has an **InvokeSchedule** object with the PSyIR tree that describes the PSy layer invoke subroutine. This subroutine is called by the Algorithm layer and itself calls one or more kernels and/or implements any required Built-in operations.

All this classes can be specialised in each PSyclone API to support the specific features of the APIs. The class diagram for the above base classes is shown below using the dynamo0.3 API as an illustration. This class diagram was generated from the source code with pyreverse and edited with inkscape.


PSy
inline()
DynamoPSy
Invokes
gen_code()
gen_ocl_init()
get()
Node
addchild()
ancestor()
backward_dependence()
calls()
dag()
dag_gen()
following()
forward_dependence()
gen_c_code()
gen_code()
indent()
is_openmp_parallel()
is_valid_location()
kern_calls()
list()
list_to_string()
loops()
preceding()
reductions()
sameParent()
sameRoot()
update()
view()
walk()
_invokes
DynamoInvokes
Invoke
first_access()
gen()
gen_code()
unique_declarations()
unique_declns_by_intent()
Schedule
view()
_0_to_n
DynInvoke
arg_for_funcspace()
field_on_space()
gen_code()
is_coloured()
unique_fss()
unique_proxy_declarations()
InvokeSchedule
gen_code()
view()
_schedule
DynInvokeSchedule
view()

The InvokeSchedule can currently contain nodes of type: **Loop**, **Kernel**, **Built-in** (see the *Built-ins* section), **Directive** (of various types), **HaloExchange**, or **GlobalSum** (the latter two are only used if distributed memory is supported and is switched on; see the *Distributed Memory* section). The order of the tree (depth first) indicates the order of the associated Fortran code.

PSyclone will initially create a "vanilla" (functionally correct but not optimised) InvokeSchedule. This "vanilla" InvokeSchedule can be modified by changing the objects within it. For example, the order that two Kernel calls appear in the generated code can be changed by changing their order in the tree. The ability to modify this high level view of a InvokeSchedule allows the PSy layer to be optimised for a particular architecture (by applying optimisations such as blocking, loop merging, inlining, OpenMP parallelisation etc.). The tree could be manipulated directly, however, to simplify optimisation, a set of transformations are supplied. These transformations are discussed in the next section.

10.2.1 InvokeSchedule visualisation

PSyclone supports visualising an InvokeSchedule (or any other PSyIR node) in two ways. First the *view()* method outputs textual information about the contents of a PSyIR node. If we were to look at the LFRic eg6 example we would see the following output:

```
>>> schedule.view()
InvokeSchedule[invoke='invoke_0', dm=True]
    0: Directive[OMP parallel do]
        Schedule[]
            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_X_code(p,z)
                    1: BuiltIn X_innerproduct_Y_code(rs_old,res,z)
    1: GlobalSum[scalar='rs_old']
```

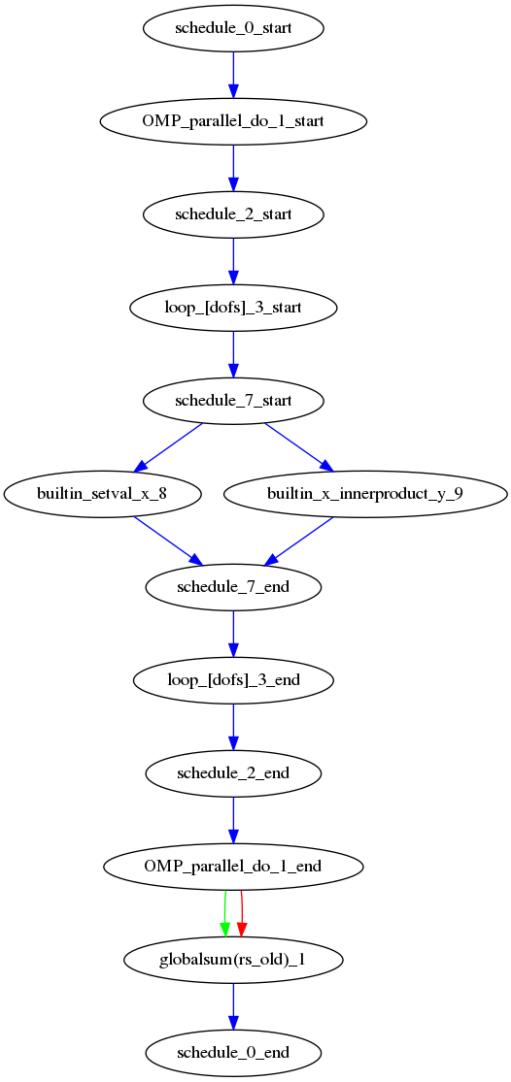
The above output tells us that the invoke name for the InvokeSchedule we are looking at is *invoke_0* and that the distributed_memory option has been switched on. Within the InvokeSchedule is an OpenMP parallel directive containing a loop which itself contains two built-in calls. As the latter of the two built-in calls requires a reduction and distributed memory is switched on, PSyclone has added a GlobalSum call for the appropriate scalar.

Second, the *dag()* method (standing for directed acyclic graph), outputs the PSyIR nodes and its 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

ELEVEN

BUILT-INS

Built-ins (named by analogy with the native functionality provided by Python) are operations which can be specified within an invoke call in the algorithm layer but do not require an associated kernel to be implemented as they are provided directly by the infrastructure.

One use of Built-ins is for commonly used operations. In this case Built-ins simplify the use of the system as users do not need to write kernel routines. Built-ins also offer a potential performance advantage as they provide a specification of what is required without an implementation. Therefore the PSy layer is free to implement these operations in whatever way it chooses.

Note: In general, PSyclone will need to know the types of the arguments being passed to any Built-ins. The parser obtains this information from an API-specific file that contains the metadata for all Built-in operations supported for that API.

11.1 Example

In the following example, the invoke call includes a call to two Built-ins (`setval_c` and `X_divideby_Y`) and a user-supplied kernel (`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 (Dynamo0.3) 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 (*c.f.* Fortran intrinsics such as `sin()`).

```
module solver_mod
  ...
  use matrix_vector_mm_mod, only: matrix_vector_kernel_mm_type
  ...

  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, res

    real(kind=r_def), parameter :: MU = 0.9_r_def
    ...
```

(continues on next page)

(continued from previous page)

```
    ! 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
```

Below is an example of a kernel that is consistent with the `matrix_vector_kernel_mm_type kernel` specified in the example above.

```
module matrix_vector_mm_mod
  type, public, extends(kernel_type) :: matrix_vector_kernel_mm_type
    private
    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_1),             &
         arg_type(GH_OPERATOR, GH_REAL, GH_READ, ANY_SPACE_1, ANY_SPACE_1) &
         /)
    integer :: operates_on = CELL_COLUMN
  contains
    procedure, nopass :: matrix_vector_mm_code
  end type
contains
  subroutine matrix_vector_mm_code(cell,        &
                                   nlayers,     &
                                   lhs, x,      &
                                   ncell_3d,    &
                                   mass_matrix, &
                                   ndf, undf, map)
  end subroutine matrix_vector_mm_code
end module matrix_vector_mm_mod
```

We now translate the algorithm layer code and generate the PSy layer code. The algorithm code is assumed to be in a file called `solver_mod.x90` (see Example 3 in *LFRic examples* section). In this case we use the top level Python interface. See the *API* section for different ways to translate/generate code.

```
> psyclone -nodm -oalg solver_mod.f90 -opsy solver_mod_psy.f90 \
> solver_mod.x90
```

The resultant generated algorithm code is given below.

Ignoring the difference in case (which is due to the output format of the code parser) the differences between the original algorithm code and the translated algorithm code are:

- The generic calls to `invoke` have been replaced by specific `CALL invoke_<xx>`. The kernel calls within the original `invoke` are removed, as are duplicate arguments and any literals, leaving the five fields and one operator as arguments;
- A `use` statement is added for the each of the new `CALL invoke_<xx>` which will call the generated PSy layer code.

The existence of calls to Built-ins has made no difference at this point:

```
SUBROUTINE jacobi_solver_algorithm(lhs, rhs, mm, mesh, n_iter)
  USE solver_mod_psy, ONLY: invoke_jacobi_iterloop
  USE solver_mod_psy, ONLY: invoke_21
  USE solver_mod_psy, ONLY: invoke_jacobi_mass_lump

  IMPLICIT NONE

  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, res

  REAL(KIND = r_def), PARAMETER :: MU = 0.9_r_def

  INTEGER(KIND = i_def) :: iter
  INTEGER(KIND = i_def) :: rhs_fs
  TYPE(function_space_type) :: fs

  ...
  CALL invoke_jacobi_mass_lump(ax, lumped_weight, mm, lhs, rhs)
  ...

END SUBROUTINE jacobi_solver_algorithm
```

A vanilla (with no distributed and shared-memory optimisations) version of the generated PSy layer is given below. As expected, the kernel code is called from the PSy layer. However, in the case of the Built-ins, the code for these has been written directly into the PSy layer:

- `setval_c` translates to the loop setting `ax_proxy%data(df) = 1.0_r_def`;
- `X_divideby_Y` translates to the loop setting `lhs_proxy%data(df) = rhs_proxy%data(df) / lumped_weight_proxy%data(df)`.

This example illustrates that Built-ins may be implemented in whatever way PSyclone sees fit with no change to the algorithm and kernel layers.

```
MODULE solver_mod_psy
  ...

  SUBROUTINE invoke_jacobi_mass_lump(ax, lumped_weight, mm, lhs, rhs)
    USE matrix_vector_mm_mod, ONLY: matrix_vector_mm_code
    TYPE(field_type), intent(in) :: ax, lumped_weight, lhs, rhs
    TYPE(operator_type), intent(in) :: mm
    ...
    !
    ! Initialise field and/or operator proxies
    !
    ax_proxy = ax%get_proxy()
    lumped_weight_proxy = lumped_weight%get_proxy()
    mm_proxy = mm%get_proxy()
    lhs_proxy = lhs%get_proxy()
```

(continues on next page)

(continued from previous page)

```
    rhs_proxy = rhs%get_proxy()
    !
    ! Initialise number of layers
    !
    nlayers = ax_proxy%vspace%get_nlayers()
    !
    ! Look-up dofmaps for each function space
    !
    map_aspc1_lumped_weight => lumped_weight_proxy%vspace%get_whole_dofmap()
    !
    ! Initialise number of DoFs for aspc1_ax
    !
    ndf_aspc1_ax = ax_proxy%vspace%get_ndf()
    undf_aspc1_ax = ax_proxy%vspace%get_undf()
    !
    ! Initialise number of DoFs for aspc1_lumped_weight
    !
    ndf_aspc1_lumped_weight = lumped_weight_proxy%vspace%get_ndf()
    undf_aspc1_lumped_weight = lumped_weight_proxy%vspace%get_undf()
    !
    ! Initialise number of DoFs for aspc1_lhs
    !
    ndf_aspc1_lhs = lhs_proxy%vspace%get_ndf()
    undf_aspc1_lhs = lhs_proxy%vspace%get_undf()
    !
    ! Call our kernels
    !
    DO df=1,undf_aspc1_ax
      ax_proxy%data(df) = 1.0_r_def
    END DO
    DO cell=1,lumped_weight_proxy%vspace%get_ncell()
      !
      CALL matrix_vector_mm_code(cell, nlayers,            &
                                 lumped_weight_proxy%data, &
                                 ax_proxy%data,            &
                                 mm_proxy%ncell_3d,        &
                                 mm_proxy%local_stencil,   &
                                 ndf_aspc1_lumped_weight,  &
                                 undf_aspc1_lumped_weight, &
                                 map_aspc1_lumped_weight(:,cell))
    END DO
    DO df=1,undf_aspc1_lhs
      lhs_proxy%data(df) = rhs_proxy%data(df) / lumped_weight_proxy%data(df)
    END DO
    !
  END SUBROUTINE invoke_jacobi_mass_lump
  ...
END MODULE solver_mod_psy
```

This example is distributed with PSyclone and can be found in <PSYCLONEHOME>/examples/lfric/eg3.

