PSyclone Documentation

Release 1.1.0

Rupert Ford and Andrew Porter

February 23, 2016

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

The GungHo project is designing and building the heart of the Met Office's next generation software (known as the dynamical core) using algorithms that will scale to millions of cores. The project is a collaboration between the Met Office, NERC (via NERC funded academics) and STFC, and the resultant software is expected to be operational in 2022.

The associated GungHo software infrastructure is being developed to support multiple meshes and element types thus allowing for future model development. GungHo is also proposing 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 and infrastructure routines) 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 GungHo project is developing the PSyclone code generation system which 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

ONE

GETTING GOING

1.1 Download

PSyclone is available for download from the GungHo repository. The latest release is 1.1.0.

```
svn co https://puma.nerc.ac.uk/svn/GungHo_svn/PSyclone/tags/vn1.1.0 PSyclone
```

The latest stable version is maintained on the trunk.

```
svn co https://puma.nerc.ac.uk/svn/GungHo_svn/PSyclone/trunk PSyclone
```

Hereon the location where you download PSyclone (including the PSyclone directory itself) will be refered to as <PSYCLONEHOME>

1.2 Dependencies

PSyclone is written in python so needs python to be installed on the target machine. PSyclone has been tested under python 2.6.5 and 2.7.3.

PSyclone immediately relies on two external libraries, f2py and pyparsing. To run the test suite you will require py.test.

1.2.1 f2py quick setup

The source code of f2py (revision 93) is provided with PSyclone in the sub-directory `f2py_93`.

To use f2py provided with PSyclone you can simply set up your PYTHONPATH variable to include this directory.

```
> export PYTHONPATH=<PSYCLONEHOME>/f2py_93:${PYTHONPATH}
```

If for some reason you need to install f2py yourself then see *f2py installation*.

1.2.2 pyparsing

PSyclone requires pyparsing, a library designed to allow parsers to be be built in Python. PSyclone uses pyparsing to parse fortran regular expressions as f2py does not fully parse these, (see http://pyparsing.wikispaces.com for more information).

PSyclone has been tested with pyparsing version 1.5.2 which is a relatively old version but is currently the version available in the Ubuntu software center.

You can test if pyparsing is already installed on your machine by typing `import pyparsing` from the python command line. If pyparsing is installed, this command will complete succesfully. If pyparsing is installed you can

check its version by typing `pyparsing.__version__` after succesfully importing it. Versions higher than 1.5.2 should work but have not been tested.

If pyparsing is not installed on your system you can install it from within Ubuntu using the software center (search for the "python-pyparsing" module in the software center and install). If you do not run Ubuntu you could follow the instructions here http://pyparsing.wikispaces.com/Download+and+Installation.

1.2.3 py.test

The PSyclone test suite uses py.test. This is not needed to use PSyclone but is useful to check whether PSyclone is working correctly on your system. You can test whether it is already installed by simply typing `py.test` at a shell prompt. If it is present you will get output that begins with

```
======================== test session starts ==================
```

If you do not have it then py.test can be installed from here http://pytest.org/latest/ (or specifically here http://pytest.org/latest/getting-started.html).

1.3 Environment

In order to use PSyclone (including running the test suite and building documentation) you will need to tell Python where to find the PSyclone source and the f2py source (if you have not already done the latter):

```
> export PYTHONPATH=<PSYCLONEHOME>/src:<PSYCLONEHOME>/f2py_93:${PYTHONPATH}
```

1.4 Test

Once you have the necessary dependencies installed and your environment configured, you can check that things are working by using the PSyclone test suite. These tests is not required and can be skipped if preferred:

```
> cd <PSYCLONEHOME>/src/tests
> py.test
```

If everything is working as expected then you should see output similar to:

```
============================= test session starts ==============================
platform linux2 -- Python 2.6.5 -- py-1.4.29 -- pytest-2.7.2
rootdir: /home/rupert/proj/GungHoSVN/PSyclone_r3373_scripts/src/tests, inifile:
collected 175 items

alggen_test.py .......xxxxxxxxxxx.
dynamo0p1_transformations_test.py .
dynamo0p3_test.py ....................................x
f2pygen_test.py ....x..........
generator_test.py ...................
ghproto_transformations_test.py x
gocean0p1_transformations_test.py .......
gocean1p0_test.py ....
gocean1p0_transformations_test.py ......................x........
parser_test.py ..........
psyGen_test.py ..............................

=================== 160 passed, 15 xfailed in 13.59 seconds ====================
```

1.5 Run

You are now ready to try running PSyclone on the examples. One way of doing this is to use the generator.py script:

```
> cd <PSYCLONEHOME>/src
> python ./generator.py
usage: generator.py [-h] [-oalg OALG] [-opsy OPSY] [-api API] [-s SCRIPT]
                    [-d DIRECTORY] [-l]
                    filename
generator.py: error: too few arguments
```

As indicated above, the generator.py 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.

Examples are provided in the examples directory. There are 3 subdirectories (dynamo, gocean and gunghoproto) corresponding to different API's that are supported by PSyclone. In this case we are going to use one of the dynamo examples

```
> cd <PSYCLONEHOME>/examples/dynamo/eg1
> python ../../../src/generator.py -api dynamo0.1 \
> -oalg dynamo_alg.f90 -opsy dynamo_psy.f90 dynamo.F90
```

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

You can also use the runme.py example to see the interactive API in action. This script contains:

```
from parse import parse
from psyGen import PSyFactory

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

# 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

# List the invokes that the PSy layer has
print psy.invokes.names

# Examine the 'schedule' (e.g. loop structure) that each
# invoke has
schedule=psy.invokes.get('invoke_0_v3_kernel_type').schedule
schedule.view()

schedule=psy.invokes.get('invoke_1_v3_solver_kernel_type').schedule
schedule.view()
```

It can be run non-interactively as follows:

```
> cd <PSYCLONEHOME>/example/dynamo/eg1
> python runme.py
```

However, to understand this example in more depth it is instructive to cut-and-paste from the runme.py file into your own, interactive python session:

```
> cd <PSYCLONEHOME>/example/dynamo/eg1
> python
```

In addition to the runme.py script, there is also runme_openmp.py which illustrates how one applies an OpenMP transform to a loop schedule within the PSy layer. The initial part of this script is the same as that of runme.py (above) and is therefore omitted here:

```
# List the various invokes that the PSy layer contains
print psy.invokes.names

# Get the loop schedule associated with one of these
# invokes
schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
schedule.view()

# Get the list of possible loop transformations
from psyGen import TransInfo
t=TransInfo()
print t.list

# Create an OpenMPLoop-transformation object
ol=t.get_trans_name('OMPLoopTrans')

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

# Replace the original loop schedule of the selected invoke
# with the new, transformed schedule
psy.invokes.get('invoke_v3_kernel_type')._schedule=new_schedule
# Generate the Fortran code for the new PSy layer
print psy.gen
```

CHAPTER

TWO

GENERATOR SCRIPT

The simplest way to run PSyclone is to use the generator.py script. This script is located in the <PSYCLONE-HOME>/src directory. The script takes an algorithm file as input and outputs modified algorithm code and generated PSy code. This section walks through its functionality. The *API* section gives a more concise overview.

2.1 Running

The generator.py script is designed to be run from the command line. It is typically invoked as an argument to the python interpreter:

```
> python <PSYCLONEHOME>/src/generator.py <args>
```

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

```
> python <PSYCLONEHOME>/src/generator.py -h
```

2.2 Basic Use

The simplest way to use generator.py is to provide it with an algorithm file.

```
> python <PSYCLONEHOME>/src/generator.py alg.f90
```

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

```
> cd <PSYCLONEHOME>/src
> python ./generator.py generator.py
'Parse Error: Error, program, module or subroutine not found in ast'
```

Warning: In the current version of PSyclone an unhelpful error ending with the following may occur

```
AttributeError: 'Line' object has no attribute 'tofortran'
```

This is due to the parser failing to parse the algorithm code and is very likely to be due to the algorithm code containing a syntax error.

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

2.3 Choosing the API

In the previous section we relied on PSyclone using the default API. The default API, along with the supported API's can be seen by running the generator.py script with the -h option.

If you use a particular API frequently and it is not the default then you can change the default by editing the config.py file in the <PSYCLONEHOME>/src directory.

