PSyclone Developer Guide

Release 2.3.1

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

Jun 17, 2022

CONTENTS

A PSyclone developer will, by definition, be working with the GitHub PSyclone repository rather than installing a released version from pypi (using e.g. `pip install psyclone`). This section describes the general set-up necessary when using PSyclone in this way. It also describes some of the development practises of the PSyclone project.

More detailed instructions for the Ubuntu and OpenSUSE Linux distributions may be found in the *System-specific Developer Set-up* Section.

CHAPTER

ONE

INSTALLATION

Although PSyclone releases always work with a released version of fparser, the same is not always true of other versions (e.g. the HEAD of the master branch). For those versions of PSyclone requiring fparser functionality that is not yet in a release, we use the git submodule feature such that the PSyclone repository always has a link to the correct version of fparser. In order to obtain this version the PSyclone repository must be cloned with the `--recursive` flag:

```
> git clone --recursive https://github.com/stfc/PSyclone.git
```

Alternatively, if you already have a local clone of the PSyclone github repository then doing:

```
> cd <PSYCLONEHOME>
> git submodule init
> git submodule update --init
```

will fetch the fparser submodule. Failure to do this will mean that for example the `<PSYCLONEHOME>/external/fparser` directory will be empty.

Note that after cloning the repository from GitHub, the local copy will be on the master branch. If you are working with some other branch then this must be checked out by doing:

```
> cd <PSYCLONEHOME>
> git checkout <BRANCH_NAME>
```

Once the above steps have been performed, the `<PSYCLONEHOME>/external/fparser` directory will contain the correct version of the fparser code. This can then be installed using `pip`:

```
> cd <PSYCLONEHOME>/external/fparser
> pip install --user .
```

Once you have the correct version of fparser installed you are ready to install PSyclone itself. Again, the simplest way of doing this is to use `pip`:

```
> cd <PSYCLONEHOME>
> pip install --user -e .
```

where `-e` requests an 'editable' installation so that changes to the PSyclone source are immediately reflected in the installed package. (For alternatives to using pip please see the Getting Going section.)

CHAPTER

TWO

TEST SUITE

The PSyclone test suite is integral to the development process and all new code must be covered (i.e. executed) by one or more tests. As described in Getting Going, the test suite is written for use with `pytest`.

Tests should be run from the `<PSYCLONEHOME>/src/psyclone/tests` directory, from which all tests in subdirectories will be automatically found and started. If only a subset of all tests need to be run, `pytest` can be invoked from the corresponding subdirectory or with that subdirectory or filename as an argument.

2.1 Fixtures

Various pytest fixtures (https://docs.pytest.org/en/latest/fixture.html) are provided as part of the PSyclone test suite. These are implemented in `<PSYCLONEHOME>/src/psyclone/tests/conftest.py` and are automatically discovered by pytest.

Those fixtures available for use when implementing tests are (in alphabetical order):

Fixture name	Description
annexed	Supplies a test with the various possible values of the LFRic *annexed_dofs* option.
dist_mem	Supplies a test with the various possible values of the *distributed-memory* option (only applicable to the LFRic and GOcean APIs currently). Also monkeypatches the global configuration object with the corresponding setting.
fortran_reader	Provides a Fortran PSyIR front-end object to convert Fortran code snippets into PSyIR.
fortran_writer	Provides a Fortran PSyIR back-end object to convert PSyIR trees into Fortran code.
have_graphviz	True if the Python bindings to the graphviz package (used when generating DAG visualisations) are available. Does *not* check that the underlying graphviz library is installed.
kernel_outputdir	Sets the output directory used by PSyclone for transformed kernels to be *tmpdir* (a built-in pytest fixture) and then returns *tmpdir*. Any test that directly or indirectly causes kernels to be transformed needs to use this fixture in order to avoid having unwanted files created within the git working tree.
parser	Creates an fparser2 parser for the Fortran2008 standard. This is an expensive operation so this fixture is only run once per test session.

In addition, there are two fixtures that are automatically run (just once) whenever a test session is begun. The first of these, `setup_psyclone_config`, ensures that the PSyclone configuration file used when running the test suite is the one distributed with PSyclone and not any locally-modified version. The second, `infra_compile`, sets-up the `tests.utilities.Compile` class with any compilation-testing flags (see *Compilation testing*) provided to the pytest command line. It also ensures that (if compilation testing is enabled) the LFRic-stub and GOcean infrastructure libraries are compiled prior to any tests running.

2.2 Coverage

The easiest and most user-friendly way of checking the coverage of any new code is to use CodeCov (https://codecov.io/gh/stfc/PSyclone) which is integrated with GitHub. Coverage for Pull Requests is automatically reported and will appear as a comment on the Pull Request. This comment is then automatically updated whenever new code is pushed to the associated branch.

For checking test coverage on your local machine you will need to install the `cov` plugin (`pip install pytest-cov`). You can then request various types of coverage report when running the test suite. e.g. to ask for a terminal report of missed lines for the `dynamo0p3` module you would do:

```
> cd <PSYCLONEHOME>
> pytest --cov-report term-missing --cov psyclone.dynamo0p3
```

Note that you specify the python module name, and not the file name. This will produce output along the lines of:

```
----------- coverage: platform linux, python 3.5.4-final-0 -----------
Name                         Stmts   Miss  Cover   Missing
----------------------------------------------------------
src/psyclone/dynamo0p3.py     2540     23    99%   558, 593, 777, 2731, 2972, 3865, 4132-
↪4133, 4135-4136, 4139-4140, 4143-4144, 4149-4151, 4255, 4270, 4488, 5026, 6540, 6658,␣
↪6768
```

showing the line numbers which are not covered. By using `--cov` more than once you can report on more than one file. You can also request only selected tests to be run by specifying the file names on the command line. Additionally html output can be created by adding the option `--cov-report html`:

```
> cd <PSYCLONEHOME>/src/psyclone/tests
> pytest --cov-report term-missing --cov-report html --cov psyclone.dynamo0p3 ./
↪dynamo0p3_basis_test.py ./parse_test.py
```

The html output can be viewed with a browser at `file:///.../tests/htmlcov/index.html` and it highlights all source lines in red that are not covered by at least one test.

2.3 Parallel execution

The size of the test suite is such that running all of it in serial can take many minutes, especially if you have requested a coverage report. It is therefore very helpful to run it in parallel and pytest provides support for this via the `xdist` plugin (`pip install pytest-xdist`). Once you have this plugin, the test suite may be run in parallel simply by providing the number of cores to use via the `-n` flag:

```
> cd <PSYCLONEHOME>
> pytest -n 4
```

Running the test suite in parallel also changes the order in which tests are run which can reveal any problems resulting from tests not being sufficiently isolated from one another.