11.2 Supported Built-in operations

The list of supported Built-ins is API-specific and therefore is described under the documentation of each API.

11.3 Adding new Built-in operations

1. Identify the PSyclone source file for the API to be extended. *e.g.* for the LFRic API it is `src/psyclone/domain/lfric/lfric_builtins.py`.
2. Edit this source file to create the class for this new call. It must inherit from the API-specific parent class for Built-in operations (`LFRicBuiltInKern` for the LFRic API).
3. Implement `__str__` and `gen_code()` methods for this new class.
4. Add the name of the new Built-in operation and its corresponding class to the `BUILTIN_MAP` dictionary in that source file.
5. Add metadata describing this call to the appropriate file specified in the `BUILTIN_DEFINITIONS_FILE` in that source file. For the LFRic API this is `src/psyclone/parse/lfric_builtins_mod.f90`.
6. Add relevant tests to the PSyclone test files for the API to be extended. *e.g.* for the LFRic API they are * `src/psyclone/tests/domain/lfric/lfric_builtins_test.py`, * `src/psyclone/tests/domain/lfric/lfric_integer_builtins_test.py`. The tests rely on `single_invoke` Fortran examples in the relevant `src/psyclone/tests/test_files/` subdirectory.
7. Add an appropriate Fortran `single_invoke` example for the new Built-in in the relevant `src/psyclone/tests/test_files/` subdirectory. *e.g.* for the LFRic API it is `src/psyclone/tests/test_files/dynamo0p3/`. Names of examples follow the template `<category.number>.<subcategory.number>_<built-in_name>.f90`. *e.g.* for the LFRic API `<category.number>` is 15 and `<built-in_name>` follows the *LFRic API Built-in naming scheme*.
8. Document the new Built-in in the documentation of the relevant API (*e.g.* `doc/dynamo0p3.rst` for LFRic (Dynamo0.3) API).

If the API being extended does not currently support any Built-ins then the `BUILTIN_MAP` and `BUILTIN_DEFINITIONS_FILE` module variables must be added to the source file for the API. A Fortran module file must be created in the PSyclone `src/parse` directory (with the name specified in `BUILTIN_DEFINITIONS_FILE`) containing metadata describing the Built-in operations. Finally, `parse.get_builtin_defs()` must be extended to import `BUILTIN_MAP` and `BUILTIN_DEFINITIONS_FILE` for this API.

CHAPTER

TWELVE

LFRIC (DYNAMO0.3) API

This section describes the LFRic (Dynamo0.3) 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 (Dynamo0.3) 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*). The Met Office Collaboration Wiki (login required) provides more detailed information about the dynamical core's formulation and data model.

The GungHo 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.

12.1 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.

12.1.1 Example

An example LFRic (Dynamo0.3) API invoke call is given below with various different types of objects supported by the API. These different objects and their use are discussed in the following sections.

```
real(kind=r_def)           :: rscalar
integer(kind=i_def)        :: iscalar
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),               &
```

(continues on next page)

(continued from previous page)

```
            int_builtin2(iscalar, field7),                    &
            kernel2(field1, stencil_extent, field3, lscalar), &
            assembly_kernel(cma_op1, operator1),              &
            name="some_calculation"                           &
          )
call invoke( prolong_kernel_type(field1, field4),             &
            restrict_kernel_type(field5, field6)
          )
```

Please see the *Algorithm layer* section for a description of the `name` argument.

Objects in the LFRic API can be categorised by their functionality as data structures and information that specifies supported operations on a particular data structure. These data structures are represented by the five LFRic (Dynamo 0.3) API argument types: *scalar*, *field*, *field vector*, *operator* and *column-wise operator*. All of them except the field vector are represented in the above example. `qr` represents a quadrature object which provides information required by a kernel to operate on fields (see section *Quadrature* for more details).

12.1.2 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*).

12.1.3 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.

12.1.4 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.

12.1.5 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).

12.1.6 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, 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`.

12.1.7 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 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.

12.1.8 Stencils

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 (Dynamo0.3) 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'"
```

12.1.9 Inter-grid

From the Algorithm layer, an Invoke for inter-grid kernels (those that map fields between grids of different resolution) 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. (Hence the second, separate Invoke in the example Algorithm code given at the beginning of this Section.)

12.2 PSy-layer

The general details of the PSy-layer are explained in the *PSy layer* section. This section describes any dynamo0p3 specific issues.

12.2.1 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.

12.3 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. In the LFRic API there are five different Kernel types; general purpose, CMA, inter-grid, domain and *Built-ins*. In the case of built-ins, PSyclone generates the source of the kernels. This section explains the rules for the other four, user-supplied kernel types and then goes on to describe their metadata and subroutine arguments.

Domain kernels are distinct from the other three, user-supplied kernel types in that they must be passed data for the whole domain rather than a single cell-column. This permits the use of kernels that have not been written to conform to the single-column 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.

12.3.1 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.

12.3.2 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 aguments must therefore be declared in the metadata as `GH_READ` - see *below*.

12.3.3 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.

12.3.4 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.

12.3.5 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.

12.3.6 Metadata

The code below outlines the elements of the LFRic (Dynamo0.3) 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.
- 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` and `setval_x` 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 compu-

tation 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_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 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).

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:dynamo0.3-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 *Stencils*). 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 *Stencils*).

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. For user-supplied kernels this is currently only permitted to be `CELL_COLUMN` or `DOMAIN`. 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
dof	Single DoF (currently *Built-ins* only)
domain	All columns of cells

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
```

12.3.7 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`.
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 of kind `r_def` with extent equal to the number of unique degrees of freedom for the space that the field is on. 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 of kind `r_def`. 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 of kind `r_def`. The first two dimensions are the local degrees of freedom for the `to` and `from` spaces, respectively. The third dimension is `ncell_3d`;
    2) If it is a CMA operator, include a `real`, 3-dimensional array of kind `r_def`. 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 kind `r_def` and is of rank 1. 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`.

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).

12.3.8 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).

12.4 Built-ins

The basic concept of a PSyclone Built-in is described in the *Built-ins* section. In the LFRic (Dynamo0.3) 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`, `int_X`, and from `integer` to `real`, `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. For clarity, the calculation performed by each Built-in is described using Fortran array syntax; this does not necessarily reflect the actual implementation of the Built-in (*e.g.* it could be implemented by PSyclone generating a call to an optimised Maths library).

As described in the PSy-layer *Argument Intents* section, the Fortran intent of LFRic *field* objects is always `in`. The field or scalar that has its data modified by a Built-in is marked in **bold**.

Note: 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(:)`). PSyclone issue #1149 will revisit the representation of declarations and computations in the descriptions of individual Built-ins.

12.4.1 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.

12.4.2 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>_<RHSarg>` for one *RHS* argument or `<RHSargs>_<operationname>_<RHSargs>` for more;
    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>`), except for the Built-in that converts the data type of field arguments from `integer` to `real` (see rule 7 below).
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 (except for the data-type conversion Built-ins, see rule 7 below).

7) As in the case of Built-in field argument rules, the names of the field data-type conversion Built-ins, int_X (converts field data from real to integer) and real_X (converts field data from integer to real), are the only exceptions for the naming of Built-ins in Fortran above.

12.4.3 Built-in operations on real-valued fields

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(:)
```

where:

- type(field_type), intent(in) :: **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(:)
```

where:

- type(field_type), intent(in) :: **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(:)
```

where:

- real(r_def), intent(in) :: *rscalar*
- type(field_type), intent(in) :: **field2**, *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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field**

aX_plus_Y

aX_plus_Y (*field3*, *rscalar*, *field1*, *field2*)

Performs `Z = aX + Y`:

```
field3(:) = rscalar*field1(:) + field2(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field3**, *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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field1**, *field2*

aX_plus_bY

aX_plus_bY (*field3*, *rscalar1*, *field1*, *rscalar2*, *field2*)

Performs `Z = aX + bY`:

```
field3(:) = rscalar1*field1(:) + rscalar2*field2(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar1*, *rscalar2*
- `type(field_type), intent(in) ::` **field3**, *field1*, *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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar1*, *rscalar2*
- `type(field_type), intent(in) ::` **field1**, *field2*

aX_plus_aY

aX_plus_aY (*field3*, *rscalar*, *field1*, *field2*)

Performs `Z = aX + aY = a(X + Y)`:

```
field3(:) = rscalar*(field1(:) + field2(:))
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field3**, *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(:)
```

where:

- `type(field_type), intent(in) ::` **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(:)
```

where:

- `type(field_type), intent(in) ::` **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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field2**, *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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field**

aX_minus_Y

aX_minus_Y (*field3*, *rscalar*, *field1*, *field2*)

Performs `Z = aX - Y`:

```
field3(:) = rscalar*field1(:) - field2(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field3**, *field1*, *field2*

X_minus_bY

X_minus_bY (*field3*, *field1*, *rscalar*, *field2*)

Performs `Z = X - bY`:

```
field3(:) = field1(:) - rscalar*field2(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field3**, *field1*, *field2*

inc_X_minus_bY

inc_X_minus_bY (*field1*, *rscalar*, *field2*)

Performs `X = X - bY` (decrements the first field):

```
field1(:) = field1(:) - rscalar*field2(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field1**, *field2*

aX_minus_bY

aX_minus_bY (*field3*, *rscalar1*, *field1*, *rscalar2*, *field2*)

Performs `Z = aX - bY`:

```
field3(:) = rscalar1*field1(:) - rscalar2*field2(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar1*, *rscalar2*
- `type(field_type), intent(in) ::` **field3**, *field1*, *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(:)
```

where:

- `type(field_type), intent(in)` :: **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(:)
```

where:

- `type(field_type), intent(in) ::` **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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field2**, *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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **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(:)
```

where:

- `type(field_type), intent(in) ::` **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(:)
```

where:

- `type(field_type), intent(in) ::` **field1**, *field2*

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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field2**, *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(:)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **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
```

where:

- `type(field_type), intent(in) ::` **field**
- `real(r_def), intent(in) ::` *constant*

setval_X

setval_X (*field2*, *field1*)

Sets a field *field2* equal (DoF per DoF) to another field *field1* (`Y = X`):

```
field2(:) = field1(:)
```

where:

- `type(field_type), intent(in) ::` **field2**, *field1*

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
```

where:

- `type(field_type), intent(in) ::` **field**
- `real(r_def), intent(in) ::` *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:

- `type(field_type), intent(in) ::` **field**
- `integer(i_def), intent(in) ::` *iscalar*

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.