If your code uses an API that is different to the default then you can specify this as an argument to the generator.py script.

```
> python <PSYCLONEHOME>/src/generator.py -api dynamo0.1 alg.f90
```

2.4 File output

By default the modified algorithm code and the generated PSy code is output to the terminal. These can be output to a file by using the -oalg <file> and -opsy <file> options respectively. For example, the following will output the generated psy code to a file but the algorithm code will be output to the terminal

```
> python <PSYCLONEHOME>/src/generator.py -opsy psy.f90 alg.f90
```

2.5 Kernel directory

When an algorithm file is parsed, the parser looks for the associated kernel files. The way this is done is that any kernel routine specified in 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))
end program no_use
> python <PSYCLONEHOME>/src/generator.py no_use.f90
"Parse Error: kernel call 'testkern_type' must be named in a use statement"
```

If the name of kernel is provided in a use statement then the parser will look for a file with the same name as 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))
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.

```
> python <PSYCLONEHOME>/src/generator.py use.f90
Kernel file 'testkern.[fF]90' not found in <location>
```

The -d option can be used to tell the generator.py script where to look for Kernel files. The -d option tells the generator.py script that the required Kernel code is somewhere within the specified directory hierarchy. The script will recurse from the specified directory path to look for the required file. There must be only one instance of the specified file within the specified directory:

```
> cd <PSYCLONEHOME>/src
> python ./generator.py -d . use.f90
More than one match for kernel file 'testkern.[fF]90' found!
> python ./generator.py -d tests/test_files/dynamo0p3 -api dynamo0.3 use.f90
[code output]
```

Note: The -d option is limited to a single directory. Therefore a current limitation in PSyclone is that all required Kernel files required by an algorithm file must exist within a directory hierarchy where their file names are unique.

2.6 Transformation script

By default the generator.py script 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.

2.7 Fortran line length

By default the generator.py script 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 the -l option is specified in the generator.py script, the output will be line wrapped so that the output line lengths are always within the 132 character limit.

The -l option also 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.

CHAPTER

THREE

KERNEL LAYER

In the PSyKAl separation of concerns, Kernel code (code which is created to run within the Kernel layer), works over a subset of a field (such as a column). 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.

3.1 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 `integrate_one_kernel` type specifies the appropriate metadata information describing the kernel code for the `gunghoproto` api.

```
module integrate_one_module
  use kernel_mod
  implicit none

  private
  public integrate_one_kernel
  public integrate_one_code

  type, extends(kernel_type) :: integrate_one_kernel
    type(arg) :: meta_args(2) = (/&
         arg(READ, (CG(1)*CG(1))**3, FE), &
         arg(SUM, R, FE)/)
    integer :: ITERATES_OVER = CELLS
    contains
    procedure, nopass :: code => integrate_one_code
  end type integrate_one_kernel

contains

  subroutine integrate_one_code(layers, p1dofm, X, R)
```

```
    integer, intent(in) :: layers
    integer, intent(in) :: p1dofm(6)
    real(dp), intent(in) :: X(3,*)
    real(dp), intent(inout) :: R
  end subroutine integrate_one_code

end module integrate_one_module
```

3.2 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 API's 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 API's will contain information about the arguments in an array called `meta_args`, a specification of what the kernel code iterates over in a variable called `iterates_over` and a reference to the kernel code as a type bound procedure.

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

CHAPTER

FOUR

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 and infrastructure routines) 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.

4.1 API

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

In an `invoke` call, the algorithm layer developer adds `call invoke()` to their code and within the content of the `invoke` call they add a reference to the required Kernel and the data to pass to it. For example,

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

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 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
...
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
```

CHAPTER

FIVE

PSY LAYER

In the PSyKAl separation of concerns, the PSy layer is responsible for linking together the Algorithm layer and Kernel layer. Its functional responsibilities are to

1. map the arguments supplied by an Algorithm `invoke` call to the arguments required by a Kernel call (as these will not have a one-to-one correspondance).
2. call the Kernel routine so that it covers the required iteration space and
3. 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.

5.1 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 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 metadata and by reading the order of kernels 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 parse import parse
from psyGen import PSyFactory

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

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

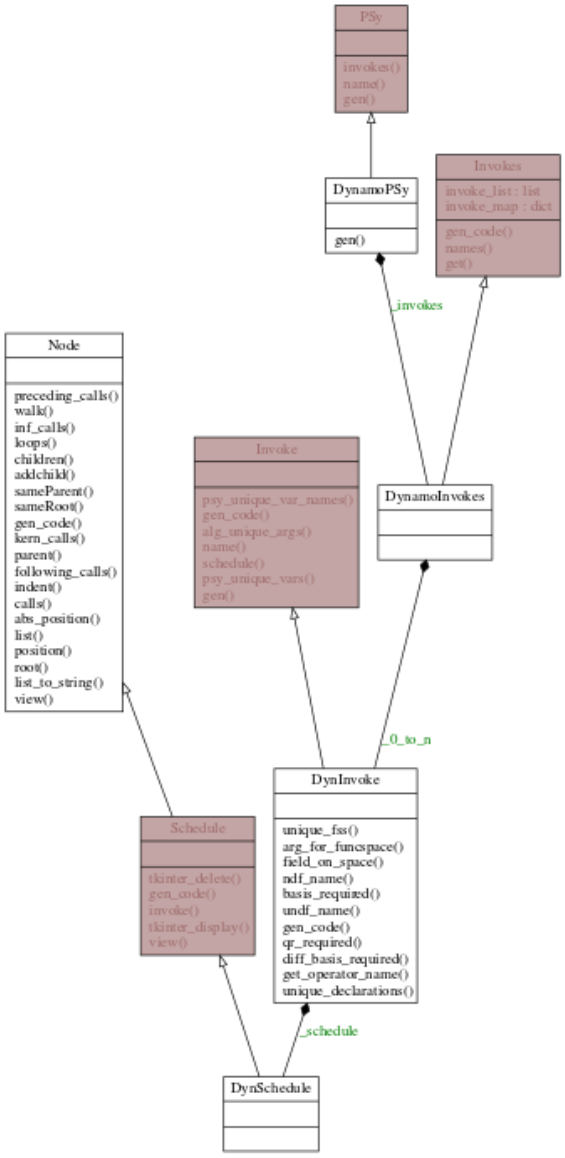
```
# Create the PSy-layer object using the invokeInfo
psy = PSyFactory(api).create(invokeInfo)
# Generate the Fortran code for the PSy layer
print psy.gen
```

5.2 Structure

PSyclone provides a hierarchy of base classes which specific API's can subclass to support their particular API. All API's implemented so far, follow this hierarchy.

At the top level is the **PSy** class. The PSy class has an **Invokes** class. The **Invokes** class can contain one or more **Invoke** classes (one for each invoke in the algorithm layer). Each **Invoke** class has a **Schedule** class.

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.



5.3 API

class 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.parse()` as its input and stores this in a way suitable for optimisation and code generation.

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

For example:

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

inline(*module*)
inline all kernel subroutines into the module that are marked for inlining. Avoid inlining the same kernel more than once.

class psyGen.**Invokes**(*alg_calls*, *Invoke*)
Manage the invoke calls

class psyGen.**Invoke**(*alg_invocation*, *idx*, *Schedule*, *reserved_names*=[])
Manage an individual invoke call

class psyGen.**Schedule**(*Loop*, *Inf*, *alg_calls*=[])
Stores schedule information for an invocation call. Schedules can be optimised using transformations.

```
>>> from parse import parse
>>> ast, info = parse("algorithm.f90")
>>> from psyGen import PSyFactory
>>> api = "..."
>>> psy = PSyFactory(api).create(info)
>>> invokes = psy.invokes
>>> invokes.names
>>> invoke = invokes.get("name")
>>> schedule = invoke.schedule
>>> schedule.view()
```

5.4 Schedule

A PSy **Schedule** object consists of a tree of objects which can be used to describe the required schedule for a PSy layer subroutine which is called by the algorithm layer and itself calls one or more Kernels. These objects can currently be a **Loop**, a **Kernel** or a **Directive** (of various types). 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) schedule.

This "vanilla" schedule 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 schedule allows the PSy layer to be optimised for a particular architecture (by applying optimisations such as blocking, loop merging, inlining 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.

CHAPTER

SIX

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 layer for a particular architecture, however transformations could be added for other reasons, such as to aid debugging or for performance monitoring.

6.1 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

class 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 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)

list

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

num_trans

return the number of transformations available

6.2 Available

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 API's e.g. dynamo). Currently these different types of transformation are indicated by their names.

