PSyclone Documentation

Release 1.0.2

Rupert Ford and Andrew Porter

October 28, 2015

1 Getting Going **3**
1.1 Download . . . 3
1.2 Dependencies . . . 3
1.3 Environment . . . 4
1.4 Test . . . 4
1.5 Run . . . 5

2 Kernel layer **7**
2.1 API . . . 7
2.2 Metadata . . . 8

3 Algorithm layer **9**
3.1 API . . . 9

4 PSy layer **11**
4.1 Code Generation . . . 11
4.2 Structure . . . 12
4.3 API . . . 13
4.4 Schedule . . . 14

5 Transformations **15**
5.1 Finding . . . 15
5.2 Available . . . 15
5.3 Applying . . . 19

6 API **23**
6.1 The parse module . . . 24
6.2 The transformations module . . . 24
6.3 The psyGen module . . . 30
6.4 The algGen module . . . 30

7 Stub Generation **33**
7.1 Quick Start . . . 33
7.2 Introduction . . . 33
7.3 Use . . . 34
7.4 Kernels . . . 34
7.5 Example . . . 35
7.6 Errors . . . 37
7.7 Rules . . . 38

8 f2py installation **41**

9 Indices and tables 43

Python Module Index 45

Index 47

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 1.0.2.

```
svn co https://puma.nerc.ac.uk/svn/GungHo_svn/PSyclone/tags/vn1.0.2 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]
                    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

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).

2.1 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
```

2.2 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

THREE

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.

3.1 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

FOUR

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.

4.1 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)
```

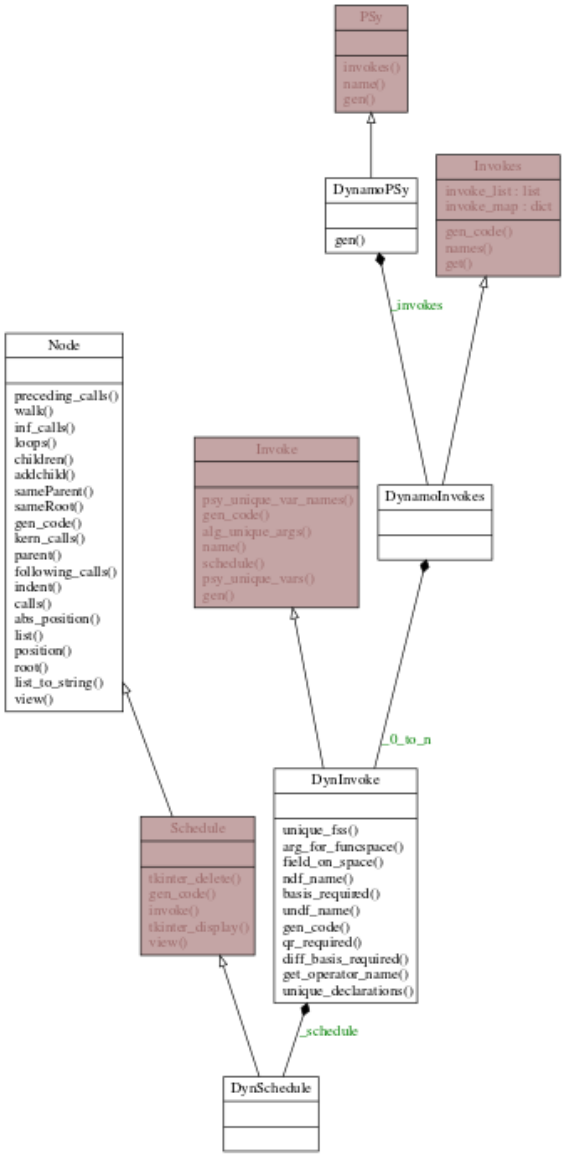
```
# Create the PSy-layer object using the invokeInfo
psy = PSyFactory(api).create(invokeInfo)
# Generate the Fortran code for the PSy layer
print psy.gen
```

4.2 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.



4.3 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()
```

4.4 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

FIVE

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.

5.1 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

5.2 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 inlining will not work correctly are 1) if a variable is declared as a module variable and used within the Kernel subroutine 2) if a variable is included via use association at the module level and used within the Kernel subroutine. There are currently no checks that these rules are being followed when inlining 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

5.3 Applying

Transformations can be applied either interactively or through a script.

5.3.1 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.

5.3.2 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*)

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).

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")
```

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

SIX

API

generator.py

-h

-oalg <filename>

-opsy <filename>

-api <api>

-s <script>

-d <directory>

-r <filename>

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. 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 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*)

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).

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")
```

6.1 The parse module

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

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).

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")
```

6.2 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, 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 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. 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. 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 inlining will not work correctly are 1) if a variable is declared as a module variable and used within the Kernel subroutine 2) if a variable is included via use association at the module level and used within the Kernel subroutine. There are currently no checks that these rules are being followed when inlining 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.

6.3 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.

6.4 The algGen module

class `algGen.`**Alg**(*ast*, *psy*)
Generate a modified algorithm code for a single algorithm specification. Takes the ast of the algorithm specifi-

cation 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

CHAPTER

SEVEN

STUB GENERATION

7.1 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
```

7.2 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.