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:

- `real(r_def), intent(out) ::` **innprod**
- `type(field_type), intent(in) ::` *field1*, *field2*

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:

- `real(r_def), intent(out) ::` **innprod**
- `type(field_type), intent(in) ::` *field*

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.

---

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:

- `real(r_def), intent(out) ::` **sumfld**
- `type(field_type), intent(in) ::` *field*

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 using the Fortran intrinsic `sign` function as `Y = sign(a, X)`, where `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)
```

where:

- `real(r_def), intent(in) ::` *rscalar*
- `type(field_type), intent(in) ::` **field2**, *field1*

Conversion of `real` to `integer` field elements

A Built-in which takes a `real` field and converts it to an `integer` field is denoted with the keyword **int**.

int_X

int_X (*ifield2*, *field1*)

Converts `real`-valued field elements to `integer`-valued field elements using the Fortran intrinsic `int` function as `Y = int(X, i_def)`. Here `Y` is an `integer`-valued field and `X` is the `real`-valued field being converted:

```
ifield2 = INT(field1, i_def)
```

where:

- `type(integer_field_type), intent(in) ::` **ifield2**
- `type(field_type), intent(in) ::` *field1*

Note: The correct `integer` kind, `i_def`, is read from the PSyclone *configuration file*.

12.4.4 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.

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(:)
```

where:

- `type(integer_field_type), intent(in) ::` **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(:)
```

where:

- `type(integer_field_type), intent(in) ::` **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(:)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **ifield2**, *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(:)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **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(:)
```

where:

- `type(integer_field_type), intent(in) ::` **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(:)
```

where:

- `type(integer_field_type), intent(in) ::` **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(:)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **ifield2**, *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(:)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **ifield**

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(:)
```

where:

- `type(integer_field_type), intent(in)` :: **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(:)
```

where:

- `type(integer_field_type), intent(in) ::` **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(:)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **ifield2**, *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(:)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **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
```

where:

- `type(integer_field_type), intent(in) ::` **ifield**
- `integer(i_def), intent(in) ::` *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(:)
```

where:

- `type(integer_field_type), intent(in) ::` **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 using the Fortran intrinsic `sign` function as `Y = sign(a, X)`, where `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)
```

where:

- `integer(i_def), intent(in) ::` *iscalar*
- `type(integer_field_type), intent(in) ::` **ifield2**, *ifield1*

Conversion of `integer` to `real` field elements

A Built-in which takes an `integer` field and converts it to a `real` field is denoted with the keyword **real**.

real_X

real_X (*field2*, *ifield1*)

Converts `integer`-valued field elements to `real`-valued field elements using the Fortran intrinsic `real` function as `Y = real(X, r_def)`. Here `Y` is a `real`-valued field and `X` is the `integer`-valued field being converted:

```
field2 = REAL(ifield1, r_def)
```

where:

- `type(field_type), intent(in) ::` **field2**
- `type(integer_field_type), intent(in) ::` *ifield1*

Note: The correct `real` kind, `r_def`, is read from the PSyclone *configuration file*.

12.5 Boundary Conditions

In the Dynamo0.3 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).

12.6 Conventions

The naming of Dynamo0.3 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 *Stub Generation* 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.

12.7 Configuration

The general and the LFRic-API-specific configuration options are described in the *Configuration* section.

12.7.1 Annexed DoFs

When a kernel operates on DoFs (rather than cell-columns) for a continuous field using distributed memory (see the *Distributed Memory* Section), then 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 Dynamo0.3 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 *dynamo0.3* 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 (Dynamo0.3) developers section.

12.7.2 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.

12.7.3 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.

12.7.4 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.

12.8 Transformations

This section describes the Dynamo0.3-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 Dynamo0.3-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 Dynamo0.3-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 Dynamo0.3 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 Dynamo0.3 API. This is because the properties that it makes constant are API specific.

The Dynamo0.3-API-specific transformations currently available are given below. If the name of a transformation includes "Dynamo0p3" it means that the transformation is only valid for this particular API. If the name of the transformation includes "Dynamo" then it should work with all versions of the Dynamo API.

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.

CHAPTER

THIRTEEN

GOCEAN1.0 API

13.1 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.

13.2 The GOcean Library

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 version 1.0 of the GOcean Library (GOLib v.1.0). Currently this library is distributed separately from PSyclone and is available from http://puma.nerc.ac.uk/trac/GOcean.

13.2.1 Grid

The GOLib 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)

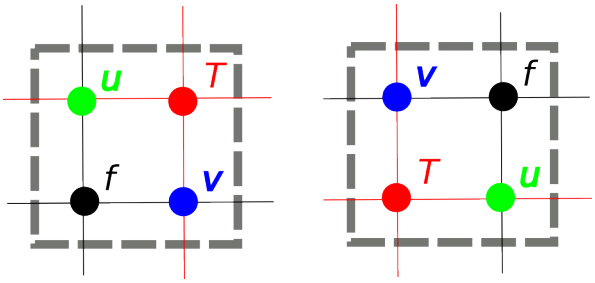
2. The boundary conditions on the domain for the x, y and z dimensions (see below). The value for the z dimension is currently ignored.
3. The 'index offset' - the convention used for indexing into offset fields.

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.

13.2.2 Fields

Once a model has a grid defined it will require one or more fields. The GOLib 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.

13.2.3 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 the GOLib in 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 the GOLib. 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.

13.3 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 Library*).

13.3.1 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.

13.4 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.

13.4.1 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 13.1: 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

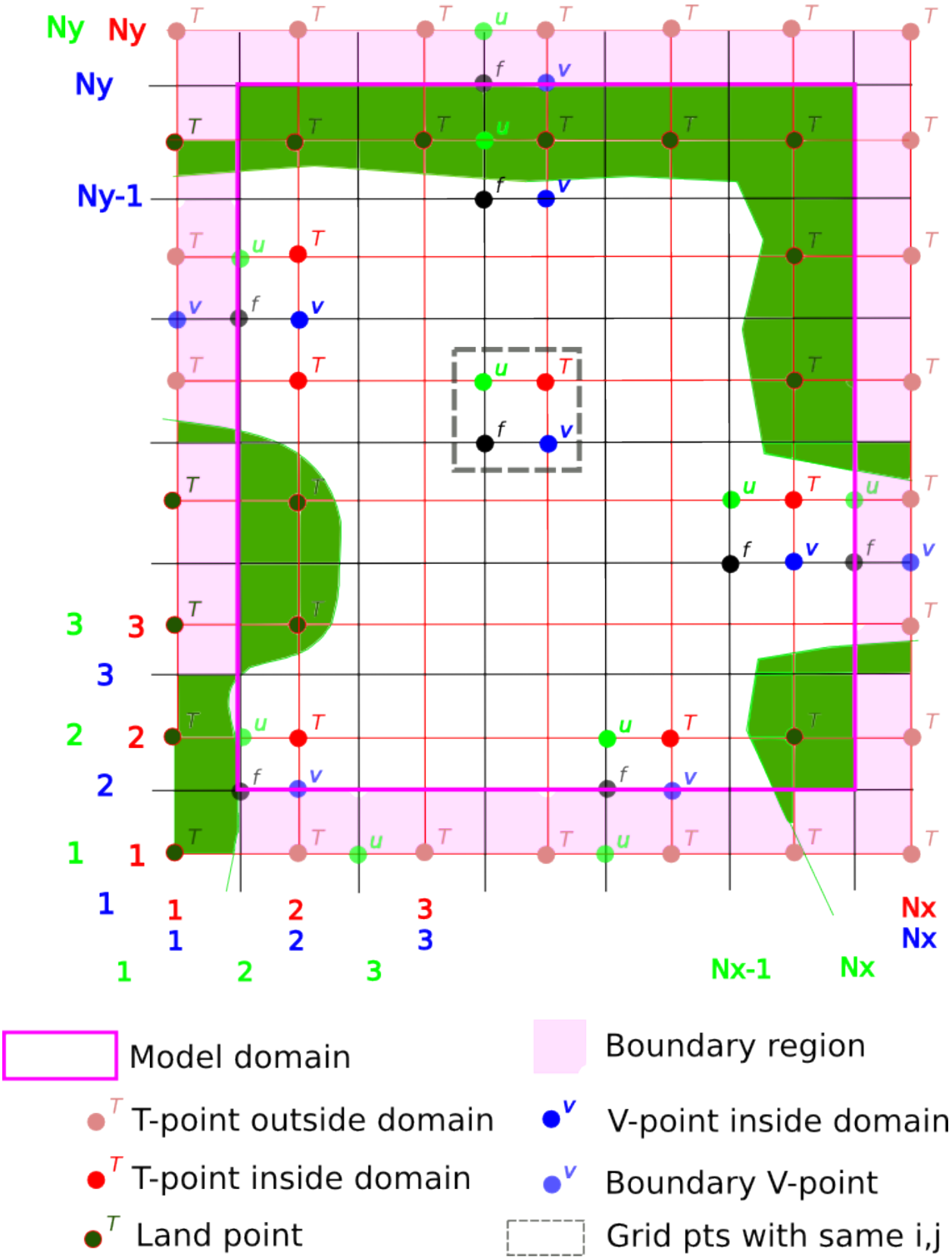
1) it only captures the depth of the stencil in a particular direction, not the actual accesses. Therefore, there is no way to distinguish between the stencil `a(i+2,j)` and the stencil `a(i+1,j) + a(i+2,j)`.

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
```

13.4.2 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.

13.5 Built-ins

The GOcean 1.0 API does not support any built-in operations.

13.6 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.

13.7 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:

```
[gocean1.0]
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.

13.7.1 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.

13.7.2 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).

13.7.3 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:

```
[gocean1.0]
DEBUG_MODE = true
```

Currently, only the OpenCL Invokes generate additional debugging code.

13.8 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*.

CHAPTER

FOURTEEN

NEMO API

In contrast to the other APIs supported by PSyclone, the NEMO API is designed to work with source code that does *not* follow the PSyKAl separation of concerns. Instead, the NEMO source code is treated as if it were a manually written PSy layer with all kernels in-lined. This approach relies upon the NEMO Coding Conventions [nem13] in order to reason about the code being processed. Rather than construct an InvokeSchedule for the PSy layer from scratch (as is done for other APIs), the InvokeSchedule is constructed by parsing the supplied Fortran code and generating a higher-level representation.

Note: the NEMO API is currently only a prototype. The known issues are listed in *Limitations*.

14.1 Algorithm

Since NEMO source is treated as a pre-existing PSy layer, this API does not have the concept of an Algorithm layer.