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

class `transformations.`**`KernelModuleInlineTrans`**

Switches on, or switches off, the inlining of a Kernel subroutine into the PSy layer module. For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> inline_trans = KernelModuleInlineTrans()
>>>
>>> ischedule, _ = inline_trans.apply(schedule.children[0].children[0])
>>> ischedule.view()
```

> **Warning:** For this transformation to work correctly, the Kernel subroutine must only use data that is passed in by argument, declared locally or included via use association within the subroutine. Two examples where in-lining will not work correctly are:
>
> 1. A variable is declared within the module that `contains` the Kernel subroutine and is then accessed within that Kernel;
> 2. A variable is included via use association at the module level and accessed within the Kernel subroutine.
>
> *There are currently no checks that these rules are being followed when in-lining so the onus is on the user to ensure correctness.*

`apply`(*node*, *inline=True*)

Checks that the node is of the correct type (a Kernel) then marks the Kernel to be inlined, or not, depending on the value of the inline argument. If the inline argument is not passed the Kernel is marked to be inlined.

`name`

Returns the name of this transformation as a string

class `transformations.`**`LoopFuseTrans`**

Provides a loop-fuse transformation. For example:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("dynamo.F90")
>>> psy=PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
>>>
>>> from transformations import LoopFuseTrans
>>> trans=LoopFuseTrans()
>>> new_schedule,memento=trans.apply(schedule.children[0],
                                     schedule.children[1])
>>> new_schedule.view()
```

`apply`(*node1*, *node2*)

Fuse the loops represented by `node1` and `node2`

`name`

Returns the name of this transformation as a string

class `transformations.`**`ColourTrans`**

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

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> ctrans = ColourTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>>     cschedule, _ = ctrans.apply(child)
>>>
>>> csched.view()
```

apply(*node*)
Converts the Loop represented by `node` into a nested loop where the outer loop is over colours and the inner loop is over points of that colour.

name
Returns the name of this transformation as a string

class `transformations.`**OMPLoopTrans**(*omp_schedule='static'*)
Adds an orphaned OpenMP directive to a loop. i.e. the directive must be inside the scope of some other OMP Parallel REGION. This condition is tested at code-generation time. For example:

```
>>> from parse import parse,ParseError
>>> from psyGen import PSyFactory,GenerationError
>>> api="gocean1.0"
>>> filename="nemolite2d_alg.f90"
>>> ast,invokeInfo=parse(filename,api=api,invoke_name="invoke")
>>> psy=PSyFactory(api).create(invokeInfo)
>>> print psy.invokes.names
>>>
>>> from psyGen import TransInfo
>>> t=TransInfo()
>>> ltrans = t.get_trans_name('OMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule=psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule=schedule
>>>
# Apply the OpenMP Loop transformation to *every* loop
# in the schedule
>>> for child in schedule.children:
>>>     newschedule,memento=ltrans.apply(child)
>>>     schedule = newschedule
>>>
# Enclose all of these loops within a single OpenMP
# PARALLEL region
>>> rtrans.omp_schedule("dynamic,1")
>>> newschedule,memento = rtrans.apply(schedule.children)
>>>
>>>
```

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

```
!$OMP DO
do ...
   ...
```

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

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

name
Returns the name of this transformation as a string

omp_schedule
Returns the OpenMP schedule that will be specified by this transformation. The default schedule is 'static'

class `transformations.`**OMPParallelTrans**
Create an OpenMP PARALLEL region by inserting directives. For example:

```
>>> from parse import parse, ParseError
>>> from psyGen import PSyFactory, GenerationError
>>> api="gocean1.0"
>>> filename="nemolite2d_alg.f90"
>>> ast,invokeInfo=parse(filename,api=api,invoke_name="invoke")
>>> psy=PSyFactory(api).create(invokeInfo)
>>>
>>> from psyGen import TransInfo
>>> t=TransInfo()
>>> ltrans = t.get_trans_name('GOceanOMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule=psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule=schedule
>>>
>>> # Apply the OpenMP Loop transformation to *every* loop
>>> # in the schedule
>>> for child in schedule.children:
>>>     newschedule,memento=ltrans.apply(child)
>>>     schedule = newschedule
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> newschedule, _ = rtrans.apply(schedule.children)
>>> newschedule.view()
```

apply(*nodes*)
Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Loops in the schedule within a single OpenMP region. `nodes` can be a single Node or a list of Nodes.

name
Returns the name of this transformation as a string

class `transformations.`**OMPParallelLoopTrans**(*omp_schedule='static'*)
Adds an OpenMP PARALLEL DO directive to a loop.

For example:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("dynamo.F90")
>>> psy=PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
```

```
>>>
>>> from transformations import OMPParallelLoopTrans
>>> trans=OMPParallelLoopTrans()
>>> new_schedule,memento=trans.apply(schedule.children[0])
>>> new_schedule.view()
```

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

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

name
Returns the name of this transformation as a string

6.3 Applying

Transformations can be applied either interactively or through a script.

6.3.1 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 parse import parse
from psyGen import PSyFactory

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

# 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 ...

```
# Get the list of possible loop transformations
from psyGen import TransInfo
t = TransInfo()
print t.list

# Create an OpenMPLoop-transformation
ol = t.get_trans_name('OMPParallelLoopTrans')

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

# Replace the original loop schedule of the selected invoke
# with the new, transformed schedule
invoke.schedule=new_schedule

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

More examples of use of the interactive application of transformations can be found in the runme*.py files within the examples/dynamo/eg1 and examples/dynamo/eg2 directories. Some simple examples of the use of transformations are also given in the previous section.

6.3.2 Script

PSyclone provides a Python script (**generator.py**) 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:

```
> python generator.py algspec.f90
> python generator.py -oalg alg.f90 -opsy psy.f90 -api dynamo0.3 algspec.f90
```

The generator.py 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:

```
> python generator.py -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 ...

```
> python generator.py -s ./opt.py algspec.f90
> python generator.py -s ../scripts/opt.py algspec.f90
> python generator.py -s /home/me/PSyclone/scripts/opt.py algspec.f90
```

PSyclone also provides the same functionality via a function (which is what the **generator.py** script calls internally)

generator.**generate**(*filename*, *api=''*, *kernel_path=''*, *script_name=None*, *line_length=False*)

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

Parameters

- **filename** (*str*) – The file containing the algorithm specification.
- **kernel_path** (*str*) – The directory from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification)
- **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).
- **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.

Returns The algorithm code and the psy code.

Return type ast

Raises IOError if the filename or search path do not exist

For example:

```
>>> from generator import generate
>>> psy, alg = generate("algspec.f90")
>>> psy, alg = generate("algspec.f90", kernel_path="src/kernels")
>>> psy, alg = generate("algspec.f90", script_name="optimise.py")
>>> psy, alg = generate("algspec.f90", line_length=True)
```

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 transformations import OMPParallelLoopTrans
    invoke = psy.invokes.get('invoke_0_v3_kernel_type')
    schedule = invoke.schedule
    ol = OMPParallelLoopTrans()
    new_schedule, _ = ol.apply(schedule.children[0])
    invoke.schedule = new_schedule
    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) contained within the PSy layer.

An example of the use of transformations scripts can be found in the examples/dynamo/eg3 directory. Please read the examples/dynamo/README file first as it explains how to run the example.

CHAPTER

SEVEN

DYNAMO0.3 API

This section describes the dynamo0.3 application programming interface (API). This API explains what a user needs to write in order to make use of the dynamo0.3 API in PSyclone.

As with all PSyclone API's the dynamo0.3 API specifies how a user needs to write the algorithm layer and the kernel layer to allow PSyclone to generate the PSy layer. These algorithm and kernel API's are discussed separately in the following sections.

7.1 Algorithm

Note: To be written.

7.2 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. This section explains the dynamo0.3-specific metadata and subroutine arguments.

7.2.1 Metadata

The code below outlines the elements of the dynamo0.3 API kernel metadata, 1) 'meta_args', 2) 'meta_funcs', 3) 'iterates_over' and 4) 'procedure'.

```
type, public, extends(kernel_type) :: my_kernel_type
  type(arg_type) :: meta_args(...) = (/ ... /)
  type(func_type) :: meta_funcs(...) = (/ ... /)
  integer :: iterates_over = cells
contains
  procedure :: my_kernel_code
end type
```

These 4 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 (metadata contained within the brackets of an `arg_type` entry), describes either a **scalar**, a **field** or an **operator**.

The first argument-metadata entry describes whether the data that is being passed is for a real scalar (`GH_RSCALAR`), an integer scalar (`GH_ISCALAR`), a field (`GH_FIELD`) or an operator (`GH_OPERATOR`). This information is mandatory.

Additionally, argument-metadata can be used to describe a vector of fields (see the *Algorithm* section for more details). If so, the size of the vector is specified using the notation `GH_FIELD*N`, where `N` is the size of the vector.