7.3 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.

7.4 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
```

7.5 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.

7.6 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' but found 'any_space_1'"
```

7.7 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 field/vector_field/operator in the order specified by the meta_args metadata.
   (a) 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. The intent is determined from the metadata (see later for an explanation).
   (b) 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).
   (c) 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 `from` and `to` function spaces (currently in an unknown order). The third dimension is the dimension size mentioned before. At the moment the `from` and `to` function spaces must be the same in the generator. The name of the operator is `"op_"<argument_position>`. Again the intent is determined from the metadata and is explained later.
4. For each function space in the order they appear in the metadata arguments
   (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). 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). 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>`.

(d) If the Kernel name is `ru_code` and the function space is `w2` then include a two dimensional integer array with intent `in`. The first dimension of this array is the local degrees of freedom and the second dimension is 2. The name of this array is boundary_dofs_w2. This is a kernel hack to support ru_kernel as boundary conditions are not currently dealt with properly. This hack will be removed in the next revision of the API.

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`.

CHAPTER

EIGHT

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

NINE

INDICES AND TABLES

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

a

algGen, 30

g

generator, 23

p

parse, 24
psyGen, 30

t

transformations, 24

Symbols

-api <api>
    generator command line option, 23
-d <directory>
    generator command line option, 23
-h
    generator command line option, 23
-oalg <filename>
    generator command line option, 23
-opsy <filename>
    generator command line option, 23
-r <filename>
    generator command line option, 23
-s <script>
    generator command line option, 23

A

Alg (class in algGen), 30
algGen (module), 30
apply() (transformations.ColourTrans method), 25
apply() (transformations.Dynamo0p3ColourTrans method), 26
apply() (transformations.Dynamo0p3OMPLoopTrans method), 26
apply() (transformations.DynamoLoopFuseTrans method), 26
apply() (transformations.DynamoOMPParallelLoopTrans method), 26
apply() (transformations.GOceanLoopFuseTrans method), 27
apply() (transformations.GOceanOMPLoopTrans method), 27
apply() (transformations.GOceanOMPParallelLoopTrans method), 27
apply() (transformations.KernelModuleInlineTrans method), 27
apply() (transformations.LoopFuseTrans method), 28
apply() (transformations.OMPLoopTrans method), 28
apply() (transformations.OMPParallelLoopTrans method), 29
apply() (transformations.OMPParallelTrans method), 30

C

ColourTrans (class in transformations), 24

D

Dynamo0p3ColourTrans (class in transformations), 25
Dynamo0p3OMPLoopTrans (class in transformations), 26
DynamoLoopFuseTrans (class in transformations), 26
DynamoOMPParallelLoopTrans (class in transformations), 26

G

gen (algGen.Alg attribute), 31
generate() (in module generator), 23
generator (module), 23
generator command line option
    -api <api>, 23
    -d <directory>, 23
    -h, 23
    -oalg <filename>, 23
    -opsy <filename>, 23
    -r <filename>, 23
    -s <script>, 23
get_trans_name() (psyGen.TransInfo method), 15
get_trans_num() (psyGen.TransInfo method), 15
GOceanLoopFuseTrans (class in transformations), 27
GOceanOMPLoopTrans (class in transformations), 27
GOceanOMPParallelLoopTrans (class in transformations), 27
GOConstLoopBoundsTrans (class in transformations), 26

I

inline() (psyGen.PSy method), 30
Invoke (class in psyGen), 14
Invokes (class in psyGen), 14

K

KernelModuleInlineTrans (class in transformations), 27

L

list (psyGen.TransInfo attribute), 15
LoopFuseTrans (class in transformations), 27

N

name (transformations.ColourTrans attribute), 25
name (transformations.Dynamo0p3ColourTrans attribute), 26
name (transformations.Dynamo0p3OMPLoopTrans attribute), 26
name (transformations.DynamoLoopFuseTrans attribute), 26
name (transformations.DynamoOMPParallelLoopTrans attribute), 26
name (transformations.GOceanLoopFuseTrans attribute), 27
name (transformations.GOceanOMPLoopTrans attribute), 27
name (transformations.GOceanOMPParallelLoopTrans attribute), 27
name (transformations.GOConstLoopBoundsTrans attribute), 27
name (transformations.KernelModuleInlineTrans attribute), 27
name (transformations.LoopFuseTrans attribute), 28
name (transformations.OMPLoopTrans attribute), 29
name (transformations.OMPParallelLoopTrans attribute), 29
name (transformations.OMPParallelTrans attribute), 30
num_trans (psyGen.TransInfo attribute), 15

O

omp_schedule (transformations.OMPLoopTrans attribute), 29
OMPLoopTrans (class in transformations), 28
OMPParallelLoopTrans (class in transformations), 29
OMPParallelTrans (class in transformations), 29

P

parse (module), 24
parse() (in module parse), 24
PSy (class in psyGen), 30
psyGen (module), 30

S

Schedule (class in psyGen), 14

T

TransformationError, 30
transformations (module), 24
TransInfo (class in psyGen), 15