14.2 Constructing the PSyIR

Transformations in PSyclone are applied to an Internal Representation, the "PSyIR." In contrast to the other APIs where the PSyIR is constructed from scratch, for NEMO PSyclone must parse the existing Fortran and create a higher-level representation of it. This is done using rules based upon the NEMO Coding Conventions [nem13]. These rules are described in the following sections.

14.2.1 Loops

Explicit

PSyclone recognises the following loop types, based on the name of the loop variable:

Loop type	Loop variable
Vertical levels	jk
Latitude	ji
Longitude	jj
Tracer species	jn

PSyclone currently assumes that each of these loop types may be safely parallelised. In practice this will not always be the case (e.g. when performing a tri-diagonal solve) and this implementation will need to be refined.

Implicit

The use of Fortran array notation is encouraged in the NEMO Coding Conventions [nem13] (section 4.2) and is employed throughout the NEMO code base. The Coding Conventions mandate that the shape of every array in such expressions must be specified, e.g.:

```
onedarraya(:) = onedarrayb(:) + onedarrayc(:)
twodarray (:,:) = scalar * anothertwodarray(:,:)
```

PSyclone therefore also recognises the loops implied by this notation.

Note, not all uses of Fortran array notation in NEMO imply a loop. For instance:

```
ascalar = afunc(twodarray(:,:))
```

is actually a function call which is passed a reference to `twodarray`. However, if the quantity being assigned to is actually an array, e.g.:

```
twodarray2(:,:) = afunc(twodarray(:,:))
```

then this does represent a loop. However, currently PSyclone does not recognise any occurrences of array notation that are themselves within an array access or function call. It is therefore not yet possible to transform such implicit loops into explicit loops. It is hoped that this limitation will be removed in future releases of PSyclone by adding the ability to discover the interface to functions such as `afunc` and thus determining whether they return scalar or array quantities.

14.3 Example

A typical fragment of NEMO source code (taken from the traldf_iso routine) is shown below:

```
DO jn = 1, kjpt
 zdit (1,:,:) = 0._wp     ;     zdit (jpi,:,:) = 0._wp
 zdjt (1,:,:) = 0._wp     ;     zdjt (jpi,:,:) = 0._wp

 DO jk = 1, jpkm1
    DO jj = 1, jpjm1
       DO ji = 1, fs_jpim1
          zdit(ji,jj,jk) = ( ptb(ji+1,jj  ,jk,jn) - ptb(ji,jj,jk,jn) ) * umask(ji,jj,jk)
          zdjt(ji,jj,jk) = ( ptb(ji  ,jj+1,jk,jn) - ptb(ji,jj,jk,jn) ) * vmask(ji,jj,jk)
       END DO
    END DO
 END DO
```

PSyclone uses fparser2 to parse such source code and then generates the PSy Internal Representation of it:

```
Loop[type='tracers',field_space='None',it_space='None']
    Loop[type='None',field_space='None',it_space='None']
    Loop[type='None',field_space='None',it_space='None']
    Loop[type='None',field_space='None',it_space='None']
    Loop[type='None',field_space='None',it_space='None']
    Loop[type='levels',field_space='None',it_space='None']
        Loop[type='lat',field_space='None',it_space='None']
            Loop[type='lon',field_space='None',it_space='None']
                CodedKern[]
```

14.4 Transformations

The following transformations are specific to the NEMO API.

14.5 Limitations

The NEMO API is currently under development. Here we list the current, known limitations/issues:

1. Scalar variables inside loops are not made private when parallelising using OpenMP;
2. All recognised loops (levels, latitude etc.) are assumed to be parallelisable. This will not always be the case (e.g. tridiagonal solve has a loop-carried dependence in the vertical);
3. Labelled do-loops are not handled (i.e. they will be put inside a 'CodeBlock' in the PSyIR);
4. Loops are currently only permitted to contain one kernel. This restriction will have to be lifted in order to permit loop fusion;
5. The psyir.nodes.Node base class now has an _ast property to hold a pointer into the associated fparser2 AST. However, the psyGen.Kern class already has an _fp2_ast property that points to the whole fparser2 AST of the kernel code. This will be rationalised in #241;

CHAPTER

FIFTEEN

PSYIR : THE PSYCLONE INTERNAL REPRESENTATION

The PSyIR is at the heart of PSyclone, representing code (at both the PSy- and kernel-layer levels) in a language-agnostic form. A PSyIR 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.

15.1 PSyIR Nodes

The PSyIR consists of classes whose instances can be connected together to form a tree which represent computation in a language-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. PSy-layer classes and Kernel-layer classes. PSy-layer classes make use of a `gen_code()` or an `update()` method to create Fortran code whereas Kernel-layer classes make use of PSyIR backends to create code.

Note: This separation will be removed in the future and eventually all PSyIR classes will make use of backends with the expectation that `gen_code()` and `update()` methods will be removed. Further this separation will be superceded by a separation between `language-level PSyIR` and `domain-specific PSyIR`.

15.1.1 PSy-layer nodes

PSy-layer PSyIR classes are primarily used to create the PSy-layer. These tend to be relatively descriptive and do not specify how a particular PSyclone frontend would implement them. With the exception of `Loop`, these classes are currently not compatible with the PSyIR backends. The generic (non-api-specific) PSy-layer PSyIR nodes are: `InvokeSchedule`, `Directive`, `GlobalSum`, `HaloExchange`, `Loop` and `Kern`. The `Directive` class is subclassed into many directives associated with OpenMP and OpenACC. The `Kern` class is subclassed into `CodedKern`, `InlinedKern` and `BuiltinKern`.

15.1.2 Kernel-layer nodes

Kernel-layer PSyIR classes are currently used to describe existing code in a language independent way. Consequently these nodes are more prescriptive and are independent of a particular PSyclone frontend. These nodes are designed to be used with PSyIR backends. Two PSy-layer classes (`Loop` and `Schedule`) can also be used as Kernel-layer classes. Additionally, the `Schedule` class is further subclassed into a `Routine` and then a kernel-layer `KernelSchedule`. In addition to `KernelSchedule`, Kernel-layer PSyIR nodes are: `Loop`, `IfBlock`, `CodeBlock`, `Assignment`, `Range`, `Reference`, `Operation`, `Literal`, `Call`, `Return` and `Container`. The `Reference` class is further subclassed into `ArrayReference`, `StructureReference` and `ArrayOfStructuresReference`, the `Operation` class is further subclassed into `UnaryOperation`, `BinaryOperation` and `NaryOperation` and the `Container` class is further subclassed into `FileContainer` (representing a file that may contain more than one `Container` and/or `Routine`. Those nodes representing references to structures (derived types in Fortran) have a `Member` child node representing the member of the structure being accessed. The `Member` class is further subclassed into `StructureMember` (representing a member of a structure that is itself a structure), `ArrayMember` (a member of a structure that is an array of primitive types) and `ArrayOfStructuresMember` (a member of a structure this is itself an array of structures).

15.2 Node Descriptions

15.2.1 The Range node

class psyclone.psyir.nodes.**Range**(*ast=None*, *children=None*, *parent=None*, *annotations=None*)

The `Range` node is used to capture a range of integers via `start`, `stop` and `step` expressions. For example, `start=2`, `stop=6` and `step=2` indicates the values 2, 4 and 6.

At the moment the only valid use of `Range` in the PSyIR is to describe a set of accesses to an Array dimension (so-called array notation in Fortran). Therefore, the parent of a `Range` node should only be an `Array` node.

The `Range` node has three children nodes, the first child captures the start of the range, the second child captures the end of the range and the third captures the step within the range.

The nodes for each of the children must return an integer. Potentially valid nodes are therefore `Literal`, `Reference`, `Operation` and `CodeBlock`.

A common use case is to want to specify all the elements of a given array dimension without knowing the extent of that dimension. In the PSyIR this is achieved by using the `LBOUND`, and `UBOUND` binary operators:

```
>>> one = Literal("1", INTEGER_TYPE)
>>> # Declare a 1D real array called 'a' with 10 elements
>>> symbol = DataSymbol("a", ArrayType(REAL_TYPE, [10]))
>>> # Return the lower bound of the first dimension of array 'a'
>>> lbound = BinaryOperation.create(
        BinaryOperation.Operator.LBOUND,
        Reference(symbol), one)
>>> # Return the upper bound of the first dimension of array 'a'
>>> ubound = BinaryOperation.create(
        BinaryOperation.Operator.UBOUND,
        Reference(symbol), one)
>>> # Step defaults to 1 so no need to include it when creating range
>>> my_range = Range.create(lbound, ubound)
>>> # Create an access to all elements in the 1st dimension of array 'a'
>>> array_access = Array.create(symbol, [my_range])
```

In Fortran the above access `array_access` can be represented by `a(:)`. The Fortran front-ends and back-ends are aware of array notation. Therefore the Fortran frontend is able to convert array notation to PSyIR and the

Fortran backend is able to convert PSyIR back to array notation.

static create(*start*, *stop*, *step=None*)
Create an internally-consistent Range object. If no step is provided then it defaults to an integer Literal with value 1.

Parameters

- **start** (psyclone.psyir.nodes.Node) – the PSyIR for the start value.
- **stop** (psyclone.psyir.nodes.Node) – the PSyIR for the stop value.
- **step** (psyclone.psyir.nodes.Node or NoneType) – the PSyIR for the increment/step or None.
- **parent** (psyclone.psyir.nodes.Node or NoneType) – the parent node of this Range in the PSyIR.

Returns a fully-populated Range object.

Return type *psyclone.psyir.nodes.ranges.Range*

property start
Checks that this Range is valid and then returns the PSyIR for the starting value of the range.

Returns the starting value of this range.

Return type psyclone.psyir.nodes.Node

property step
Checks that this Range is valid and then returns the step (increment) value/expression.

Returns the increment used in this range.

Return type psyclone.psyir.nodes.Node

property stop
Checks that this Range is valid and then returns the end value/expression.

Returns the end value of this range.

Return type psyclone.psyir.nodes.Node

15.3 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 writes a text-representation of that node and all of its descendants to stdout. If the `termcolor` package is installed (see *Getting Going*) then colour highlighting is used for this output. For instance, part of the Schedule constructed for the second NEMO example is rendered as:

```
6: If[]
    BinaryOperation[operator:'OR']
        BinaryOperation[operator:'AND']
            BinaryOperation[operator:'EQ']
                Reference[name:'kpass']
                Literal[value:'1']
            Reference[name:'ln_traldf_lap']
        BinaryOperation[operator:'AND']
            BinaryOperation[operator:'EQ']
                Reference[name:'kpass']
                Literal[value:'2']
            Reference[name:'ln_traldf_blp']
    Schedule[]
        0: If[annotations='was_single_stmt']
            Reference[name:'l_ptr']
            Schedule[]
                0: CodeBlock[[<class 'fparser.two.Fortran2003.Call_Stmt'>]]
        1: If[annotations='was_single_stmt']
            Reference[name:'l_hst']
            Schedule[]
                0: CodeBlock[[<class 'fparser.two.Fortran2003.Call_Stmt'>]]
7: Loop[type='levels', field_space='None', it_space='None']
    Literal[value:'1']
    Reference[name:'jpk']
    Literal[value:'1']
    Schedule[]
        0: NemoImplicitLoop[type='None', field_space='None', it_space='None']
8: Loop[type='levels', field_space='None', it_space='None']
    Literal[value:'1']
```

Note that in this view, only those nodes which are children of Schdules 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[6].if_body.children[1]` or, using the shorthand notation (see below), `schedule[6].if_body[1]` where `schedule` is the overall parent Schedule node (omitted from the above image).