As an example, the following `meta_args` metadata describes 4 entries, the first is a real 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_RSCALAR, ...),                                        &
     arg_type(GH_FIELD, ... ),                                         &
     arg_type(GH_FIELD*3, ... ),                                       &
     arg_type(GH_OPERATOR, ...)                                        &
     /)
```

The second entry to argument-metadata (information contained within the brackets of an `arg_type`) describes how the Kernel makes use of the data being passed into it. There are 3 possible values of this metadata `GH_WRITE`, `GH_READ` and `GH_INC`. `GH_WRITE` indicates the data is modified in the Kernel before (optionally) being read. `GH_READ` indicates that the data is read and left unmodified. `GH_INC` **explanation TBD**.

For example:

```
type(arg_type) :: meta_args(4) = (/                                    &
     arg_type(GH_RSCALAR, GH_READ),                                    &
     arg_type(GH_FIELD, GH_INC, ... ),                                 &
     arg_type(GH_FIELD*3, GH_WRITE, ... ),                             &
     arg_type(GH_OPERATOR, GH_READ, ...)                               &
     /)
```

For a scalar the argument metadata contains only these two 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 3rd and 4th arguments describe the `to` and `from` function spaces respectively. In the case of a field the 3rd argument specifies the function space that the field lives on. Supported function spaces are `w0`, `w1`, `w2`, `w3`, `wtheta`, `w2h` and `w2v`.

For example:

```
type(arg_type) :: meta_args(3) = (/                                    &
     arg_type(GH_FIELD, GH_INC, W1),                                   &
```

```
     arg_type(GH_FIELD*3, GH_WRITE, W2H),                              &
     arg_type(GH_OPERATOR, GH_READ, W1, W2H)                           &
     /)
```

It may be that a Kernel is written such that a field and/or operators may be on any function space. In this case the metadata should be specified as being one of `any_space_1`, `any_space_2`, ..., `any_space_9`. 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.

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_INC, ANY_SPACE_1 ),                         &
     arg_type(GH_FIELD*3, GH_WRITE, ANY_SPACE_2 ),                     &
     arg_type(GH_OPERATOR, GH_READ, ANY_SPACE_1, ANY_SPACE_2)          &
     /)
```

Finally, field metadata supports an optional 4th argument which specifies that the field 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,extent)
```

where `type` may be one of `X1D`, `Y1D`, `CROSS` or `REGION` and extent is an integer which specifies the maximum distance from the central point that a stencil extends.

For example, the following stencil:

```
| 4 | 2 | 1 | 3 | 5 |
```

would be declared as

```
STENCIL(X1D,2)
```

the following stencil

```
|   |   | 9 |   |   |
|   |   | 5 |   |   |
| 6 | 2 | 1 | 3 | 7 |
|   |   | 4 |   |   |
|   |   | 8 |   |   |
```

would be declared as

```
STENCIL(CROSS,2)
```

and the following stencil (all adjacent cells)

```
| 9 | 5 | 8 |
| 2 | 1 | 3 |
| 6 | 4 | 7 |
```

would be declared as

```
STENCIL(REGION,1)
```

Below is an example of stencil information within the full kernel metadata.

```
type(arg_type) :: meta_args(3) = (/                                    &
     arg_type(GH_FIELD, GH_INC, W1),                                   &
     arg_type(GH_FIELD, GH_READ, W2H, STENCIL(REGION,1)),              &
     arg_type(GH_OPERATOR, GH_READ, W1, W2H)                           &
     /)
```

meta_funcs

Note: To be written.

iterates over

The 3rd type of metadata provided is `ITERATES_OVER`. This specifies that the Kernel has been written with the assumption that it is iterating over the specified entity. Currently this only has one valid value which is `CELLS`.

Procedure

The 4th and final type of metadata is `procedure` metadata. This specifies the name of the Kernel subroutine that this metadata describes.

For example:

```
procedure :: my_kernel_subroutine
```

7.2.2 Subroutine

Rules

Kernel arguments follow a set of rules which have been specified for the dynamo0.3 API. These rules are encoded in the `_create_arg_list()` method within the `DynKern` class in the `dynamo0p3.py` file. The rules, along with PSyclone's naming conventions, are:

1. If an operator is passed then include the `cells` argument. `cells` is an integer and has intent `in`.
2. Include `nlayers`, the number of layers in a column. `nlayers` is an integer and has intent `in`.
3. For each scalar/field/vector_field/operator in the order specified by the meta_args metadata:
   (a) 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).
   (b) 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 real array of type `r_def` and dimensioned as the unique degrees of freedom for the space that the field operates on. This value is passed in separately. Again, the intent is determined from the metadata (see *meta_args*).
   (c) 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).

(d) if the current entry is an operator then first include a dimension size. This is an integer. The name of this size is <operator_name>"_ncell_3d". Next include the operator. This is a real array of type r_def and is 3 dimensional. The first two dimensions are the local degrees of freedom for the to and from function spaces respectively. The third dimension is the dimension size mentioned before. 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)

   (a) Include the number of local degrees of freedom for the function space. This is an integer and has intent in. The name of this argument is "ndf_"<field_function_space>.

   (b) If there is a field on this space

      i. Include the unique number of degrees of freedom for the function space. This is an integer and has intent in. The name of this argument is "undf_"<field_function_space>.

      ii. Include the dofmap for this function space. This is an integer array with intent in. It has one dimension sized by the local degrees of freedom for the function space.

   (c) For each operation on the function space (basis, diff_basis, orientation) in the order specified in the metadata

      i. If it is a basis function, include the associated argument. This is a real array of kind r_def with intent in. It has four dimensions. The first dimension is 1 or 3 depending on the function space (w0=1,w1=3,w2=3,w3=1,wtheta=1,w2h=3,w2v=3). The second dimension is the local degrees of freedom for the function space. The third argument is the quadrature rule size which is currently named nqp_h and the fourth argument is the quadrature rule size which is currently named nqp_v. The name of the argument is "basis_"<field_function_space>

      ii. If it is a differential basis function, include the associated argument. The sizes and dimensions are the same as the basis function except for the size of the first dimension which is sized as 1 or 3 depending on different function space rules (w0=3,w1=3,w2=1,w3=1,wtheta=3,w2h=1,w2v=1). The name of the argument is "diff_basis_"<field_function_space>.

      iii. If is an orientation array, include the associated argument. The argument is an integer array with intent in. There is one dimension of size the local degrees of freedom for the function space. The name of the array is "orientation_"<field_function_space>.

5. if Quadrature is required (this is the case if any of the function spaces require a basis or differential basis function)

   (a) include nqp_h. This is an integer scalar with intent in.

   (b) include nqp_v. This is an integer scalar with intent in.

   (c) include wh. This is a real array of kind r_def with intent in. It has one dimension of size nqp_h.

   (d) include wv. This is a real array of kind r_def with intent in. It has one dimension of size nqp_v.

7.3 Conventions

There is a convention in the dynamo0.3 API kernel code 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 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.

7.4 Transformations

Note: To be written.

CHAPTER

EIGHT

GOCEAN1.0 API

8.1 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 API's, 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 API's are discussed separately in the sections below. Before these we describe the functionality provided by the GOcean Library.

8.2 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.

8.2.1 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(ARAKAWA_C,                              &
                       (/BC_EXTERNAL,BC_EXTERNAL,BC_NONE/), &
                       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 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.

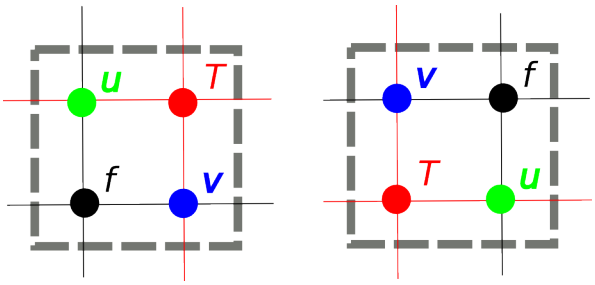
3. The 'index offset' - the convention used for indexing into offset fields.

Three types of boundary condition are currently supported:

Name	Description
BC_NONE	No boundary conditions are applied.
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*).
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 `OFFSET_SW` and `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.

8.2.2 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, U_POINTS)
sshn_v = r2d_field(model_grid, V_POINTS)
sshn_t = r2d_field(model_grid, 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 (`U_POINTS`, `V_POINTS`, `T_POINTS` or `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.

8.2.3 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 GOLib
  use field_mod ! From GOLib
  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
  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(ARAKAWA_C,                          &
                         (/BC_EXTERNAL,BC_EXTERNAL,BC_NONE/), &
                          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, U_POINTS)
  vn_fld = r2d_field(model_grid, V_POINTS)

  ! Velocity components 'after' (next time step)
  ua_fld = r2d_field(model_grid, U_POINTS)
  va_fld = r2d_field(model_grid, 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))

  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.

8.3 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*).

8.3.1 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 meta-data (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 GOLib
  use grid_mod         ! From GOLib
  use field_mod        ! From GOLib
  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.

8.4 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. This section explains the meta-data and subroutine arguments that are specific to the GOcean 1.0 API.

8.4.1 Metadata

The meta-data 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(arg), dimension(...) :: meta_args = (/ ... /)
   integer :: iterates_over = ...
   integer :: index_offset = ...
contains
  procedure, nopass :: code => my_kernel_code
end type my_kernel_type
```