2.4 Gotchas

The test utility pytest will only discover files that either start or end with "test". The PSyclone convention is to have all files ending with "_test.py", e.g. `constants_test.py`. A name using "tests" (plural) will not be automatically discovered or executed by pytest!

Note that pytest will not complain if two tests (within a module) have the same name - it will just silently ignore one of them! The best way of checking for this is to run pylint on any modified test modules. (This needs to be done anyway as one of the requirements of the *Code Review* is that all new code be pylint-clean.)

Note: You can use `pytest --collect-only` to check the names of the files and tests that would be executed, without actually executing the tests.

2.5 Documentation testing

Any code snippet included in the documentation should be tested to make sure our examples and documentation work as expected. Therefore, all examples in the documentation should be specified using `testcode` and `testoutput` directives, which allows these code snippets to be tested. For example:

```
  .. testcode::

  # access_info is an AccessInfo instance and contains one access. This
  # could be as simple as `a(i,j)`, but also something more complicated
  # like `a(i+2*j)%b%c(k, l)`.
  for indx in access_info.component_indices.iterate():
      # indx is a 2-tuple of (component_index, dimension_index)
      psyir_index = access_info.component_indices[indx]

  # Using enumerate:
  for count, indx in enumerate(access_info.component_indices.iterate()):
      psyir_index = access_info.component_indices[indx]
      # fortran writer converts a PSyIR node to Fortran:
      print("Index-id {0} of 'a(i,j)': {1}"
            .format(count, fortran_writer(psyir_index)))

.. testoutput::

    Index-id 0 of 'a(i,j)': i
    Index-id 1 of 'a(i,j)': j
```

Output should only be included if it is reasonably short. To avoid adding output to the manual, use the `:hide:` option of `testoutput`:

```
.. testoutput::
    :hide:

    Index 'i' is used.
```

The command *make doctest* will execute all tests marked in the documentation, and also any example code included in a docstring of a function or class that is documented in the manual (e.g. using `automethod`). Some tests or examples

will require data structure to be set up or modules to be imported. This can be done in a `testsetup` section. For example, here an excerpt from `dependency.rst`:

```
.. testsetup::

    from psyclone.psyir.frontend.fortran import FortranReader
    from psyclone.psyir.nodes import Loop

    code = '''subroutine sub()
    integer :: i, j, k, a(10, 10)
    a(i,j) = 1
    do i=1, 10
       j = 3
       a(i,i) = j + k
    enddo
    end subroutine sub
    '''
    psyir = FortranReader().psyir_from_source(code)
    # Take the loop node:
    loop = psyir.children[0][1]
    loop_statements = [loop]

Here might be then be several paragraphs of documentation.
Then in an example code, anything prepared in the above
code can be used, for example:

.. testcode::

    for statement in loop_statements:
        if isinstance(statement, Loop):
```

The `testsetup` section creates a variable `loop_statements` and imports the Loop class, and the actual example uses this code.

Many code snippets in python docstrings might try to parse a file, which typically cannot be found (unless the full path would be provided, which makes the example look ugly). One solution for this is to use a variable that is supposed to contain the filename, and then define this variable in the `testsetup` section. For example, the file `transformation.py` uses:

```
class ACCEnterDataTrans(Transformation):
    '''
    Adds an OpenACC "enter data" directive to a Schedule.
    For example:

    >>> from psyclone.parse.algorithm import parse
    >>> api = "gocean1.0"
    >>> ast, invokeInfo = parse(SOURCE_FILE, api=api)
    ...
    >>> dtrans.apply(schedule)
```

And the variable SOURCE_FILE is defined in the `testsetup` section of `transformations.rst`:

```
.. testsetup::

```

(continues on next page)

(continued from previous page)

```
    # Define SOURCE_FILE to point to an existing gocean 1.0 file.
    SOURCE_FILE = ("../../src/psyclone/tests/test_files/"
        "gocean1p0/test11_different_iterates_over_one_invoke.f90")

...

.. autoclass:: psyclone.transformations.ACCEnterDataTrans
   :noindex:
```

2.6 Compilation testing

The test suite provides support for testing that the code generated by PSyclone is valid Fortran. This is performed by writing the generated code to file and then invoking a Fortran compiler. This testing is not performed by default since it requires a Fortran compiler and significantly increases the time taken to run the test suite.

The Gnu Fortran compiler (gfortran) is used by default. If you wish to use a different compiler and/or supply specific flags then these are specified by further command-line flags:

```
> pytest --compile --f90=ifort --f90flags="-O3"
```

If you want to test OpenCL code created by PSyclone, you must use the command line option –compileopencl (which can be used together with –compile, and –f90 and –f90flags), e.g.:

```
> pytest --compileopencl --f90=<opencl-compiler> --f90flags="<opencl-specific flags>"
```

2.6.1 Infrastructure libraries

Since the code generated by PSyclone for the GOcean and LFRic domains makes calls to an infrastructure library, compilation tests must have access to compiler specific .mod files. For LFRic, a stub implementation of the required functions from the LFRic infrastructure is included in `tests/test_files/dynamo0p3/infrastructure`. When compilation tests are requested, the stub files are automatically compiled to create the required .mod files.

For the gocean1.0 domain a complete copy of the dl_esm_inf library is included as a submodule in `<PSYCLONEHOME>/external/dl_esm_inf`. Before running tests with compilation, make sure this submodule is up-to-date (see *Installation*). The test process will compile dl_esm_inf automatically, and all PSyclone gocean1.0 compilation tests will reference these files.

If you run the tests in parallel (see *Parallel execution* section) each process will compile its own version of the wrapper files and infrastructure library to avoid race conditions. This happens only once per process in each test session.

2.6.2 Other Dependencies

Occasionally the code that is to be compiled as part of a test may depend upon some piece of code that is not a Kernel or part of one of the supported infrastructure libraries. In order to support this, the `code_compiles` method of `psyclone.tests.utilities.Compile` allows the user to supply a list of additional files upon which kernels depend:

These files must be located in the same directory as the kernels.

CHAPTER

THREE

CONTINUOUS INTEGRATION

The PSyclone project uses GitHub Actions (https://psyclone.readthedocs.io/en/stable/examples.html#examples) for continuous integration. GitHub triggers an action whenever there is a push to a pull-request on the repository. The work performed by the action is configured in the `PSyclone/.github/workflows/python-package.yml` file.