15.4 Tree Navigation

Each PSyIR node provides several ways to navigate the AST:

The *children* and *parent* properties (available in all nodes) provide an homogeneous method to go up and down the tree hierarchy. This method is recommended when applying general operations or analysis to the tree, however, if one intends to navigate the tree in a way that depends on the type of node, the *children* and *parent* methods should be avoided. The structure of the tree may change in different versions of PSyclone and the encoded navigation won't be future-proof.

To solve this issue some Nodes also provide methods for semantic navigation:

- **`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

- **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 single Schedule as its child.

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

StructureReference.**member**()

Returns the member of the structure that this reference is to.
Return type psyclone.psyir.nodes.Member
Raises InternalError – if the first child of this node is not an instance of Member.

These are the recommended methods to navigate the tree for analysis or operations that depend on the Node type.

Additionally, the *walk* method (available in all nodes) is able to recurse through the tree and return objects of a given type. This is useful when the objective is to move down the tree to a specific node or list of nodes without information about the exact location.

Node.**walk**(*my_type*, *stop_type=None*)

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.

Parameters

- **my_type** (either a single psyclone.Node class or a tuple of such classes) – the class(es) for which the instances are collected.
- **stop_type** (None or a single psyclone.Node class or a tuple of such classes) – class(es) at which recursion is halted (optional).

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

Return type list of psyclone.Node instances.

15.5 DataTypes

The PSyIR supports the following datatypes: ScalarType, ArrayType, StructureType, DeferredType, UnknownType 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). DeferredType and UnknownType 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.

15.5.1 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, constant_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.

15.5.2 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])
```

15.5.3 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)])
```

15.5.4 Unknown DataType

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

15.5.5 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).

15.6 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 hierachy 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** (psyclone.psyir.nodes.Schedule, psyclone.psyir.nodes.Container or NoneType) – 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** (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.LocalInterface*

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*, *wildcard_import=False*, ***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.
  - **kwargs** (*unwrapped dict.*) – additional keyword arguments provided by *psyclone.psyir.symbols.Symbol*.

- **class** psyclone.psyir.symbols.**DataSymbol**(*name*, *datatype*, *constant_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.
  - **constant_value** (NoneType, item of TYPE_MAP_TO_PYTHON or psyclone.psyir.nodes.Node) – sets a fixed known expression as a permanent value for this DataSymbol. If the value is None then this symbol does not have a fixed 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.**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

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

15.6.1 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.**LocalInterface**
  The symbol just exists in the Local context
- **class** psyclone.psyir.symbols.**ImportInterface**(*container_symbol*)
  Describes the interface to a Symbol that is imported from an external PSyIR container.

  Parameters **container_symbol** (*psyclone.psyir.symbols.ContainerSymbol*) – symbol representing the external container from which the symbol is imported.

  Raises TypeError – if the container_symbol is not a ContainerSymbol.
- **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.**UnresolvedInterface**
  We have a symbol but we don't know where it is declared.

15.7 Creating PSyIR

15.7.1 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,
                        constant_value=3)
```

declares a symbol named "something" of REAL_TYPE datatype where the constant_value argument will be passed to the DataSymbol constructor.

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

15.7.2 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` and for `Loop`).

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.

15.8 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.

15.8.1 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
```

15.8.2 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.

15.8.3 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)
```

15.8.4 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()
```

15.8.5 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()
```

CHAPTER

SIXTEEN

TRANSFORMATIONS

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

16.1 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

16.2 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.

16.2.1 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})
```

16.2.2 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`.

16.3 Available transformations

Most 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 (or specific to a set of APIs e.g. dynamo). 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 NEMO and GOcean 1.0 APIs and the OpenACC Kernels

transformation is limited to the NEMO and Dynamo0.3 APIs.

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** (*dictionary of string:values or None*) – a dictionary with options for transformations.

Warning: This transformation assumes that the ABS Operator 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.**ArrayRange2LoopTrans**(*writer=<psyclone.psyir.backend.fortran.FortranWriter object>*)

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

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "nemo"
>>> filename = "tra_adv_compute.F90"
>>> ast, invoke_info = parse(filename, api=api)
>>> psy = PSyFactory(api).create(invoke_info)
>>> schedule = psy.invokes.invoke_list[0].schedule
>>>
>>> from psyclone.psyir.nodes import Assignment
>>> from psyclone.psyir.transformations import ArrayRange2LoopTrans,     >>>     ␣
↪TransformationError
>>>
>>> schedule.view()
>>> trans = ArrayRange2LoopTrans()
>>> for assignment in schedule.walk(Assignment):
>>>     while True:
>>>         try:
>>>             trans.apply(assignment)
>>>         except TransformationError:
>>>             break
>>> schedule.view()
```

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

Apply the ArrayRange2Loop transformation to the specified node. The node must be an assignment. The rightmost range node in each array within the assignment is replaced with a loop index and the assignment is placed within a loop iterating over that index. 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.

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** (*dictionary of string:values or None*) – 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.**HoistTrans**(*writer=<psyclone.psyir.backend.fortran.FortranWriter object>*)

This transformation takes an assignment and moves it outside of its parent loop if it is valid to do so. 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"
...         "  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
```

(continues on next page)

(continued from previous page)

```
  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** (*dictionary of string:values or None*) – a dictionary with options for transformations.

---

class psyclone.psyir.transformations.**LoopFuseTrans**(*writer=<psyclone.psyir.backend.fortran.FortranWriter object>*)

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.

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

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

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** (*dictionary of string:values or None*) – a dictionary with options for transformations.

---

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. Currently only the matrix vector version of MATMUL is supported.

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)
```

apply(*node*, *options=None*)
Apply the MATMUL intrinsic conversion transformation to the specified node. This node must be a MATMUL BinaryOperation. Currently only the matrix vector version of MATMUL is supported. The arguments are permitted to have additional dimensions (i.e. more than 2 for the matrix and more than 1 for the vector) but the matrix can only have two indices which are ranges and these must be the first two indices and the vector can only have one index that is a range and this must be the first index. Further, the ranges must 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 BinaryOperation node is converted to equivalent inline code.

Parameters

- **node** (psyclone.psyir.nodes.BinaryOperation) – a MATMUL Binary-Operation node.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.

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 Operator 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:
```

(continues on next page)

(continued from previous page)

```
    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** (*dict of str:values or None*) – a dictionary with options for transformations.

Warning: This transformation assumes that the MAX Operator 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.**Min2CodeTrans**

Provides a transformation from a PSyIR MIN Operator 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
```

(continues on next page)

(continued from previous page)

```
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** (*dict of str:values or None*) – a dictionary with options for transformations.

Warning: This transformation assumes that the MIN Operator 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.**ChunkLoopTrans**(*writer=<psyclone.psyir.backend.fortran.FortranWriter object>*)

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
    integer :: ji_out_var
    do ji_out_var = 1, 100, 32
        ji_el_inner = MIN(ji_out_var + 32, 100)
        do ji = ji_out_var, ji_el_inner, 1
            tmp(ji) = 2 * ji
```

(continues on next page)

(continued from previous page)

```
        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** (*dict of str:values or None*) – 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.psyir.transformations.**OMPTaskwaitTrans**(*writer=<psyclone.psyir.backend.fortran.FortranWriter object>*)

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 = "gocean1.0"
>>> 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, OMPTaskwaitTrans
>>> singletrans = OMPSingleTrans()
>>> paralleltrans = OMPParallelTrans()
>>> tasklooptrans = OMPTaskloopTrans()
>>> taskwaittrans = OMPTaskwaitTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> 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:
```

(continues on next page)

(continued from previous page)

```
>>>     tasklooptrans.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)
>>> 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** (*dict of string:values or None*) – 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

---

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.

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.

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.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 = "gocean1.0"
>>> 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
>>> schedule.view()
>>>
>>> # Enclose all children within a single profile region
>>> p_trans.apply(schedule.children)
>>> 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** (*dictionary of string:values or None*) – 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** (*dictionary of string:values or None*) – 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.**Sign2CodeTrans**

Provides a transformation from a PSyIR SIGN Operator 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 BinaryOperation. The SIGN BinaryOperation 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 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) – a SIGN BinaryOperation node.
- **symbol_table** (*psyclone.psyir.symbols.SymbolTable*) – the symbol table.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.

Warning: This transformation assumes that the SIGN Operator 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.

16.4 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.

16.4.1 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.

16.4.2 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.psyir.transformations.**FoldConditionalReturnExpressionsTrans**(*writer=<psyclone.psyir.backend.fort*
object>)

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** (*dict of string:values or None*) – 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** (*dict of string:values or None*) – a dictionary with options for transformations.

Raises **TransformationError** – if the node is not a Routine.

Note: This transformation is only supported by the GOcean 1.0 API.

16.5 Applying

Transformations can be applied either interactively or through a script.

16.5.1 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 psyclone.parse.algorithm import parse
>>> from psyclone.psyGen import PSyFactory

# This example uses the LFRic (Dynamo 0.3) API
>>> api = "dynamo0.3"

# Parse the file containing the algorithm specification and
# return the Abstract Syntax Tree and invokeInfo objects
>>> ast, invokeInfo = parse("dynamo.F90", api=api)

# Create the PSy-layer object using the invokeInfo
>>> psy = PSyFactory(api).create(invokeInfo)

# Optionally generate the vanilla PSy layer fortran
>>> print(psy.gen)
```

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)

# Get the required invoke
>>> invoke = psy.invokes.get('invoke_0_v3_kernel_type')

# Get the schedule associated with the required invoke
>>> schedule = invoke.schedule
>>> schedule.view()
```

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
```

(continues on next page)

(continued from previous page)

```
>>> ol = OMPParallelLoopTrans()

# Apply it to the loop schedule of the selected invoke
>>> ol.apply(schedule.children[0])
>>> schedule.view()

# Generate the Fortran code for the new PSy layer
>> print(psy.gen)
```

16.5.2 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 code. For example:

```
> psyclone algspec.f90
> psyclone -oalg alg.f90 -opsy psy.f90 -api dynamo0.3 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 Python search path **PYTHONPATH** will be used to try to find the script file.

Alternatively, script files may be specified with a path. In this case the file is expected to be found in the specified location. 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).

psyclone.generator.**generate**(*filename*, *api=''*, *kernel_paths=None*, *script_name=None*, *line_length=False*, *distributed_memory=None*, *kern_out_path=''*, *kern_naming='multiple'*)

Takes a PSyclone algorithm specification as input and outputs the associated generated algorithm and psy codes suitable for compiling with the specified kernel(s) and support infrastructure. Uses the `parse.algorithm.parse()` function to parse the algorithm specification, the `psyGen.PSy` class to generate the PSy code and the `alg_gen.Alg` class to generate the modified algorithm code.