These four meta-data 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 **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 `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 `arg`:

```
type(arg) :: meta_args(2) = (/                                    &
     arg( ... ),                                                  &
     arg( ... )                                                   &
     /)
```

Argument-metadata (metadata contained within the brackets of an `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(arg) :: meta_args(4) = (/                                    &
     arg(WRITE, ... ),                                            &
     arg(READ, ... ),                                             &
     arg(READ, ... ),                                             &
     arg(READ, ...)                                               &
     /)
```

The second entry to argument-metadata (information contained within the brackets of an `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 meta-data 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 `CU`, `CV`, `CF` and `CT`. GOcean Kernels can also take scalar quantities as arguments. Since these do not live on grid-points they are specified as either `R_SCALAR` or `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(arg) :: meta_args(4) = (/                                    &
     arg(WRITE, CT, ... ),                                        &
     arg(READ,  CU, ... ),                                        &
     arg(READ,  R_SCALAR, ... ),                                  &
     arg(READ,  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:

Name	Description	Type
grid_area_t	Cell area at T point	Real array, rank=2
grid_area_u	Cell area at U point	Real array, rank=2
grid_area_v	Cell area at V point	Real array, rank=2
grid_mask_t	T-point mask (1=wet, 0=dry)	Integer array, rank=2
grid_dx_t	Grid spacing in x at T points	Real array, rank=2
grid_dx_u	Grid spacing in x at U points	Real array, rank=2
grid_dx_v	Grid spacing in x at V points	Real array, rank=2
grid_dy_t	Grid spacing in y at T points	Real array, rank=2
grid_dy_u	Grid spacing in y at U points	Real array, rank=2
grid_dy_v	Grid spacing in y at V points	Real array, rank=2
grid_lat_u	Latitude of U points (gphiu)	Real array, rank=2
grid_lat_v	Latitude of V points (gphiv)	Real array, rank=2
grid_dx_const	Grid spacing in x if constant	Real, scalar
grid_dy_const	Grid spacing in y if constant	Real, scalar

These are stored in a dictionary named `GRID_PROPERTY_DICT` at the top of the `gocean1p0.py` file. All of the rank-two arrays have the first rank as longitude (x) and the second as latitude (y).

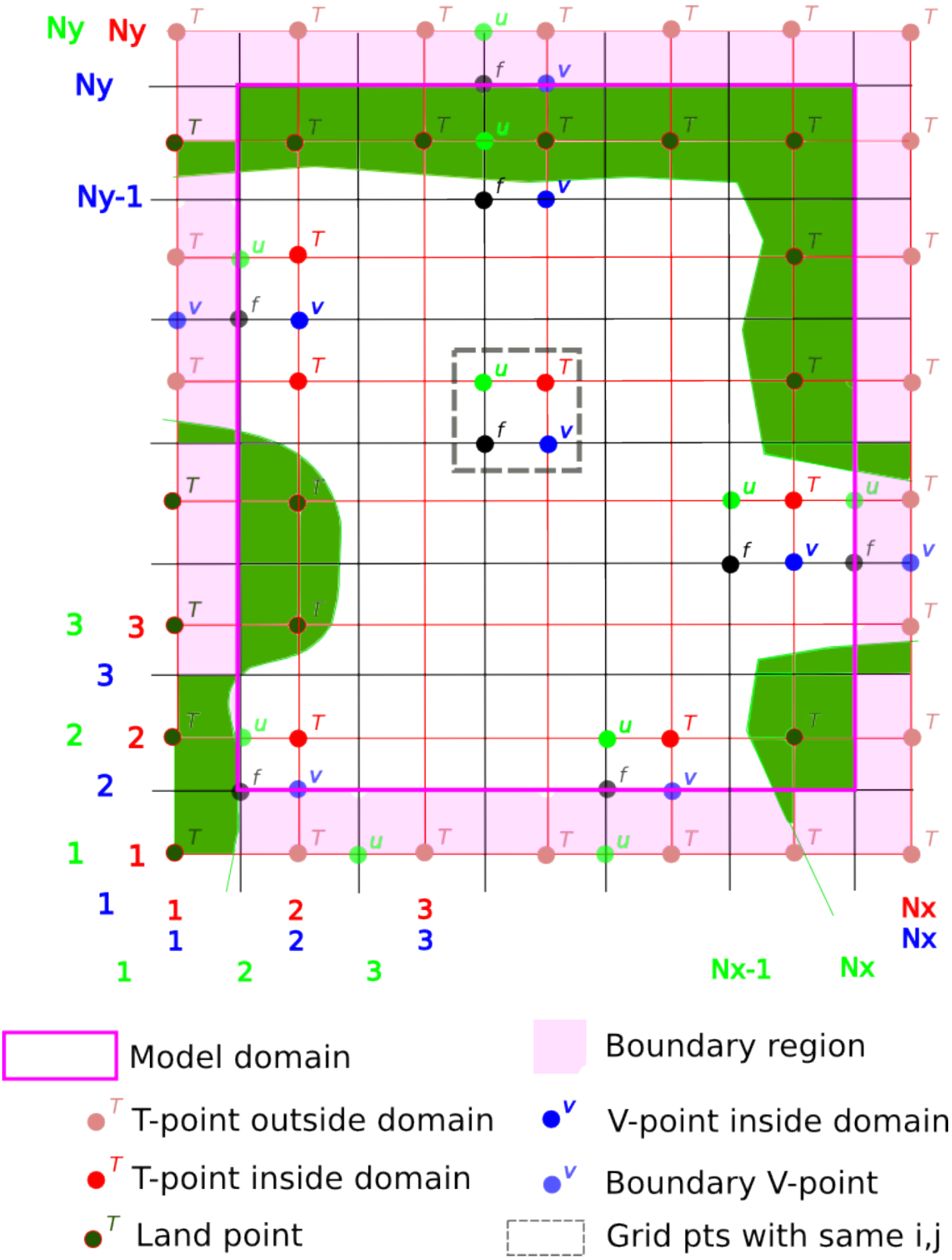
For scalar and field arguments the argument meta-data contains a third argument which must be 'POINTWISE'. This is not currently used in this version of the GOcean API. For grid-property arguments there is no third meta-data argument. Therefore, the full argument meta-data for our previous example will be:

```
type(arg) :: meta_args(4) = (/                                    &
     arg(WRITE, CT,        POINTWISE),                            &
```

```
    arg(READ,  CU,       POINTWISE),                    &
    arg(READ,  R_SCALAR, POINTWISE),                    &
    arg(READ,  GRID_AREA_U)                             &
    /)
```

Iterates Over

The second element of kernel meta-data 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. The supported values are: `INTERNAL_PTS`, `EXTERNAL_PTS` and `ALL_PTS`. These may be understood by considering the following diagram of an example model configuration:



INTERNAL_PTS are then those points that are within the Model domain (fuscia box), EXTERNAL_PTS are those

outside the domain and ALL_PTS encompasses all grid points in the model. The chosen value is specified in the kernel-meta data like so:

```
integer :: iterates_over = INTERNAL_PTS
```

Index Offset

The third element of kernel meta-data, 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; OFFSET_NE, OFFSET_SW. The scheme used by a kernel is specified in the meta-data as, e.g.:

```
integer :: index_offset = 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 meta-data is procedure meta-data. This specifies the name of the Kernel Fortran subroutine that this meta-data describes.