Currently there are five main checks performed, in order of increasing computational cost (so that we 'fail fast'):

1. All links within all MarkDown files are checked. Those links to skip (because they are e.g. password protected) are specified in the `PSyclone/.github/workflows/mlc_config.json` configuration file.
2. All examples in the Developer Guide are checked for correctness by running `make doctest`.
3. The code base, examples and tutorials are lint'ed with flake8. (Configuration of flake8 is performed in `setup.cfg`.)
4. All links within the Sphinx documentation (rst files) are checked (see note below);
5. All of the examples are tested (for Python versions 3.6, 3.8 and 3.10.0) using the `Makefile` in the `examples` directory. No compilation is performed; only the `transform` (performs the PSyclone transformations) and `notebook` (runs the various Jupyter notebooks) targets are used. The `transform` target is run 2-way parallel (`-j 2`).
6. The full test suite is run for Python versions 3.6, 3.8 and 3.10.0 but without the compilation checks. `pytest` is passed the `-n auto` flag so that it will run the tests in parallel on as many cores as are available (currently 2 on GHA instances).

Since we try to be good 'open-source citizens' we do not do any compilation testing using GitHub as that would use a lot more compute time. Instead, it is the responsibility of the developer and code reviewer to run these checks locally (see *Compilation testing*).

By default, the GitHub Actions configuration uses `pip` to install the dependencies required by PSyclone before running the test suite. This works well when PSyclone only depends upon released versions of other packages. However, PSyclone relies heavily upon fparser which is also under development. Occasionally it may be that a given branch of PSyclone requires a version of fparser that is not yet released. As described in *Installation*, PSyclone has fparser as a git submodule. In order to configure GitHub Actions to use that version of fparser instead of a release, the `python-package.yml` file must be edited and the line executing `pip install external/fparser` must be uncommented.

Note that this functionality is only for development purposes. Any release of PSyclone must work with a released version of fparser and therefore the line described above must be commented out again before making a release.

A single run of the test suite on GitHub Actions uses approximately 20 minutes of CPU time and we run the test suite on three different versions of Python. Therefore, it is good practise to avoid triggering the tests unnecessarily (e.g. when we know that a certain commit won't pass). This may be achieved by including the "[skip ci]" tag (without the quotes) in the associated commit message.

3.1 Link checking

The link checking performed for the Sphinx documentation uses Sphinx's *linkcheck* functionality. Some URLs are excluded from this checking (due to ssl isues with an outdated http server or pages requiring authentication) and this is configured in the `conf.py` file of each document. Note also that anchors on GitHub actually have "user-content-" prepended but this is not shown in the links displayed by the browser (see https://github.com/sphinx-doc/sphinx/issues/6779). Therefore, any links to such anchors provided in the rst sources *must include* this "user-content-" text when specifying an anchor.

Since both the User and Developer Guides contain links to the Reference Guide, the issue of ensuring such links are correct is complex since a given PR may well alter the (auto-generated) Reference Guide but that version is, by definition, not yet available on Read The Docs (RTD). The solution to this is to perform the link checking against a *local* version of the Reference Guide rather than the one on RTD. For this to work, any links to the Reference Guide must be parameterised so that the correct URL can be generated, depending upon whether or not link checking is being performed. This parameterisation is achieved by implementing a Sphinx plugin which provides the *:ref_guide:* role. (The source for this plugin may be found in the `PSyclone/docs/_ext/apilinks.py` file.) The format to use when adding a link to the Reference Guide is then, e.g.:

```
:ref_guide:`anchor text psyclone.psyir.symbols.html#psyclone.psyir.symbols.UnknownType`
```

The URL to prepend to the supplied target is set via a new Sphinx configuration variable named `ref_guide_base` in the `conf.py` file. The final step is to set this appropriately, depending on whether or not the documentation is being built as part of a GitHub Actions run. The GHA configuration file `PSyclone/.github/workflows/python-package.yml` contains a step that sets the `GITHUB_PR_NUMBER` environment variable to the number of the current pull request. This is then queried within the `conf.py` file and, if set, the base URL is set to be that of a local webserver (started up as part of the GHA run). Otherwise, the base URL is set to be that of the latest version of the docs on RTD.

Since links between the User and Developer Guide use `intersphinx`, these may simply be configured using the `intersphinx_mapping` dictionary within `conf.py`.

CHAPTER

FOUR

PERFORMANCE

4.1 Exceptions

PSyclone exceptions are designed to provide useful information to the user. When there are problems transforming the PSyIR it can be useful to use one of the backends to provide the code causing problems in an easily readable form.

However, transformation exceptions can also be usefully used to only apply a transformation to valid parts of a tree. For example:

```
for node in nodes:
    try:
        transform(node)
    except TransformationError:
        pass
```

If a transformation is called many times in the way described above the exception string generated by the transformation error can cause PSyclone to run very slowly - particularly if the exception makes use of one of the backends.

The solution to this problem is to use the `LazyString` utility class (see `psyclone/errors.py`). This utility takes a function that returns a string and only executes the function if the `str` method is called for the class. This will not be the case for the above code as the exception string is not used.

This approach is currently used in the `CreateNemoKernelTrans` transformation and internally in the `TransformationError` exception (so that this transformation does not accidentally cause the string to be evaluated).

If a transformation is used in the way described above and PSyclone subsequently runs more slowly it is recommended that the `LazyString` class is used. It could be mandated that all transformation exceptions use this approach but so far this problem has only been found in one use case so it has been decided to modify the code as and when required.

CHAPTER

FIVE

CODE REVIEW

Before a branch can be merged to master it must pass code review. The guidelines for performing a review (i.e. what is expected from the developer) are available on the GitHub PSyclone wiki pages: https://github.com/stfc/PSyclone/wiki.

CHAPTER

SIX

THE PSYCLONE INTERNAL REPRESENTATION (PSYIR)

The PSyclone Intermediate Representation (PSyIR) is a language-independent Intermediate Representation that PSyclone uses to represent the PSy (Parallel System) and the Kernel (serial units of work) layers of an application that can be constructed from scratch or produced from existing code using one of the PSyIR front-ends. Its design is optimised to represent high-performance parallel applications in an expressive, mutable and extensible way:

- It is **expressive** because it is intended to be created and/or manipulated by HPC software experts directly when optimizing or porting the code.
- It is **mutable** because it is intended to be programmatically manipulated (usually though PSyclone scripts) and maintain a coherent state (with valid relationships, links to symbols, links to dependencies) after each manipulation.
- It is **extensible** because it is intended to be used as the core component of domain-specific systems which include additional abstract concepts or logic not captured in the generic representation.