Parameters

- **filename** (*str*) – the file containing the algorithm specification.
- **api** (*str*) – the name of the API to use. Defaults to empty string.
- **kernel_paths** (*list of str or NoneType*) – the directories from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification). Defaults to None.
- **script_name** (*str*) – a script file that can apply optimisations to the PSy layer (can be a path to a file or a filename that relies on the PYTHONPATH to find the module). Defaults to None.

- **line_length** (*bool*) – a logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms. The default is False.
- **distributed_memory** (*bool*) – a logical flag specifying whether to generate distributed memory code. The default is set in the 'config.py' file.
- **kern_out_path** (*str*) – directory to which to write transformed kernel code. Defaults to empty string.
- **kern_naming** (*bool*) – the scheme to use when re-naming transformed kernels. Defaults to "multiple".

Returns 2-tuple containing the fparser1 AST for the algorithm code and the fparser1 AST or a string (for NEMO) of the psy code.

Return type (fparser.one.block_statements.BeginSource, fparser.one.block_statements.Module) or (fparser.one.block_statements.BeginSource, str)

Raises

- **GenerationError** – if an invalid API is specified.
- **GenerationError** – if an invalid kernel-renaming scheme is specified.
- **IOError** – if the filename or search path do not exist.

For example:

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("algspec.f90")
>>> alg, psy = generate("algspec.f90", kernel_paths=["src/kernels"])
>>> alg, psy = generate("algspec.f90", script_name="optimise.py")
>>> alg, psy = generate("algspec.f90", line_length=True)
>>> alg, psy = generate("algspec.f90", distributed_memory=False)
```

A valid script file must contain a **trans** function which accepts a **PSy** object as an argument and returns a **PSy** object, i.e.:

```
def trans(psy)
    ...
    return psy
```

It is up to the script what it does with the PSy object. The example below does the same thing as the example in the *Interactive* section.

```
def trans(psy):
    from psyclone.transformations import OMPParallelLoopTrans
    invoke = psy.invokes.get('invoke_0_v3_kernel_type')
    schedule = invoke.schedule
    ol = OMPParallelLoopTrans()
    ol.apply(schedule.children[0])
    return psy
```

Of course the script may apply as many transformations as is required for a particular 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).

16.6 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 Do** 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.

16.6.1 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.

16.6.2 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.

16.6.3 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*.

16.7 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 files follow the *–kernel-renaming* argument conventions, but in addition to the *<modulename>* they also include the *<kernelname>* as part of the filename in the format: *<modulename>_<kernelname>_index.cl*

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.

```
from psyclone.psyir.transformations import \
    FoldConditionalReturnExpressionsTrans
from psyclone.domain.gocean.transformations import GOOpenCLTrans, \
    GOMoveIterationBoundariesInsideKernelTrans


def trans(psy):
    '''
    Transformation routine for use with PSyclone. Applies the OpenCL
    transform to the first Invoke in the psy object.

    :param psy: the PSy object which this script will transform.
    :type psy: :py:class:`psyclone.psyGen.PSy`
    :returns: the transformed PSy object.
    :rtype: :py:class:`psyclone.psyGen.PSy`

    '''
    ocl_trans = GOOpenCLTrans()
    fold_trans = FoldConditionalReturnExpressionsTrans()
    move_boundaries_trans = GOMoveIterationBoundariesInsideKernelTrans()

    # Get the Schedule associated with the first Invoke
    invoke = psy.invokes.invoke_list[0]
    sched = invoke.schedule

    # Provide kernel-specific OpenCL optimization options
    for idx, kern in enumerate(sched.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
    ocl_trans.apply(sched, 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
```

16.8 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 NEMO and Dynamo0.3 APIs) 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. Again, please see PSyclone/examples/gocean/eg2 for an example. This transformation is currently not supported for kernels in the Dynamo0.3 API.

16.9 SIR

It is currently not possible for PSyclone to output SIR code without using a script. Two examples of such scripts are given in example 4 for the NEMO API, one of which includes transformations to remove PSyIR intrinsics, 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).

CHAPTER

SEVENTEEN

DISTRIBUTED MEMORY

PSyclone supports the generation of code for distributed memory machines. When this option is switched on, PSyclone takes on responsibility for both performance and correctness, as described below.

17.1 Correctness

PSyclone is responsible for adding appropriate distributed memory communication calls to the PSy layer to ensure that the distributed memory code runs correctly. For example, a stencil operation will require halo exchanges between the different processes.

The burden of correctly placing distributed memory communication calls has traditionally been born by the user. However, PSyclone is able to determine the placing of these within the PSy-layer, thereby freeing the user from this responsibility. Thus, the Algorithm and Kernel code remain the same, irrespective of whether the target architecture does or does not require a distributed memory solution.

17.2 Performance

PSyclone adds **HaloExchange** and **GlobalSum** objects to the generated PSyIR **InvokeSchedule** at the required locations. The halo-exchange and global-sum objects are exposed here for the purposes of optimisation. For example the halo-exchange and/or global-sum objects may be moved in the schedule (via appropriate transformations) to enable overlap of computation with communication.

Note: When these optimisations are implemented, add a reference to the *Transformations* Section.

A halo exchange is required with distributed memory when a processor requires data from its halo and the halo information is out of date. One example is where a field is written to and then read using a stencil access. Halo exchanges have performance implications so should only be used where necessary.

A global sum is required with distributed memory when a scalar is written to. Global sums can have performance implications so should only be used where necessary. Global sums currently only occur in certain Built-in kernels. The description of Built-ins indicates when this is the case.

17.3 Implementation

Within the contents of an `invoke()` call, PSyclone is able to statically determine which communication calls are required and where they should be placed. However, PSyclone has no information on what happens outside `invoke()` calls and thus is unable to statically determine whether communication is required between these calls. The solution we use is to add run-time flags in the PSy layer to keep track of whether data has been written to and read from. These flags are then used to determine whether communication calls are required upon entry to an `invoke()`.

17.4 Control

Support for distributed memory can be switched on or off with the default being on. The default can be changed permanently by modifying the `DISTRIBUTED_MEMORY` variable in the `psyclone.cfg` configuration file to `false` (see *Configuration*).

Distributed memory can be switched on or off from the `psyclone` script using the `-dm`/`--dist_mem` or `-nodm`/`--no_dist_mem` flags, respectively.

For interactive access, the distributed memory option can be changed interactively from the `PSyFactory` class by setting the optional `distributed_memory` flag; for example:

```
psy = PSyFactory(api=api, distributed_memory=False)
```

Similarly the distributed memory option can be changed interactively from the `generate` function by setting the optional `distributed_memory` flag, e.g.:

```
psy, alg = generate("file.f90", distributed_memory=False).
```

17.5 Status

Distributed memory support is currently supported by the `dynamo0.3` and the `gocean1p0` APIs. The remaining APIs ignore the distributed memory flag and continue to produce code without any distributed memory functionality, irrespective of its value.

CHAPTER

EIGHTEEN

STUB GENERATION

18.1 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

```
> genkernelstub <PATH>/my_file.f90
```

18.2 Introduction

PSyclone provides a kernel stub generator for the LFRic (Dynamo 0.3) 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 require 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.

18.3 Use

Before using the stub generator, 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 stub generator script `genkernelstub` comes with the PSyclone installation. A quick check `> which genkernelstub` should return the location of the <PSYCLONEINSTALL>`/bin` directory.

The easiest way to use the stub generator is to run the supplied `genkernelstub` script.

```
> genkernelstub
usage: genkernelstub [-h] [-o OUTFILE] [-api API] [-l] filename
genkernelstub: error: the following arguments are required: filename
```

You can get information about the `genkernelstub` arguments using `-h` or `--help`:

```
>  genkernelstub -h
usage: genkernelstub [-h] [-o OUTFILE] [-api API] [-l] filename

Create Kernel stub code from Kernel metadata

positional arguments:
  filename              Kernel metadata

optional arguments:
  -h, --help            show this help message and exit
  -o OUTFILE, --outfile OUTFILE
                        filename of output
  -api API              choose a particular api from ['dynamo0.3'], default
                        dynamo0.3
  -l, --limit           limit the fortran line length to 132 characters
```

As is indicated when using the `-h` option, the `-api` option only accepts `dynamo0.3` (LFRic) at the moment and is redundant as this option is also the default. However the number of supported APIs is expected to expand in the future.

The `-o`, or `--outfile` option allows the user to specify that the output should be written to a particular file. If `-o` is not specified then the python `print` statement is used. Typically the print statement 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 (please see the *Line Length* chapter for more details).

18.4 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: `simple.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/simple.f90
```

18.5 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/simple.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) /)
    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 `simple.f90` example:

```
> genkernelstub tests/test_files/dynamo0p3/simple.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 (Dynamo 0.3) 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
```

(continues on next page)

(continued from previous page)

```
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

contains

  subroutine ru_code()
  end subroutine ru_code

end module ru_kernel_mod
```

If we run the kernel stub generator on this example:

```
> genkernelstub 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
```

(continues on next page)

(continued from previous page)

```
    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
    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`.

```
> genkernelstub ../../examples/lfric/eg5/conservative_flux_kernel_mod.F90
```

18.6 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
```

(continues on next page)

(continued from previous page)

```
tests/test_files/dynamo0p3/testkern_dofs_mod.f90
tests/test_files/dynamo0p3/testkern_invalid_fortran.F90
tests/test_files/dynamo0p3/testkern_short_name.F90
tests/test_files/dynamo0p3/testkern_no_datatype.F90
tests/test_files/dynamo0p3/testkern_qr.F90
```

`testkern_invalid_fortran.F90`, `testkern_no_datatype.F90`, `testkern_short_name.F90` and `testkern_qr.F90` are designed to be invalid for PSyclone stub generation testing purposes and should produce appropriate errors. Two examples are below:

```
> genkernelstub tests/test_files/dynamo0p3/testkern_invalid_fortran.F90
Error: 'Parse Error: Code appears to be invalid Fortran'

> genkernelstub tests/test_files/dynamo0p3/testkern_no_datatype.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 (Dynamo 0.3) fields* in the stub generator. 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:

```
> genkernelstub 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:

```
> genkernelstub tests/test_files/dynamo0p3/testkern_qr.F90
Error: "Parse Error: Error, module name 'testkern_qr' does not have
'_mod' as an extension. This convention is assumed."
```

CHAPTER

NINETEEN

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.

19.1 Script

The *psyclone* script provides the -l option to wrap lines. Please see the *Fortran line length* section for more details.