For example:

```
procedure :: my_kernel_code
```

8.4.2 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:
   (a) For a field; the field array itself. A field array is a real array of kind wp and rank two. The first rank is longitude (x) and the second latitude (y).
   (b) For a scalar; the variable itself. A real scalar is of kind wp.
   (c) 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 meta-data for this kernel is:

```
 type, extends(kernel_type) :: bc_solid_u
   type(arg), dimension(2) :: meta_args =  &
        (/ arg(WRITE, CU, POINTWISE),      &
           arg(READ,      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 = ALL_PTS

   integer :: index_offset = 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.

8.5 Conventions

There is a convention in the GOcean 1.0 API kernel code 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 does not require this convention to be followed in the GOcean 1.0 API.

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.

8.6 Transformations

In this section we describe the transformations that are specific to the GOcean 1.0 API. For an overview of transformations in general see *Transformations*.

class transformations.**GOceanLoopFuseTrans**
Performs error checking (that the loops are over the same grid-point type) before calling the LoopFuseTrans.apply() method of the base class in order to fuse two GOcean loops.

apply(*node1*, *node2*)
Fuse the two GOcean loops represented by node1 and node2

name
Returns the name of this transformation as a string

class transformations.**GOceanOMPParallelLoopTrans**(*omp_schedule='static'*)
GOcean specific OpenMP Do loop transformation. Adds GOcean specific validity checks (that supplied Loop is an inner or outer loop). Actual transformation is done by base class.

apply(*node*)
Perform GOcean-specific loop validity checks then call OMPParallelLoopTrans.apply().

name
Returns the name of this transformation as a string

class transformations.**GOceanOMPLoopTrans**(*omp_schedule='static'*)
GOcean-specific orphan OpenMP loop transformation. Adds GOcean specific validity checks (that the node is either an inner or outer Loop). Actual transformation is done by base class.

apply(*node*)
Perform GOcean specific loop validity checks then call *:py:meth:'OMPLoopTrans.apply*.

name
Returns the name of this transformation as a string

class transformations.**GOConstLoopBoundsTrans**
Switch on (or off) the use of constant loop bounds within a GOSchedule. In the absence of constant loop bounds, PSyclone will generate loops where the bounds are obtained by de-referencing a field object, e.g.:

```
DO j = my_field%grid%internal%ystart, my_field%grid%internal%ystop
```

Some compilers are able to produce more efficient code if they are provided with information on the relative trip-counts of the loops within an Invoke. With constant loop bounds switched on, PSyclone generates code like:

```
ny = my_field%grid%simulation_domain%ystop
...
DO j = 1, ny-1
```

In practice, the application of the constant loop bounds looks something like, e.g.:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> import os
>>> TEST_API = "gocean1.0"
>>> _,info = parse(os.path.join("tests", "test_files", "gocean1p0",
>>>                             "single_invoke.f90"),
>>>                api=TEST_API)
>>> psy = PSyFactory(TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0_compute_cu')
>>> schedule = invoke.schedule
>>>
>>> from transformations import GOConstLoopBoundsTrans
>>> clbtrans = GOConstLoopBoundsTrans()
>>>
>>> newsched, _ = clbtrans.apply(schedule)
```

```
>>> # or, to turn off const. looop bounds:
>>> # newsched, _ = clbtrans.apply(schedule, const_bounds=False)
>>>
>>> newsched.view()
```

name
Return the name of the Transformation as a string

CHAPTER

NINE

STUB GENERATION

9.1 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 from the PSyclone src directory

```
> python ./genkernelstub.py my_file.f90
```

9.2 Introduction

PSyclone provides a kernel stub generator for the dynamo0.3 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 dynamo0.3 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 dynamo0.3 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 dynamo0.3 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.

9.3 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. For the remainder of this section the location where PSyclone is installed (including the PSyclone directory itself) will be referred to as `<PSYCLONEHOME>`.

The easiest way to use the stub generator is to use the supplied script called `genkernelstub.py`, which is located in the `src` directory:

```
> cd <PSYCLONEHOME>/src
> python ./genkernelstub.py
usage: genkernelstub.py [-h] [-o OUTFILE] [-api API] filename
genkernelstub.py: error: too few arguments
```

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

```
>  python genkernelstub.py -h
usage: genkernelstub.py [-h] [-o OUTFILE] [-api API] 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
```

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

The `-o`, or `--outfile` option allows the user 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.

9.4 Kernels

Any dynamo0.3 kernel can be used as input to the stub generator. Example Kernels can be found in the dynamo repository or, for more simple cases, in the `tests/test_files/dynamo0p3` directory. In the latter directory the majority start with `testkern`. The exceptions are: `simple.f90`, `ru_kernel_mod.f90` and `matrix_vector_mm_mod.F90`. The following test kernels can be used to generate kernel stub code:

```
tests/test_files/dynamo0p3/testkern_chi_2.F90
tests/test_files/dynamo0p3/testkern_chi.F90
tests/test_files/dynamo0p3/testkern_operator_mod.f90
tests/test_files/dynamo0p3/testkern_operator_nofield_mod.f90
tests/test_files/dynamo0p3/testkern_orientation.F90
tests/test_files/dynamo0p3/testkern_operator_orient_mod.f90
tests/test_files/dynamo0p3/testkern_qr.F90
tests/test_files/dynamo0p3/ru_kernel_mod.f90
tests/test_files/dynamo0p3/simple.f90
```

9.5 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
type, extends(kernel_type) :: simple_type
    type(arg_type), dimension(1) :: meta_args =  &
        (/ arg_type(gh_field,gh_write,w1) /)
    integer, parameter :: iterates_over = cells
  contains
    procedure() :: 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 dynamo0.3 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:

```
> python genkernelstub.py tests/test_files/dynamo0p3/simple.f90
```

we get the following kernel stub output:

```
MODULE simple_code_mod
  IMPLICIT NONE
  CONTAINS
  SUBROUTINE simple_code(nlayers, field_1_w1, ndf_w1, undf_w1, map_w1)
    USE constants_mod, ONLY: r_def
    IMPLICIT NONE
    INTEGER, intent(in) :: nlayers
    INTEGER, intent(in) :: undf_w1
    REAL(KIND=r_def), intent(out), dimension(undf_w1) :: field_1_w1
    INTEGER, intent(in) :: ndf_w1
    INTEGER, intent(in), dimension(ndf_w1) :: map_w1
  END SUBROUTINE simple_code
END MODULE simple_code_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 module 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 (undf) which is also passed into the kernel (the fourth argument). The naming convention is to call each field a field, followed by it's 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 dynamo 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 is given below.

```
module ru_kernel_mod
type, public, extends(kernel_type) :: ru_kernel_type
  private
  type(arg_type) :: meta_args(4) = (/                                    &
       arg_type(GH_FIELD,   GH_INC,  W2),                                &
       arg_type(GH_FIELD,   GH_READ, W3),                                &
       arg_type(GH_FIELD,   GH_READ, W0),                                &
       arg_type(GH_FIELD*3, 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 :: iterates_over = CELLS
contains
  procedure, nopass ::ru_code
end type
contains
subroutine ru_code()
end subroutine ru_code
end module ru_kernel_mod
```

If we run the kernel stub generator on this example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/ru_kernel_mod.f90
```

we obtain the following output:

```
MODULE ru_code_mod
  IMPLICIT NONE
  CONTAINS
  SUBROUTINE ru_code_code(nlayers, field_1_w2, field_2_w3, field_3_w0, field_4_w0_v1, field_4_w0_v2,
    USE constants_mod, ONLY: r_def
    IMPLICIT NONE
    INTEGER, intent(in) :: nlayers
    INTEGER, intent(in) :: undf_w2
    INTEGER, intent(in) :: undf_w3
    INTEGER, intent(in) :: undf_w0
    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_3_w0
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_4_w0_v1
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_4_w0_v2
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_4_w0_v3
    INTEGER, intent(in) :: ndf_w2
    INTEGER, intent(in), dimension(ndf_w2) :: map_w2
    REAL(KIND=r_def), intent(in), dimension(3,ndf_w2,nqp_h,nqp_v) :: basis_w2
    REAL(KIND=r_def), intent(in), dimension(1,ndf_w2,nqp_h,nqp_v) :: diff_basis_w2
    INTEGER, intent(in), dimension(ndf_w2,2) :: boundary_dofs_w2
    INTEGER, intent(in) :: ndf_w3
    INTEGER, intent(in), dimension(ndf_w3) :: map_w3
    REAL(KIND=r_def), intent(in), dimension(1,ndf_w3,nqp_h,nqp_v) :: basis_w3
    INTEGER, intent(in) :: ndf_w0
    INTEGER, intent(in), dimension(ndf_w0) :: map_w0
    REAL(KIND=r_def), intent(in), dimension(1,ndf_w0,nqp_h,nqp_v) :: basis_w0
    REAL(KIND=r_def), intent(in), dimension(3,ndf_w0,nqp_h,nqp_v) :: diff_basis_w0
    INTEGER, intent(in) :: nqp_h, nqp_v
    REAL(KIND=r_def), intent(in), dimension(nqp_h) :: wh
    REAL(KIND=r_def), intent(in), dimension(nqp_v) :: wv
  END SUBROUTINE ru_code_code
END MODULE ru_code_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 1-4 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.

9.6 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/matrix_vector_mm_mod.f90
tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90
tests/test_files/dynamo0p3/testkern_any_space_2_mod.f90
tests/test_files/dynamo0p3/testkern.F90
tests/test_files/dynamo0p3/testkern_invalid_fortran.F90
tests/test_files/dynamo0p3/testkern_no_datatype.F90
tests/test_files/dynamo0p3/testkern_short_name.F90
```

`testkern_invalid_fortran.F90`, `testkern_no_datatype.F90`, `testkern_short_name.F90`, `testkern.F90` and `matrix_vector_mm_mod.f90` are designed to be invalid for PSyclone testing purposes and should produce appropriate errors. For example:

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

`any_space` is not currently supported in the stub generator so `testkern_any_space_1_mod.f90` and `testkern_any_space_2_mod.f90` should fail with appropriate warnings because of that. For example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90
Error: "Generation Error: Unknown space, expecting one of ['w0', 'w1', 'w2', 'w3',
'wtheta', 'w2h', 'w2v'] but found 'any_space_1'"
```

CHAPTER

TEN

LINE LENGTH

By default PSyclone 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.

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.

10.1 Script

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

10.2 Interactive

When using PSyclone interactively the line length of the input algorithm and Kernel files can be checked by setting the `parse.parse()` function's "line_length" argument to "True".