To achieve these design goals we use a Normalised Heterogeneous AST representation together with a Type System and a Symbol Table. By **heterogeneous** we mean that we distinguish between AST nodes using Python class inheritance system and each node has its particular (and semantically relevant) navigation and behaviour methods. For instance the `Assignment` node has `lhs` and `rhs` properties to navigate to the left-hand-side and right-hand-side operators of the Assignment. It also means we can indentify a node using its type with `isinstance(node, Assignment)`. Nevertheless, we maintain a **normalised** core of node relationships and functionality that allows us to build tree walkers, tree visitors and dependency analysis tools without the need to consider the implementation details of each individual sub-class.

The common functionality that all nodes must have is defined in the PSyIR base class *Node*. See the list of all PSyIR common methods in the Node reference guide.

More information about the type system and symbols and how PSyIR can be transformed back to a particular language using the back-ends (Writers) is provided in the following sections of this guide.

6.1 How to create new PSyIR Nodes

In order to create a new PSyIR node, either for adding a new core PSyIR node or to extend the functionality in one of the PSyclone APIs, it is mandatory to perform the following steps:

1. The new node must inherit from `psyclone.psyir.nodes.Node` or one of its sub-classes. Note that `Node` won't be accepted as a child anywhere in the tree. It may be appropriate to specialise one of the existing subclasses of Node, rather than Node itself. A good starting point would be to consider `psyclone.psyir.nodes.Statement` (which will be accepted inside any Schedule) or `psyclone.psyir.nodes.DataNode` (which will be accepted anywhere that the node can be evaluated to a data element).
2. Set the `_text_name` and the `_color_key` class string attributes. These attributes will provide standard behaviour for the `__str__` and `view()` methods.

3. Set the `_children_valid_format` class string attribute and specialise the static method `_validate_child(position, child)`. These define, textually and logically, what types of nodes will be accepted as children of the new node:

   - `_children_valid_format` is the textual representation that will be used in error messages. It is expressed using tokens with the same name as the PSyIR classes and the following symbols:
     - `|`: or operand.
     - `,`: concatenation operand.
     - `[ expression ]*`: zero or more instances of the expression.
     - `[ expression ]+`: one or more instances of the expression.
     - <LeafNode>: NULL operand (no children accepted).

     For instance, an expression that accepts a statement as a first child and one or more DataNodes after it would be: `Statement [, DataNode]+`.
   - `_validate_child(position, child)` returns a boolean which indicates whether the given child is a valid component for the given position.

Note: Note that the valid children of a node are described two times, once in `_children_valid_format` and another in `_validate_child`, and it is up to the developer to keep them coherent in order to provide sensible error messages. Alternatively we could create an implementation where the textual representation is parsed and the validation method is generated automatically, hence avoiding the duplication. Issue #765 explores this alternative.

---

4. If any of the attributes introduced by this method should not be shallow-copied when creating a duplicate of this PSyIR branch, specialise the `_refine_copy` method to perform the appropriate copy actions.
5. If any of the attributes in this node should be used to compute the equality of two nodes, specialise the `__eq__` member to perform the appropriate checks. The default `__eq__` behaviour is to check both instance types are exactly the same, and each of their children also pass the equality check. The only restriction on this implementation is that it must call the `super().__eq__(other)` as part of its implementation, to ensure any inherited equality checks are correctly checked. The default behaviour ignores annotations and comment attributes, as they should not affect the semantics of the PSyIR tree.

For example, if we want to create a node that can be found anywhere where a statement is valid, and in turn it accepts one and only one DataNode as a child, we would write something like:

```
from psyclone.psyir.nodes import Statement, DataNode


class MyNode(Statement):
    ''' MyNode is an example node that can be found anywhere where statement
    is valid, and in turn it accepts one and only one DataNode as a children.
    '''
    _text_name = "MyNodeName"
    _colour = "blue"
    _children_valid_format = "DataNode"

    @staticmethod
    def _validate_child(position, child):
```

This implementation already provides the basic PSyIR functionality and the node can be integrated and used in the PSyIR tree:

```
>>> mynode = MyNode(children=[Literal("1", INTEGER_TYPE)])

>>> mynode.children.append(Literal("2", INEGER_TYPE))
...
psyclone.errors.GenerationError: Generation Error: Item 'Literal' can't be
child 1 of 'MyNodeName'. The valid format is: 'DataNode'.

>>> schedule.addchild(mynode)

>>> print(schedule.view())
Schedule[]
    MyNodeName[]
            Literal[value:'1', Scalar<INTEGER, UNDEFINED>]
```

For a full list of methods available in any PSyIR node see the Node reference guide.

Note: For convenience, the PSyIR children validation is performed with both: Node methods (e.g. `node.addchild()`) and also list methods (e.g. `node.children.extend([node1, node2])`).

To achieve this, we sub-classed the Python list and redefined all methods that modify the list by calling first the PSyIR provided validation method and subsequently, if valid, calling the associated list method.

6.2 The parent-child relationship

To facilitate the PSyIR tree navigation, the parent-child relationship between nodes is represented with a double reference (providing `node.parent` and `node.children` navigational properties).

However, to maintain the consistency of the double reference, we don't allow the node API to manually specify its `parent` reference. It is always the responsibility of a parent node to update the `parent` reference of its children. To make this possible for any operation applied to the `node.children` list, we provide this functionality in the same list subclass specialisation that does the child validation checks explained in the previous section. Therefore, all the following list operations will work as expected:

```
node.children.insert(node1)  # Will set node1.parent reference to node
node.children.extend([node2, node3])  # Will set node2 and node3 parent
                                      # references to node
del node.children[1]  # Will unset the parent reference of children[1]
node.children = []  # Will unset the parent references of all its previous
                    # children
node.detach()  # Will ask node.parent to free node, as node can't change
               # the connection by itself
```

The only exception to the previous consistency rule is when a node constructor is given the parent reference when building a PSyIR tree top-down. In this case, the single-direction reference will be accepted temporarily, but a child connection operation will need to be done eventually to satisfy the other part of the connection. Any attempt to insert the new node as a child of another node not specified in the constructor will fail as this would break the consistency with the predefined parent reference. For example:

```
assignment = Assignment()
rhs = Reference(symbol1, parent=assignment)  # Predefined parent reference
lhs = Reference(symbol2, parent=assignment)  # Predefined parent reference
```

(continues on next page)

(continued from previous page)

```
assignment.children = [lhs, rhs]  # Finalise parent-child relationship

node = Reference(symbol3, parent=assignment)
lhs.addchild(node)  # Will produce a Generation error because the node
                    # constructor specified that its parent would be the
                    # 'assignment' node
```

Note that a node which already has a parent won't be accepted as a child of another node, as this could break any previously existing parent-child relationship.

```
node1.children.insert(child)  # Valid
node2.children.insert(child)  # Will produce a GenerationError
```

Methods like `node.detach()`, `node.copy()` and `node.pop_all_children()` can be used to move or replicate existing children into different nodes.