19.2 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
```

19.3 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

TWENTY

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 *Stub Generation* Section) where the name of the metadata to be parsed is determined from the module name.

CHAPTER

TWENTYONE

API

21.1 The generator module

This module provides the PSyclone 'main' routine which is intended to be driven from the bin/psyclone executable script. 'main' takes an algorithm file as input and produces modified algorithm code and generated PSy code. A function, 'generate', is also provided which has the same functionality as 'main' but can be called from within another Python program.

psyclone.generator.**generate**(*filename*, *api=''*, *kernel_paths=None*, *script_name=None*, *line_length=False*, *distributed_memory=None*, *kern_out_path=''*, *kern_naming='multiple'*)

Takes a PSyclone algorithm specification as input and outputs the associated generated algorithm and psy codes suitable for compiling with the specified kernel(s) and support infrastructure. Uses the `parse.algorithm.parse()` function to parse the algorithm specification, the `psyGen.PSy` class to generate the PSy code and the `alg_gen.Alg` class to generate the modified algorithm code.

Parameters

- **filename** (*str*) – the file containing the algorithm specification.
- **api** (*str*) – the name of the API to use. Defaults to empty string.
- **kernel_paths** (*list of str or NoneType*) – the directories from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification). Defaults to None.
- **script_name** (*str*) – a script file that can apply optimisations to the PSy layer (can be a path to a file or a filename that relies on the PYTHONPATH to find the module). Defaults to None.
- **line_length** (*bool*) – a logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms. The default is False.
- **distributed_memory** (*bool*) – a logical flag specifying whether to generate distributed memory code. The default is set in the 'config.py' file.
- **kern_out_path** (*str*) – directory to which to write transformed kernel code. Defaults to empty string.
- **kern_naming** (*bool*) – the scheme to use when re-naming transformed kernels. Defaults to "multiple".

Returns 2-tuple containing the fparser1 AST for the algorithm code and the fparser1 AST or a string (for NEMO) of the psy code.

Return type (fparser.one.block_statements.BeginSource, fparser.one.block_statements.Module) or (fparser.one.block_statements.BeginSource, str)

Raises

- **GenerationError** – if an invalid API is specified.
- **GenerationError** – if an invalid kernel-renaming scheme is specified.
- **IOError** – if the filename or search path do not exist.

For example:

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("algspec.f90")
>>> alg, psy = generate("algspec.f90", kernel_paths=["src/kernels"])
>>> alg, psy = generate("algspec.f90", script_name="optimise.py")
>>> alg, psy = generate("algspec.f90", line_length=True)
>>> alg, psy = generate("algspec.f90", distributed_memory=False)
```

21.2 The parse module

Module that uses the Fortran parser fparser2 to parse PSyclone-conformant Algorithm code.

psyclone.parse.algorithm.**parse**(*alg_filename*, *api=''*, *invoke_name='invoke'*, *kernel_paths=None*, *line_length=False*)

Takes a PSyclone conformant algorithm file as input and outputs a parse tree of the code contained therein and an object containing information about the 'invoke' calls in the algorithm file and any associated kernels within the invoke calls.

Parameters

- **alg_filename** (*str*) – the file containing the algorithm specification.
- **api** (*str*) – the PSyclone API to use when parsing the code. Defaults to empty string.
- **invoke_name** (*str*) – the expected name of the invocation calls in the algorithm code. Defaults to "invoke".
- **kernel_paths** (*list of str or NoneType*) – the paths to search for kernel source files (if different from the location of the algorithm source). Defaults to None.
- **line_length** (*bool*) – a logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms and an error raised if not. The default is False.

Returns 2-tuple consisting of the fparser2 parse tree of the Algorithm file and an object holding details of the invokes found.

Return type (fparser.two.Fortran2003.Program, psyclone.parse.FileInfo)

For example:

```
>>> from psyclone.parse.algorithm import parse
>>> ast, info = parse("alg.f90")
```

21.3 The transformations module

21.4 The psyGen module

This module provides generic support for PSyclone's PSy code optimisation and generation. The classes in this method need to be specialised for a particular API and implementation.

class psyclone.psyGen.**PSy**(*invoke_info*)

Base class to help manage and generate PSy code for a single algorithm file. Takes the invocation information output from the function `parse.algorithm.parse()` as its input and stores this in a way suitable for optimisation and code generation.

Parameters **invoke_info** (`psyclone.parse.algorithm.FileInfo`) – An object containing the required invocation information for code optimisation and generation. Produced by the function `parse.algorithm.parse()`.

For example:

```
>>> from psyclone.parse.algorithm import parse
>>> ast, info = parse("argspec.F90")
>>> from psyclone.psyGen import PSyFactory
>>> api = "..."
>>> psy = PSyFactory(api).create(info)
>>> print(psy.gen)
```

property container

Returns the container associated with this PSy object

Return type `psyclone.psyir.nodes.Container`

abstract property gen

Abstract base class for code generation function.

Returns root node of generated Fortran AST.

Return type `psyclone.psyir.nodes.Node`

property invokes

Returns the list of invokes.

Return type `psyclone.psyGen.Invokes` or derived class

property name

Returns the name of the PSy object.

Return type str

21.5 The alg_gen module

This module provides the Alg class and supporting exception-handling to translate the original algorithm file into one that can be compiled and linked with the generated PSy code.

class psyclone.alg_gen.**Alg**(*parse_tree*, *psy*, *invoke_name='invoke'*)

Generate a modified algorithm code for a single algorithm specification. Takes the parse tree of the algorithm specification output from the function *psyclone.parse.algorithm.parse()* and an instance of the psyGen.PSy class as input. The latter allows consistent names to be generated between the algorithm (callng) and psy (callee) layers.

For example:

```
>>> from psyclone.algorithm.parse import parse
>>> parse_tree, info = parse("argspec.F90")
>>> from psyclone.psyGen import PSy
>>> psy = PSy(info)
>>> from psyclone.alg_gen import Alg
>>> alg = Alg(parse_tree, psy)
>>> print(alg.gen)
```

Parameters

- **parse_tree** (fparser.two.utils.Base) – an object containing a parse tree of the algorithm specification which was produced by the function *psyclone.parse.algorithm.parse()*. Assumes the algorithm will be parsed by fparser2 and expects a valid program unit, program, module, subroutine or function.
- **psy** (*psyclone.psyGen.PSy*) – an object containing information about the PSy layer.
- **invoke_name** (*str*) – the name that the algorithm layer uses to indicate an invoke call. This is an optional argument that defaults to the name "invoke".

property gen

Return modified algorithm code.

Returns the modified algorithm specification as an fparser2 parse tree.

Return type fparser.two.utils.Base

21.6 The line_length module

Provides support for breaking long fortran lines into smaller ones to allow the code to conform to the maximum line length limits (132 for f90 free format is the default)

class psyclone.line_length.**FortLineLength**(*line_length=132*)

This class take a free format fortran code as a string and line wraps any lines that are larger than the specified line length

property length

returns the maximum allowed line length

long_lines(*fortran_in*)

returns true if at least one of the lines in the input code is longer than the allowed length. Otherwise returns false

process(*fortran_in*)
takes fortran code as a string as input and output fortran code as a string with any long lines wrapped appropriately

CHAPTER

TWENTYTWO

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 with the *PSyclone installation*. 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. An example library that extracts input and output data into a NetCDF file is included with PSyclone (see *NetCDF Extraction Examples*).

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.

NAN Test: The callbacks can be used to make sure that all floating point input and output parameters of a kernel are not a `NaN` (not-a-number) or infinite. See *NAN Test* 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 PSyData API for full details about the PSyData API.

22.1 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
```

(continues on next page)

(continued from previous page)

```
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 an invoke (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 miniapp (the executable is named `gravity_wave`):

```
def trans(psy):
    from psyclone.psyir.transformations import ReadOnlyVerifyTrans
    from psyclone.psyir.nodes import Loop
    read_only_verify = ReadOnlyVerifyTrans()

    for invoke in psy.invokes.invoke_list:
        schedule = invoke.schedule
        for node in schedule:
            if isinstance(node, Loop):
                read_only_verify.apply(node)

    return psy
```

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.

22.1.1 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.

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`.

22.1.2 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*.

22.2 NAN Test

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 not `NaN` or infinite. If they are, an error message like the following is printed, but the program is not aborted:

```
PSyData: Variable a_fld has the invalid value Inf at index/indices 1 1 in module 'main'␣
↪region 'update'.
```

Is uses the function `IEEE_IS_FINITE` from the ieee_arithmetic module for this test. Note that only floating point numbers will be tested. Integer numbers do not have a bit pattern for 'infinity' or `NaN`.

The runtime libraries for GOcean and LFRic are based on a jinja-template contained in the directory `<PSYCLONEHOME>/lib/nan_test`. 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/nan_test/lfric` and``lib/nan_test/dl_esm_inf`` subdirectories, respectively. For more information on how to build and link these libraries, please refer to the relevant `README.md` files.

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.

CHAPTER

TWENTYTHREE

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, 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.

23.1 Interface to Third Party Profiling Tools

PSyclone comes with *wrapper libraries* to support usage of 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/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 (PSyData API).

Most libraries in `lib/profiling` need to be linked in with the corresponding 3rd party profiling tool. 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.

23.2 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:

23.2.1 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:

```
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()`.

23.2.2 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:

```
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()`).

23.3 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 (including all kernel calls in this invoke), and/or around each individual kernel.

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 kernels` will surround each outer loop created by PSyclone with start and end profiling calls.

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):

```
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']
                ...
```

(continues on next page)

(continued from previous page)

```
                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:

```
GOInvokeSchedule[invoke='invoke_1']
    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,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:

```
GOInvokeSchedule[invoke='invoke_1']
    0: [Profile]
        Schedule[]
            0: [Profile]
                Schedule[]
```

(continues on next page)

(continued from previous page)

```
                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']
                    ...
                    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]
```

23.4 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:

```
from psyclone.psyir.transformations import ProfileTrans

p_trans = ProfileTrans()
schedule = psy.invokes.get('invoke_0').schedule
schedule.view()

# Enclose some children within a single profile region
p_trans.apply(schedule.children[1:3])
```

(continues on next page)

(continued from previous page)

```
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:

```
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)}
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!

23.5 Naming Profiling Regions

A profile region derives its name from two components:

module_name A string identifying the psy-layer containing this profile node.

region_name A string identifying the invoke containing this profile node and its location within the invoke (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 the API according to the following rules:

For the *NEMO API*,

- the *module_name* string is set to the name of the parent function/subroutine/program. This name is unique as Fortran requires these names to be unique within a program.
- the *region_name* is set to 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).

For the *LFRic (Dynamo0.3)* and *GOcean1.0* 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:

```
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]
                  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]
          2: 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_qr_code(f1,f2,m1,a,m2,istp)
                     [module_inline=False]
```

This is the code created for this example:

```
MODULE container
 CONTAINS
 SUBROUTINE invoke_0(a, f1, f2, m1, m2, istp, qr)
   ...
```

(continues on next page)

(continued from previous page)

```
    CALL psy_data_3%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", &
                             0, 0)
    DO cell=1,mesh%get_last_halo_cell(1)
      CALL testkern_code(...)
    END DO
    ...
    CALL psy_data_2%PreStart("multi_functions_multi_invokes_psy", &
                     "invoke_0:testkern_code:r3", 0, 0)
    DO cell=1,mesh%get_last_halo_cell(1)
      CALL testkern_code(...)
    END DO
    ...
    CALL psy_data_2%PostEnd()
    CALL psy_data_1%PostEnd()
    ...
    DO cell=1,mesh%get_last_halo_cell(1)
      CALL testkern_qr_code(...)
    END DO
    ...
    CALL psy_data_3%PostEnd()
    ...
  END SUBROUTINE invoke_0
END MODULE container
```

CHAPTER

TWENTYFOUR

PSY KERNEL EXTRACTOR (PSYKE)

24.1 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.