```
>>> from parse 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 `parse.parse()` function.

```
>>> from 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 generator import generate
>>> from 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
```

10.3 Limitations

The `line_length.FortLineLength` class is only partially aware of fortran syntax. It has been designed to work in the cases that are anticipated to produce long lines when generating code using PSyclone, i.e. call and subroutine arguments, use statements, directives and declarations.

If other types of line are too long (e.g. an assignment) then an exception will be raised. This situation is not expected as the code generator should not produce assignment lines that are longer than 132 characters.

Other known situations that could cause an instance of the `line_length.FortLineLength` class to fail are

1. When an inline comment is used at the end of line to make it too long. However, PSyclone does not generate such code.
2. When a long line includes a string. The `line_length.FortLineLength` class is not aware of strings and therefore will not produce correct code if it attempts to line break within a string. Again it is believed that PSyclone will currently not generate long lines with strings in them.

CHAPTER

ELEVEN

API

generator.py

-h

-oalg <filename>

-opsy <filename>

-api <api>

-s <script>

-d <directory>

-l

Command line version of the generator. -h prints out the command line options. If -oalg or -opsy are not provided then the generated code is printed to stdout, otherwise they are output to the specified file name. -api specifies the particular api to use. -s allows a script to be called which can modify (typically optimise) the PSy layer. -d specifies a directory to recursively search to find the associated kernel files. -l limits the maximum line length of the fortran output to 132 characters. -l uses a relatively simple algorithm which in pathological cases may produce incorrect output, so it is recommended to only use this option if necessary. generator.py Uses the `generator.generate()` function to generate the code. Please see the run documentation for more details.

For example:

```
> python generator.py algspec.f90
> python generator.py -oalg alg.f90 -opsy psy.f90 -api dynamo0.3 algspec.f90
> python generator.py -d ../kernel -s opt.py algspec.f90
> python generator.py -s ../scripts/opt.py -l algspec.f90
```

This module provides the main PSyclone command line script which takes an algorithm file as input and produces modified algorithm code and generated PSy code. A function is also provided which has the same functionality as the command line script but can be called from within another Python program.

generator.**generate**(*filename*, *api=''*, *kernel_path=''*, *script_name=None*, *line_length=False*)

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

Parameters

- **filename** (*str*) – The file containing the algorithm specification.
- **kernel_path** (*str*) – The directory from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification)

- **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).
- **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.

Returns The algorithm code and the psy code.

Return type ast

Raises IOError if the filename or search path do not exist

For example:

```
>>> from generator import generate
>>> psy, alg = generate("algspec.f90")
>>> psy, alg = generate("algspec.f90", kernel_path="src/kernels")
>>> psy, alg = generate("algspec.f90", script_name="optimise.py")
>>> psy, alg = generate("algspec.f90", line_length=True)
```

11.1 The parse module

parse.**parse**(*alg_filename*, *api=''*, *invoke_name='invoke'*, *inf_name='inf'*, *kernel_path=''*, *line_length=False*)

Takes a GungHo algorithm specification as input and outputs an AST of this specification and an object containing information about the invocation calls in the algorithm specification and any associated kernel implementations.

Parameters

- **alg_filename** (*str*) – The file containing the algorithm specification.
- **invoke_name** (*str*) – The expected name of the invocation calls in the algorithm specification
- **inf_name** (*str*) – The expected module name of any required infrastructure routines.
- **kernel_path** (*str*) – The path to search for kernel source files (if different from the location of the algorithm source).
- **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.

Return type ast,invoke_info

Raises

- **IOError** – if the filename or search path does not exist
- **ParseError** – if there is an error in the parsing
- **RuntimeError** – if there is an error in the parsing

For example:

```
>>> from parse import parse
>>> ast,info=parse("argspec.F90")
```

11.2 The transformations module

This module provides the various transformations that can be applied to the schedule associated with an invoke(). There are both general and API-specific transformation classes in this module where the latter typically apply API-specific checks before calling the base class for the actual transformation.

class `transformations.`**`ColourTrans`**

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

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> ctrans = ColourTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>>     cschedule, _ = ctrans.apply(child)
>>>
>>> csched.view()
```

`apply`(*node*)

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

`name`

Returns the name of this transformation as a string

class `transformations.`**`Dynamo0p3ColourTrans`**

Split a Dynamo 0.3 loop into colours so that it can be parallelised. For example:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> import transformations
>>> import os
>>> import pytest
>>>
>>> TEST_API = "dynamo0.3"
>>> _,info=parse(os.path.join(os.path.dirname(os.path.abspath(__file__)),
>>>              "tests", "test_files", "dynamo0p3",
>>>              "4.6_multikernel_invokes.f90"),
>>>              api=TEST_API)
>>> psy = PSyFactory(TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0')
>>> schedule = invoke.schedule
>>>
>>> ctrans = Dynamo0p3ColourTrans()
>>> otrans = DynamoOMPParallelLoopTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>>     cschedule, _ = ctrans.apply(child)
>>>
>>> # Then apply OpenMP to each of the colour loops
>>> schedule = cschedule
>>> for child in schedule.children:
>>>     newsched, _ = otrans.apply(child.children[0])
```

```
>>>
>>> newsched.view()
```

Colouring in the Dynamo 0.3 API is subject to the following rules:

- Any kernel which has a field with 'INC' access must be coloured UNLESS that field is on w3
- A kernel may have at most one field with 'INC' access
- Attempting to colour a kernel that updates a field on w3 (with INC access) should result in PSyclone issuing a warning
- Attempting to colour any kernel that doesn't have a field with INC access should also result in PSyclone issuing a warning.
- A separate colour map will be required for each field that is coloured (if an invoke contains >1 kernel call)

apply(*node*)
Performs Dynamo0.3-specific error checking and then uses the parent class to convert the Loop represented by `node` into a nested loop where the outer loop is over colours and the inner loop is over points of that colour.

name
Returns the name of this transformation as a string

class `transformations`.**Dynamo0p3OMPLoopTrans**(*omp_schedule='static'*)
Dynamo 0.3 specific orphan OpenMP loop transformation. Adds Dynamo-specific validity checks. Actual transformation is done by `base class`.

apply(*node*)
Perform Dynamo 0.3 specific loop validity checks then call `OMPLoopTrans.apply()`.

name
Returns the name of this transformation as a string

class `transformations`.**DynamoLoopFuseTrans**
Performs error checking before calling the `apply()` method of the `base class` in order to fuse two Dynamo loops.

apply(*node1*, *node2*)
Fuse the two Dynamo loops represented by `node1` and `node2`

name
Returns the name of this transformation as a string

class `transformations`.**DynamoOMPParallelLoopTrans**(*omp_schedule='static'*)
Dynamo-specific OpenMP loop transformation. Adds Dynamo specific validity checks. Actual transformation is done by the `base class`.

apply(*node*)
Perform Dynamo specific loop validity checks then call the `apply()` method of the `base class`.

name
Returns the name of this transformation as a string

class `transformations`.**GOConstLoopBoundsTrans**
Switch on (or off) the use of constant loop bounds within a GOSchedule. In the absence of constant loop bounds, PSyclone will generate loops where the bounds are obtained by de-referencing a field object, e.g.:

```
DO j = my_field%grid%internal%ystart, my_field%grid%internal%ystop
```

Some compilers are able to produce more efficient code if they are provided with information on the relative trip-counts of the loops within an Invoke. With constant loop bounds switched on, PSyclone generates code like:

```
ny = my_field%grid%simulation_domain%ystop
...
DO j = 1, ny-1
```