6.3 Selected Node Descriptions

6.3.1 ScopingNode

A *ScopingNode* is an abstract class node that defines a scoping region, this node and all its descendants have access to a shared set of symbols. These symbols are described in the *SymbolTable* (*psyclone.psyir.symbols.SymbolTable*) attached to this node.

There is a double-link between the *ScopingNode* (through the `symbol_table` property) and the *SymbolTable* (through the `scope` property) objects. To maintain a consistent connection between both objects the only public methods to update the connections are the `attach` and `detach` methods of `SymbolTable` (which takes care of both sides of the connection).

Also note that the constructor will not accept as a parameter a symbol table that already belongs to another scope. The symbol table will need to be detached or deep copied before it can be assigned to the new ScopingNode.

See the full API in the ScopingNode reference guide.

Container

The *Container* node is a *ScopingNode* that contains one or more *Container* and/or *Routine* nodes. A *Container* can be used to capture a hierarchical grouping of *Routine* nodes and a hierarchy of *Symbol* scopes i.e. a *Symbol* specified in a *Container* is visible to all *Container* and *Routine* nodes within it and their descendants. See the full Container API in the Container reference guide.

FileContainer

The *FileContainer* node is a subclass of the *Container* node and is used to capture the concept of a file that contains one or more *Container* and/or *Routine* nodes. Whilst this structure is the same as for a *Container*, it is useful to distinguish between the two as backends may need to deal differently with a *FileContainer* and a *Container*.

A *FileContainer* is always created at the root of the PSyIR tree when parsing Fortran code, as a Fortran file can contain one or more program units (captured as *Containers* and/or *Routines*). PSyIR tree when parsing Fortran code, as Fortran code has the concept of a program (captured as a *FileContainer*) that can contain one or more program units (captured as *Containers* and/or *Routines*). See the full *FileContainer* API in the FileContainer reference guide.

Schedule

The *Schedule* is a *ScopingNode* that represents a sequence of statements. See the full *Schedule* API in the Schedule reference guide.

Routine

The *Routine* node is a subclass of *Schedule* that represents any program unit (subroutine, function or main program). As such it extends *Schedule* through the addition of the *return_symbol* (required when representing a function) and *is_program* properties. It also adds the *create* helper method for constructing a valid *Routine* instance. It is an important node in PSyclone because two of its specialisations: *InvokeSchedule* and *KernelSchedule* (described below), are used as the root nodes of PSy-layer invokes and kernel subroutines. This makes them the starting points for any walking of the PSyIR tree in PSyclone transformation scripts and a common target for the application of transformations.

InvokeSchedule

The *InvokeSchedule* is a PSyIR node that represents an invoke subroutine in the PSy-layer. It specialises the *psyclone.psyir.nodes.Routine* functionality with a reference to its associated *psyclone.psyGen.Invoke* object.

Note: This class will be renamed to *InvokeRoutine* in issue #909.

KernelSchedule

The *KernelSchedule* is a PSyIR node that represents a Kernel subroutine. As such it is a subclass of *psyclone.psyir.nodes.Routine* with *return_type* set to *None* and *is_program* set to *False*.

Note: This class will be renamed to *KernelRoutine* in issue #909.

6.3.2 Control-Flow Nodes

The PSyIR has three control flow nodes: *IfBlock*, *Loop* and *Call*. These nodes represent the canonical structure with which conditional branching constructs, iteration constructs and accessing other blocks of code are built. Additional language-specific syntax for branching and iteration will be normalised to use these same constructs. For example, Fortran has the additional branching constructs *ELSE IF* and *CASE*: when a Fortran code is translated into the PSyIR, PSyclone will build a semantically equivalent implementation using *IfBlocks*. Similarly, Fortran also has the *WHERE* construct and statement which are represented in the PSyIR with a combination of *Loop* and *IfBlock* nodes. Such nodes in the new tree structure are annotated with information to enable the original language-specific syntax to be recreated if required (see below). See the full IfBlock API in the IfBlock reference guide. The PSyIR also supports the concept of named arguments for *Call* nodes, see the *Named arguments* section for more details.

Note: A Call node (like the CodeBlock) inherits from both Statement and DataNode because it can be found in Schedules or inside Expressions, however this has some shortcomings, see issue #1437.

Control-Flow Node annotation

If the PSyIR is constructed from existing code (using e.g. the fparser2 frontend) then it is possible that information about that code may be lost. This is because the PSyIR is only semantically equivalent to certain code constructs. In order that information is not lost (making it possible to e.g. recover the original code structure if desired) Nodes may have *annotations* associated with them. The annotations, the Node types to which they may be applied and their meanings are summarised in the table below:

Annotation	Node types	Origin
was_elseif	*IfBlock*	*else if*
was_single_stmt	*IfBlock*, *Loop*	*if(logical-expr)expr* or Fortran *where(array-mask)array-expr*
was_case	*IfBlock*	Fortran *select case*
was_where	*Loop*, *IfBlock*	Fortran *where* construct

Note: A *Loop* may currently only be given the *was_single_stmt* annotation if it also has the *was_where* annotation. (Thus indicating that this *Loop* originated from a WHERE *statement* in the original Fortran code.) The PSyIR represents Fortran single-statement loops (often called array notation) as arrays with ranges in the appropriate indices.

Loop Node

The *Loop* node is the cannonical representation of a bounded loop, it has the start, stop, step and loop_body of the loop as its children. The node has the same semantics than the Fortran do construct: the boundary values are inclusive (both are part of the iteration space) and the start, stop and step expressions are evaluated just once at the beginning of the loop.

For more details on the *Loop* node, see the full API in the reference guide.

6.3.3 Ranges

The PSyIR has the *Range* node which represents a range of integer values with associated start, stop and step properties. e.g. the list of values [4, 6, 8, 10] would be represented by a *Range* with a start value of 4, a stop value of 10 and a step of 2 (all stored as *Literal* nodes). This class is intended to simplify the construction of Loop nodes as well as to support array slicing (see below). However, this functionality is under development and at this stage neither of those options have been implemented.