24.1.1 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 PSyData API). 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.

24.1.2 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

As noted in the *Distributed Memory* section, support for distributed memory in PSyclone is currently limited to the *LFRic (Dynamo0.3) API*. Since the implementation generates calls to LFRic infrastructure (e.g. runtime checks for status of field halos), code extraction is not allowed when distributed memory is enabled.

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 GOcean1.0 API.

24.2 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])
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)
    2: Loop[type='',field_space='w2',it_space='cells', upper_bound='ncells']
        ...
```

(continues on next page)

(continued from previous page)

```
        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']
                ...
                Schedule[]
```

(continues on next page)

(continued from previous page)

```
                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])
schedule.view()
```

The generated code is now:

```
! ExtractStart
CALL extract_psy_data%PreStart("unknown-module", "setval_c", 1, 3)
CALL extract_psy_data%PreDeclareVariable("f2", f2)
CALL extract_psy_data%PreDeclareVariable("cell_post", cell)
CALL extract_psy_data%PreDeclareVariable("df_post", df)
CALL extract_psy_data%PreDeclareVariable("f3_post", f3)
...
CALL extract_psy_data%PreEndDeclaration
CALL extract_psy_data%ProvideVariable("f2", f2)
...
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()
  !
  CALL testkern_code_w2_only(nlayers, f3_proxy%data, f2_proxy%data, ndf_w2, undf_w2, map_
↪w2(:,cell))
```

(continues on next page)

(continued from previous page)

```
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("f3_post", f3)
CALL extract_psy_data%PostEnd
!
! ExtractEnd
```

Note: At this stage Built-Ins are not fully supported, resulting in `f2` being incorrectly detected as an input parameter, and not as an output parameter. This issue is tracked in #637.

Examples in `examples/lfric/eg12` directory demonstrate how to apply code extraction by utilising PSyclone transformation scripts (see *Examples* section for more information).

24.3 NetCDF Extraction Examples

PSyclone comes with example NetCDF-based extraction *libraries* in lib/extract/netcdf.

24.3.1 NetCDF Extraction for GOcean

The library in lib/extract/netcdf/dl_esm_inf implements the full PSyData API for use with the *GOcean1.0* dl_esm_inf infrastructure library. In order to compile this library, you must have NetCDF installed. When running the code, it will create a NetCDF file for the 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 NetCDF 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 stored as NetCDF 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`.

The output file contains the values of all variables used in the subroutine. The `GOceanExtractTrans` can automatically create a driver program which will read the NetCDF 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 NetCDF 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.

24.3.2 NetCDF Extraction for LFRic

The library in lib/extract/netcdf/lfric implements the full PSyData API for use with the *LFRic* infrastructure library. In order to compile this library, you must have NetCDF installed. When running the code, it will create a NetCDF file for the instrumented code region.

As in the case of e.g. *read-only verification*, this library uses the pared-down LFRic infrastructure located in a clone of PSyclone repository, `<PSYCLONEHOME>/src/psyclone/tests/test_files/dynamo0p3/infrastructure`. However, this needs to be changed for any user (for instance with PSyclone installation). Please refer to the relevant `README.md` documentation on how to build and link this library.

Note: Driver creation in LFRic is not yet fully supported, and is tracked in issue #1392.

CHAPTER

TWENTYFIVE

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.

25.1 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]
DEFAULTAPI = dynamo0.3
DEFAULTSTUBAPI = dynamo0.3
DISTRIBUTED_MEMORY = true
REPRODUCIBLE_REDUCTIONS = false
REPROD_PAD_SIZE = 8
```

(continues on next page)

(continued from previous page)

```
PSYIR_ROOT_NAME = psyir_tmp
VALID_PSY_DATA_PREFIXES = profile, extract
```

and an optional API specific section, for example for the `dynamo0.3` section:

```
[dynamo0.3]
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
RUN_TIME_CHECKS = false
NUM_ANY_SPACE = 10
NUM_ANY_DISCONTINUOUS_SPACE = 10
```

or for `gocean1.0`:

```
[gocean1.0]
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.

25.1.1 `DEFAULT` Section

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

Entry	Description
DEFAULTAPI	The API that PSyclone assumes an Algorithm/Kernel conforms to if no API is specified. Must be one of the APIs supported by PSyclone ("dynamo0.1", "dynamo0.3", "gocean0.1", "gocean1.0" and "nemo"). If there is no API specified and there is only one API-specific section in the config file loaded, this API will be used. This value can be overwritten by the command line option '-api'. If there is no API entry in the config file, and '-api' is not specified on the command line, "dynamo0.3" is used as default.
DEFAULTSTUBAPI	The API that the kernel-stub generator assumes by default. Must be one of the stub-APIs supported by PSyclone ("dynamo0.3" only at this stage).
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.
REPRODUCIBLE_REDUCTIONS	Whether or not to generate code for reproducible OpenMP reductions (see *Reductions*) by default.
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.)
PSYIR_ROOT_NAME	The root for generated PSyIR symbol names if one is not supplied when creating a symbol. Defaults to "psyir_tmp".
VALID_PSY_DATA_PREFIXES	Which class prefixes are permitted in any PSyData-related transformations. See *PSyData API* for details.

25.1.2 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 APIs do not support increment or sum, so they only define three mappings for read, write, and readwrite.

25.1.3 `dynamo0.3` 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*.
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*.

25.1.4 `gocean1.0` 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.

25.1.5 `NEMO` Section

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

Entry	Description
mapping-TYPE	This declares a mapping for a certain loop level, specified as TYPE. Each value must have three key:value pairs. A value can be empty if it is not required or not known, but the key must still be specified. The required keys are: `var`: the variable name that indicates the loop level, `start`: the first loop iteration, and `stop`: the last loop iteration. Each loop detected by the NEMO API will be given one of the TYPE values specified in the configuration file. See the example below for more details.
index-order	Specifies the order in which loops are created when converting an implicit loop to an explicit loop. All values in this comma-separated list must have a corresponding `mapping-TYPE` value defined.

Below we show an example of the NEMO section of a PSyclone configuration file. Note how the values in `index-order` have corresponding mapping entries, e.g. `mapping-lon`, `mapping-lat` etc.:

```
mapping-lon = var: ji, start: 1, stop: jpi
mapping-lat = var: jj, start: 1, stop: jpj
mapping-levels = var: jk, start: 1, stop: jpk
mapping-tracers = var: jt, start: 1, stop:
mapping-unknown = var: , start: 1, stop:

index-order = lon, lat, levels, tracers
```

If a NEMO loop then uses `Do jj=...`, PSyclone will give this loop the type 'lat', because the loop uses the variable name specified in the configuration file for a loop of type 'lat'. The loop type can be accessed using `loop.loop_type`, i.e. in this example it will be `loop.loop_type == 'lat'`.

The entry `mapping-unknown` has an empty value for the key 'var'. This means that the type 'unknown' will be used for any loop that can not be mapped using any of the other variable names in the configuration file.

BIBLIOGRAPHY

[nem13] *NEMO Coding Conventions*. 2013. URL: https://forge.ipsl.jussieu.fr/nemo/attachment/wiki/Literature/NEMO_coding.conv_v3.pdf.

[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.

PYTHON MODULE INDEX

p

psyclone.alg_gen, 180
psyclone.generator, 177
psyclone.line_length, 180
psyclone.parse.algorithm, 178
psyclone.psyGen, 179

INDEX

A

Alg (*class in psyclone.alg_gen*), 180
ArgumentInterface (*class in psyclone.psyir.symbols*), 134

C

condition() (*psyclone.psyir.nodes.IfBlock method*), 129
container (*psyclone.psyGen.PSy property*), 179
ContainerSymbol (*class in psyclone.psyir.symbols*), 133
create() (*psyclone.psyir.nodes.Range static method*), 127

D

DataSymbol (*class in psyclone.psyir.symbols*), 133
dir_body() (*psyclone.psyir.nodes.RegionDirective method*), 129

E

else_body() (*psyclone.psyir.nodes.IfBlock method*), 129

F

FortLineLength (*class in psyclone.line_length*), 180

G

gen (*psyclone.alg_gen.Alg property*), 180
gen (*psyclone.psyGen.PSy property*), 179
generate() (*in module psyclone.generator*), 177
get_trans_name() (*psyclone.psyGen.TransInfo method*), 139
get_trans_num() (*psyclone.psyGen.TransInfo method*), 139

I

if_body() (*psyclone.psyir.nodes.IfBlock method*), 129
ImportInterface (*class in psyclone.psyir.symbols*), 134
indices() (*psyclone.psyir.nodes.ArrayReference method*), 129
invokes (*psyclone.psyGen.PSy property*), 179

L

length (*psyclone.line_length.FortLineLength property*), 180
lhs() (*psyclone.psyir.nodes.Assignment method*), 128
list (*psyclone.psyGen.TransInfo property*), 139
LocalInterface (*class in psyclone.psyir.symbols*), 134
long_lines() (*psyclone.line_length.FortLineLength method*), 180

M

member() (*psyclone.psyir.nodes.StructureReference method*), 129
module
    psyclone.alg_gen, 180
    psyclone.generator, 177
    psyclone.line_length, 180
    psyclone.parse.algorithm, 178
    psyclone.psyGen, 179

N

name (*psyclone.psyGen.PSy property*), 179
num_trans (*psyclone.psyGen.TransInfo property*), 139

P

parse() (*in module psyclone.parse.algorithm*), 178
process() (*psyclone.line_length.FortLineLength method*), 180
PSy (*class in psyclone.psyGen*), 179
psyclone.alg_gen
    module, 180
psyclone.generator
    module, 177
psyclone.line_length
    module, 180
psyclone.parse.algorithm
    module, 178
psyclone.psyGen
    module, 179

R

Range (*class in psyclone.psyir.nodes*), 126

rhs() (*psyclone.psyir.nodes.Assignment method*), 129
RoutineSymbol (*class in psyclone.psyir.symbols*), 133

S

start (*psyclone.psyir.nodes.Range property*), 127
step (*psyclone.psyir.nodes.Range property*), 127
stop (*psyclone.psyir.nodes.Range property*), 127
Symbol (*class in psyclone.psyir.symbols*), 132
SymbolTable (*class in psyclone.psyir.symbols*), 132

T

TransInfo (*class in psyclone.psyGen*), 139

U

UnresolvedInterface (*class in psyclone.psyir.symbols*), 134

W

walk() (*psyclone.psyir.nodes.Node method*), 129