In practice, the application of the constant loop bounds looks something like, e.g.:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> import os
>>> TEST_API = "gocean1.0"
>>> _,info = parse(os.path.join("tests", "test_files", "gocean1p0",
>>>                             "single_invoke.f90"),
>>>                api=TEST_API)
>>> psy = PSyFactory(TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0_compute_cu')
>>> schedule = invoke.schedule
>>>
>>> from transformations import GOConstLoopBoundsTrans
>>> clbtrans = GOConstLoopBoundsTrans()
>>>
>>> newsched, _ = clbtrans.apply(schedule)
>>> # or, to turn off const. looop bounds:
>>> # newsched, _ = clbtrans.apply(schedule, const_bounds=False)
>>>
>>> newsched.view()
```

name
Return the name of the Transformation as a string

class transformations.**GOceanLoopFuseTrans**
Performs error checking (that the loops are over the same grid-point type) before calling the LoopFuseTrans.apply() method of the base class in order to fuse two GOcean loops.

apply(*node1*, *node2*)
Fuse the two GOcean loops represented by node1 and node2

name
Returns the name of this transformation as a string

class transformations.**GOceanOMPLoopTrans**(*omp_schedule='static'*)
GOcean-specific orphan OpenMP loop transformation. Adds GOcean specific validity checks (that the node is either an inner or outer Loop). Actual transformation is done by base class.

apply(*node*)
Perform GOcean specific loop validity checks then call *:py:meth:'OMPLoopTrans.apply*.

name
Returns the name of this transformation as a string

class transformations.**GOceanOMPParallelLoopTrans**(*omp_schedule='static'*)
GOcean specific OpenMP Do loop transformation. Adds GOcean specific validity checks (that supplied Loop is an inner or outer loop). Actual transformation is done by base class.

apply(*node*)
Perform GOcean-specific loop validity checks then call OMPParallelLoopTrans.apply().

name
Returns the name of this transformation as a string

class transformations.**KernelModuleInlineTrans**
Switches on, or switches off, the inlining of a Kernel subroutine into the PSy layer module. For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> inline_trans = KernelModuleInlineTrans()
>>>
>>> ischedule, _ = inline_trans.apply(schedule.children[0].children[0])
>>> ischedule.view()
```

> **Warning:** For this transformation to work correctly, the Kernel subroutine must only use data that is passed in by argument, declared locally or included via use association within the subroutine. Two examples where in-lining will not work correctly are:
> 1. A variable is declared within the module that `contains` the Kernel subroutine and is then accessed within that Kernel;
> 2. A variable is included via use association at the module level and accessed within the Kernel subroutine.
>
> *There are currently no checks that these rules are being followed when in-lining so the onus is on the user to ensure correctness.*

apply(*node*, *inline=True*)
Checks that the node is of the correct type (a Kernel) then marks the Kernel to be inlined, or not, depending on the value of the inline argument. If the inline argument is not passed the Kernel is marked to be inlined.

name
Returns the name of this transformation as a string

class transformations.**LoopFuseTrans**
Provides a loop-fuse transformation. For example:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("dynamo.F90")
>>> psy=PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
>>>
>>> from transformations import LoopFuseTrans
>>> trans=LoopFuseTrans()
>>> new_schedule,memento=trans.apply(schedule.children[0],
                                     schedule.children[1])
>>> new_schedule.view()
```

apply(*node1*, *node2*)
Fuse the loops represented by `node1` and `node2`

name
Returns the name of this transformation as a string

class transformations.**OMPLoopTrans**(*omp_schedule='static'*)
Adds an orphaned OpenMP directive to a loop. i.e. the directive must be inside the scope of some other OMP Parallel REGION. This condition is tested at code-generation time. For example:

```
>>> from parse import parse,ParseError
>>> from psyGen import PSyFactory,GenerationError
```

```
>>> api="gocean1.0"
>>> filename="nemolite2d_alg.f90"
>>> ast,invokeInfo=parse(filename,api=api,invoke_name="invoke")
>>> psy=PSyFactory(api).create(invokeInfo)
>>> print psy.invokes.names
>>>
>>> from psyGen import TransInfo
>>> t=TransInfo()
>>> ltrans = t.get_trans_name('OMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule=psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule=schedule
>>>
# Apply the OpenMP Loop transformation to *every* loop
# in the schedule
>>> for child in schedule.children:
>>>     newschedule,memento=ltrans.apply(child)
>>>     schedule = newschedule
>>>
# Enclose all of these loops within a single OpenMP
# PARALLEL region
>>> rtrans.omp_schedule("dynamic,1")
>>> newschedule,memento = rtrans.apply(schedule.children)
>>>
>>>
```

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

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

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

name
Returns the name of this transformation as a string

omp_schedule
Returns the OpenMP schedule that will be specified by this transformation. The default schedule is 'static'

class `transformations.`**OMPParallelLoopTrans**(*omp_schedule='static'*)
Adds an OpenMP PARALLEL DO directive to a loop.

For example:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("dynamo.F90")
>>> psy=PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
>>>
>>> from transformations import OMPParallelLoopTrans
```

```
>>> trans=OMPParallelLoopTrans()
>>> new_schedule,memento=trans.apply(schedule.children[0])
>>> new_schedule.view()
```

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

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

name
Returns the name of this transformation as a string

class transformations.**OMPParallelTrans**
Create an OpenMP PARALLEL region by inserting directives. For example:

```
>>> from parse import parse, ParseError
>>> from psyGen import PSyFactory, GenerationError
>>> api="gocean1.0"
>>> filename="nemolite2d_alg.f90"
>>> ast,invokeInfo=parse(filename,api=api,invoke_name="invoke")
>>> psy=PSyFactory(api).create(invokeInfo)
>>>
>>> from psyGen import TransInfo
>>> t=TransInfo()
>>> ltrans = t.get_trans_name('GOceanOMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule=psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule=schedule
>>>
>>> # Apply the OpenMP Loop transformation to *every* loop
>>> # in the schedule
>>> for child in schedule.children:
>>>     newschedule,memento=ltrans.apply(child)
>>>     schedule = newschedule
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> newschedule, _ = rtrans.apply(schedule.children)
>>> newschedule.view()
```

apply(*nodes*)
Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Loops in the schedule within a single OpenMP region. `nodes` can be a single Node or a list of Nodes.

name
Returns the name of this transformation as a string

exception transformations.**TransformationError**(*value*)
Provides a PSyclone-specific error class for errors found during code transformation operations.

11.3 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 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.parse() as its input and stores this in a way suitable for optimisation and code generation.

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

For example:

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

inline(*module*)

inline all kernel subroutines into the module that are marked for inlining. Avoid inlining the same kernel more than once.

11.4 The algGen module

class algGen.**Alg**(*ast*, *psy*)

Generate a modified algorithm code for a single algorithm specification. Takes the ast of the algorithm specification output from the function parse.parse() and an instance of the psyGen.PSy class as input.

Parameters

- **ast** (*ast*) – An object containing an ast of the algorithm specification which was produced by the function parse.parse().
- **psy** (*PSy*) – An object (psyGen.PSy) containing information about the PSy layer.

For example:

```
>>> from parse import parse
>>> ast,info=parse("argspec.F90")
>>> from psyGen import PSy
>>> psy=PSy(info)
>>> from algGen import Alg
>>> alg=Alg(ast,psy)
>>> print(alg.gen)
```

gen

Generate modified algorithm code

Return type ast

11.5 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 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

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 wrappe appropriately

CHAPTER

TWELVE

F2PY INSTALLATION

PSyclone requires version 3 of f2py, a library designed to allow fortran to be called from python (see http://code.google.com/p/f2py/wiki/F2PYDevelopment for more information). PSyclone makes use of the fortran parser (fparser) contained within.

The source code of f2py (revision 93) is provided with PSyclone in the sub-directory `f2py_93`. If you would prefer to install f2py rather than simply use it as is (see the previous section) then the rest of this section explains how to do this.

f2py uses the numpy distutils package to install. In version 1.6.1 of distutils (currently the default in Ubuntu) distutils supports interactive setup. In this case to install f2py using gfortran and gcc (for example) you can perform the following (where "cgcc", "fgfortran", "1" and "2" are interactive commands to setup.py)

```
> cd f2py_93
> sudo ./setup.py
cgcc
fgfortran
1
> sudo ./setup.py
cgcc
fgfortran
2
```

For later versions of distutils (1.8.0 has been tested) where the interactive setup has been disabled you can perform the following (using g95 and gcc in this case):

```
> cd f2py_93
> sudo ./setup.py build -fcompiler=g95 -ccompiler=gcc
> sudo ./setup.py install
```

For more information about possible build options you can use the available help:

```
> ./setup.py --help
> ./setup.py build --help
> ./setup.py build --help-fcompiler
```

In particular, if you do not have root access then the python modules can be installed in your user account by specifying –user to the install command:

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

This causes the software to be installed under ${HOME}/.local/

CHAPTER

THIRTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

a

g

l

p

t

Symbols

A

C

D

F

G

I

K

L

N

O

P

S

T