The *Range* node must also provide support for array-slicing constructs where a user may wish to represent either the entire range of possible index values for a given dimension of an array or a sub-set thereof. e.g. in the following Fortran:

```
real, dimension(10, 5) :: my_array
call some_routine(my_array(1, :))
```

the argument to *some_routine* is specified using array syntax where the lone colon means *every* element in that dimension. In the PSyIR, this argument would be represented by an *ArrayReference* node with the first entry in its *shape* being an integer *Literal* (with value 1) and the second entry being a *Range*. In this case the *Range* will have a start value of *LBOUND(my_array, 1)*, a stop value of *UBOUND(my_array, 1)* and a step of *Literal("1")*. Note that *LBOUND* and *UBOUND* will be instances of *BinaryOperation*. (For the particular code fragment given above, the values are in fact known [1 and 5, respectively] and could be obtained by querying the Symbol Table.)

See the full Range API in the Range reference guide.

6.3.4 Operation Nodes

Arithmetic operations and various intrinsic/query functions are represented in the PSyIR by sub-classes of the *Operation* node. The operations are classified according to the number of operands:

- Those having one operand are represented by *psyclone.psyir.nodes.UnaryOperation* nodes,
- those having two operands are represented by *psyclone.psyir.nodes.BinaryOperation* nodes.
- and those having more than two or a variable number of operands are represented by *psyclone.psyir.nodes.NaryOperation* nodes.

See the documentation for each Operation class in the Operation, UnaryOperation, BinaryOperation and NaryOperation sections of the reference guide.

Note that where an intrinsic (such as Fortran's *MAX*) can have a variable number of arguments, the class used to represent it in the PSyIR is determined by the actual number of arguments in a particular instance. e.g. *MAX(var1, var2)* would be represented by a *psyclone.psyir.nodes.BinaryOperation* but *MAX(var1, var2, var3)* would be represented by a *psyclone.psyir.nodes.NaryOperation*.

The PSyIR supports the concept of named arguments for operation nodes, see the *Named arguments* section for more details.

6.3.5 CodeBlock Node

The PSyIR CodeBlock node contains code that has no representation in the PSyIR. It is useful as it allows the PSyIR to represent complex code by using CodeBlocks to handle the parts which contain unsupported language features. One approach would be to work towards capturing all language features in the PSyIR, which would gradually remove the need for CodeBlocks. However, the purpose of the PSyIR is to capture code concepts that are relevant for performance, not all aspects of a code, therefore it is likely that CodeBlocks will continue to be an important part of the PSyIR. See the full Codeblock API in the CodeBlock reference guide.

The code represented by a CodeBlock is currently stored as a list of fparser2 nodes. Therefore, a CodeBlock's input and output language is limited to being Fortran. This means that only the fparser2 front-end and Fortran back-end can be used when there are CodeBlocks within a PSyIR tree. In theory, language interfaces could be written between CodeBlocks and other PSyIR Nodes to support different back-ends but this has not been implemented.

Currently PSyIR have a single CodeBlock node that can be found in place of full Statements or being part of an expression that evaluates to a DataNode. To make this possible CodeBlock is a subclass of both: Statement and DataNode. However, in certain situations we still need to differentiate which one it is, for instance the Fortran back-end needs this information, as expressions do not need indentation and a newline whereas statements do. For this reason, CodeBlock has a `structure` method that indicates whether the code contains one or more unrecognized language expressions or one or more statements (which may themselves contain expressions).

The Fortran front-end populates the `structure` attribute using a feature of the fparser2 node list that is if the first node in the list is a statement then so are all the other nodes in the list and that if the first node in the list is an expression then so are all the other nodes in the list. This allows the `structure` method to return a single value that represents all nodes in the list. The structure of the PSyIR hierarchy is used to determine whether the code in a CodeBlock contains expressions or statements. This is achieved by looking at the parent PSyIR Node. If the parent Node is a Schedule then the CodeBlock contains one or more statements, otherwise it contains one or more expressions.

This logic works for existing PSyIR nodes and relies on any future PSyIR nodes being constructed so this continues to be true. Another solution would be to have two different nodes: StatementsCodeBlock which subclasses Statement, and DataCodeBlock which subclasses DataNode. We have chosen the first implementation for the simplicity of having a single PSyIR node instead of two, but if things get more complicated using this implementation, the second alternative could be considered again.

6.3.6 ArrayMixin

`ArrayMixin` is an abstract "mix-in" base class which implements various methods that are specific to those nodes representing arrays and array accesses. It is subclassed by `ArrayReference`, `ArrayOfStructuresReference`, `ArrayMember` and `ArrayOfStructuresMember`.

6.3.7 Reference Node

The PSyIR `Reference` Node represents a variable access. It keeps a reference to a `Symbol` which will be stored in a symbol table. See the full `Reference` API in the Reference reference guide.

6.3.8 ArrayReference Node

The PSyIR `ArrayReference` Node represents an access to one or more elements of an array variable. It keeps a reference to a Symbol which will be stored in a symbol table. The indices used to access the array element(s) are represented by the children of the node. The `ArrayReference` Node inherits from both the `Reference` and `ArrayMixin` classes. See the full API in the ArrayReference reference guide.

6.3.9 Directive

The PSyIR `Directive` Node represents a Directive, such as is used in OpenMP or OpenACC. There are two subclasses, `RegionDirective` and `StandaloneDirective`. `RegionDirective` nodes contain a schedule as their first child, which contains the code segment covered by the directive, for example a `Loop` for which an OpenMP parallel do may be applied to. Both `RegionDirective` and `StandaloneDirective` may also have `Clause` nodes as children, and can be accessed through the `clauses` member. See the full API in the Directive reference guide.

6.3.10 Named arguments

The *Call* node and the three subclasses of the *Operation* node (*UnaryOperation*, *BinaryOperation* and *NaryOperation*) all support named arguments.

The argument names are provided by the *argument_names* property. This property returns a list of names. The first entry in the list refers to the first argument, the second entry in the list refers to the second argument, etc. An argument name is stored as a string. If an argument is not a named argument then the list entry will contain *None*. For example, for the following call:

```
call example(arg0, name1=arg1, name2=arg2)
```

the following list would be returned by the *argument_names* property:

```
[None, "name1", "name2"]
```

It was decided to implement it this way, rather than adding a new (*NamedArgument*) node, as 1) there is no increase in the number and types of PSyIR nodes and 2) iterating over all children (the arguments) of these nodes is kept simple.

The following methods support the setting and updating of named arguments: *create()*, *append_named_arg()*, *insert_named_arg()* and *replace_named_arg()*.

However, this implementation raises consistency problems as it is possible to insert, modify, move or delete children (argument) nodes directly. This would make the argument names list inconsistent as the names themselves are stored within the node.

To solve this problem, the argument names are stored internally in an *_argument_names* list which not only keeps the argument names but also keeps a reference (the *id*) to the associated child argument. An internal *_reconcile()* method then checks whether the internal *_argument_names* list and the actual arguments match and fixes any inconsistencies.

The *_reconcile()* method is called before the *argument_names* property returns its values, thereby ensuring that any access to *argument_names* is always consistent.

The *_reconcile()* method looks through the arguments and tries to match them with one of the stored id's. If there is no match it is assumed that this is not a named argument. This approach has the following behaviour: the argument names are kept if arguments are re-ordered; an argument that has replaced a named argument will not be a named argument; an inserted argument will not be a named argument, and the name of a deleted named argument will be removed.

Making a copy of the *Call* node or one of the three subclasses of Operation nodes (*UnaryOperation*, *BinaryOperation* or *NaryOperation*) also causes problems with consistency between the internal *_argument_names* list and the arguments. The reason for this is that the arguments get copied and therefore have a different *id*, whereas the *id`s in the internal `_argument_names* list are simply copied. To solve this problem, the *copy()* method is specialised to update the *id`s. A second issue is that the internal `_argument_names* list may already be inconsistent when a copy is made. Therefore the *_reconcile()* method is also called in the specialisation of the *copy()* method.

6.3.11 References to Structures and Structure Members

The PSyIR has support for representing references to symbols of structure type and to members of such structures. Since the former case is still a reference to a symbol held in a symbol table, it is already captured by the `Reference` node. A reference that includes an access to a member of a structure is described by a `StructureReference` which is a subclass of `Reference`. As such, it has a `symbol` property which gives the `Symbol` that the reference is to. The *member* of the structure being accessed is described by a *Member* (or subclass) which is stored as the first and only child of the `StructureReference`. The full API is given in the StructureReference section of the reference guide.

Similarly, `ArrayOfStructuresReference` represents a reference to a *member* of one or more elements of an array of structures. As such it subclasses both `ArrayMixin` and `StructureReference`. As with the latter, the first child describes the member being accessed and will be an instance of (a subclass of) `Member`. Subsequent children (of which there must be at least one since this is an array reference) then describe the array-index expressions of the reference in the usual fashion for an `ArrayReference`. The full API is given in the ArrayOfStructuresReference section of the reference guide.

Since *members* of structures are not represented by symbols in a symbol table, references to them are *not* subclasses of `Reference`. They are instead represented by instances of `Member` (or subclasses thereof). There are four of these:

Class	Type of Accessor Nested Inside
Member	No nested accessor (i.e. is a leaf)
ArrayMember	One or more elements of an array
StructureMember	Member of a structure
ArrayOfStructuresMember	Member of one or more elements of an array of structures

These classes are briefly described below. For full details please follow the appropriate links to the Reference Guide.

Member

This node is used for accesses to members of a structure which do not contain any further accesses nested inside. In a PSyIR tree, any instance of this node type must therefore have no children and a `StructureReference` or `StructureMember` (or subclasses thereof) as parent. The full API is given in the Member section of the reference guide.

ArrayMember

This node represents an access to one or more elements of an array within a structure. As such, it subclasses both `Member` and `ArrayMixin`. Its children follow the same rules as for an *ArrayReference Node*. The full API is given in the ArrayMember section of the reference guide.

StructureMember

This node represents an access to a member of a structure that is itself a member of a structure. As such, it has a single child which subclasses `Member` and specifies which component is being accessed. The full API is given in the StructureMember section of the reference guide.

ArrayOfStructuresMember

This node represents an access to a member of one or more elements of an array of structures that is itself a member of a structure. Its first child must be a subclass of `Member`. Subsequent children represent the index expressions for the array access. The full API is given in the ArrayOfStructuresMember section of the reference guide.

6.4 Comments attached to PSyIR Nodes

Since the PSyIR is designed to support source-to-source code generation, it is desirable to keep the output code as readable as possible, and this includes keeping or adding comments to the generated code. Comments are not first-class nodes in the PSyIR because it is an abstract syntax tree and it was preferable to hide the complexity of comment nodes from the PSyIR transformations and other manipulations. Therefore, comments have been implemented as string attributes (one for preceding and another for inline comments) attached to particular nodes. And thus the location of comments on a PSyIR tree will move together with their owning node.

The group of nodes that can contain comments does not have an exclusive common ancestor, so they have been implemented with a Mixin class called CommentableMixin. A node can keep track of comments if it inherits from this class, for example:

```
from psyclone.psyir.nodes.commentable_mixin import CommentableMixin

class MyNode(Node, CommentableMixin):
    ''' Example node '''

mynode = MyNode()
mynode.preceding_comment = "A preceding comment"
mynode.inline_comment = "An inline comment"
```

From the language-level PSyIR nodes, Container, Routine and Statement have the CommentableMixin trait.

6.5 Domain-Specific PSyIR

The discussion so far has been about generic language-level PSyIR. This is located in the `psyir` directory and contains nodes, symbols, transformations, front-ends and back-ends. None of this is domain specific.

To obtain domain-specific concepts the language-level PSyIR can be specialised or extended. All domains follow the PSyKAl separation of concerns with the Algorithm-layer and the PSy-layer having its own domain-specific concepts, this can be found in `psyclone.domain.common.algorithm` and `psyclone.domain.common.psylayer` respectively (some concepts are still on `psyclone.psyGen` for legacy reasons but will be moved to the new locations over time).

6.5.1 PSy-layer concepts

- The *PSyLoop* is a *Loop* where the boundaries are given by the domain specific iteration space that the kernels are applied to. In turn it is sub-classed in all of the domains supported by PSyclone. This then allows the class to be configured with a list of valid loop 'types'. For instance, the GOcean sub-class, *GOLoop*, has "inner" and "outer" while the LFRic (dynamo0.3) sub-class, *DynLoop*, has "dofs", "colours", "colour", "" and "null". The default loop type (iterating over cells) is here indicated by the empty string. The concept of a "null" loop type is currently required because the dependency analysis that determines the placement of halo exchanges is handled within the *Loop* class. As a result, every *Kernel* call must be associated with a *Loop* node. However, the LFRic domain has support for kernels which operate on the 'domain' and thus do not require a loop over cells or dofs in the generated PSy layer. Supporting a *DynLoop* of "null" type allows us to retain the dependence-analysis functionality within the *Loop* while not actually producing a loop in the generated code. When #1148 is tackled, the dependence-analysis functionality will be removed from the *Loop* class and this concept of a "null" loop can be dropped.
- The *Kern*, which can be of type *CodedKern*, *InlinedKern* or *BuiltIn* are the singular units of computation that can be found inside a *PSyLoop*.
- The *HaloExchange* is a distributed-memory concept in the PSy-layer.
- The *GlobalSum* is a distributed-memory concept in the PSy-layer.

6.5.2 Other specializations

In LFRic there are specialisations for kernel-layer datatypes and symbols. For the algorithm layer in both GOcean1.0 and LFRic there are specialisations for invokes and kernel calls. This is discussed further in the following sections.

6.6 The LFRic PSyIR

The LFRic PSyIR is a set of subclasses of the PSyIR which captures LFRic-specific routines, datatypes and associated symbols. These subclasses are work in progress and at the moment are limited to 1) a subset of the datatypes passed into LFRic kernels by argument and by use association and 2) LFRic calls (InvokeCall and KernCall) in the LFRic algorithm-layer. Over time these will be expanded to support a) all LFRic kernel datatypes, b) all LFRic PSyIR datatypes, c) subroutines (KernRoutine etc), d) derived quantities e.g. iterator variables and eventually e) higher level LFRic PSyIR concepts, which will not be concerned with symbol tables and datatypes.

The Kernel-layer subclasses will be used to:

1) check that the data types, dimensions, intent etc. of a coded kernel's subroutine arguments conform to the expected datatypes, dimensions, intent etc as defined by the kernel metadata and associated LFRic rules.

2) represent coded kernels, which will make it easier to reason about the structure of a kernel. At the moment a coded kernel is translated into generic PSyIR. This generic PSyIR will be further translated into LFRic PSyIR using the expected datatypes as specified by the kernel metadata and associated LFRic rules.

3) replace the existing kernel stub generation implementation so that the PSyIR back ends can be used and PSyclone will rely less on `f2pygen` and `fparser1`. At the moment `kernel_interface` provides the same functionality as `kern_stub_arg_list`, except that it uses the symbol table (which keeps datatypes and their declarations together).

4) generate the PSy-layer, replacing the existing `kern_call_arg_list` and `gen_call` routines.

The Algorithm-layer subclasses will be used to:

1) help with transforming the algorithm layer.

2) help with reasoning about the algorithm layer e.g. to check that the algorithm layer and kernel metadata match.

3) generate the LFRic Algorithm-layer PSyIR e.g. in psyclone-kern.

6.6.1 Algorithm-layer Classes

The LFRic PSyIR for the Algorithm layer is captured in `domain/lfric/algorithm/psyir.py`. Three classes are currently provided, one to capture an invoke call, `LFRicAlgorithmInvokeCall` and two to capture Builtin and (coded) Kernel calls within an invoke call, `LFRicBuiltinFunctor` and `LFRicKernelFunctor` respectively.

6.6.2 Kernel-layer Classes

The LFRic PSyIR for the Kernel layer is captured in `domain/lfric/psyir.py`. The relevant classes are generated to avoid boilerplate code and to make it simpler to change the LFRic infrastructure classes in the future.

The idea is to declare different classes for the different concepts. For example `NumberOfDofsDataType()` and `NumberOfDofsDataSymbol()` classes are created and these are subclasses of `DataType` and `DataSymbol` respectively. In `NumberOfDofsDataType` the `intrinsic` and `precision` properties are pre-defined, as is the fact that it is a scalar, so these do not need to be specified. All that is needed to create a `undf` symbol is a name and the function space it represents:

```
UNDF_W3 = NumberOfUniqueDofsDataSymbol("undf_w3", "w3")
```

For arrays, (e.g. for `FieldData`) the dimensions must also be provided:

```
UNDF_W3 = NumberOfUniqueDofsDataSymbol("undf_w3", "w3")
FIELD1 = RealFieldDataDataSymbol("field1", [UNDF_W3], "w3")
```

At the moment, argument types and values are also not checked e.g. the function space argument - see issue #926. There is also no consistency checking between specified function spaces (e.g. that `UNDF_W3` is for the same function space as `FIELD1` in the above example) - see issue #927. Also, the function space attribute would be better if it were a class, rather than using a string, see issue #934.

Currently entities which can have different intrinsic types (e.g. `FieldData`) are captured as different classes (`RealFieldDataDataSymbol`, `IntegerFieldDataDataSymbol` etc). This could be modified if a single class turns out to be preferable.

6.6.3 Kernel arguments

At the moment, kernel arguments are generated by the `KernStubArgList` or `KernCallArgList` classes. However, whilst these classes generate the correct number of arguments in the correct order, they have no knowledge of the datatypes that the arguments correspond to and how the arguments relate to each other (they just output strings).

The logic and declaration of kernel variables is handled separately by the `gen_stub` method in `DynKern` and the `gen_code` method in `DynInvoke`. In both cases these methods make use of the subclasses of `DynCollection` to declare variables.

When using the symbol table in the LFRic PSyIR we naturally capture arguments and datatypes together. The `KernelInterface` class is aiming to replicate the `KernStubArgList` class and makes use of the LFRic PSyIR. The idea is that the former will replace the latter when it has the same or more functionality. At the moment, only methods required to pass the tests have been implemented in `KernelInterface` so there is more to be done, but it is also not clear what the limitations are for `KernStubArgList`.

Eventually the definition of lfric datatypes should be moved to the LFRic PSyIR, but at the moment there is a lot of information defined in the `DynCollection` subclasses. This will need to be addressed over time.

6.7 The GOcean PSyIR

GOcean makes use of algorithm-layer PSyIR specialisations.

6.7.1 Algorithm-layer Classes

The GOcean PSyIR for the Algorithm layer is captured in `domain/common/algorithm/psyir.py`. Two classes are currently provided, one to capture an invoke call, `AlgorithmInvokeCall` and the other to capture (coded) Kernel calls within an invoke call, `KernelFunctor`.

CHAPTER

SEVEN

PSYIR TYPES AND SYMBOLS

7.1 DataTypes

PSyIR DataTypes currently support Scalar, Array, Structure and empty types via the `ScalarType`, `ArrayType`, `StructureType` and `NoType` classes, respectively. The `StructureType` simply contains an `OrderedDict` of named-tuples, each of which holds the name, type and visibility of a component of the type. These types are designed to be composable: one might have an `ArrayType` with elements that are of a `StructureType` or a `StructureType` that has components that are also of (some other) `StructureType`. `NoType` is the equivalent of C's `void` and is currently only used with `RoutineSymbols` when the corresponding routine has no return type (such as Fortran subroutines).

There are two other types that are used in situations where the full type information is not currently available: `UnknownType` means that the type-declaration is not supported by the PSyIR (or the PSyIR frontend) and `DeferredType` means that the type of a particular symbol has not yet been resolved. Since `UnknownType` captures the original, unsupported symbol declaration, it is subclassed for each language for which a PSyIR frontend exists. Currently therefore this is limited to `UnknownFortranType`.

It was decided to include datatype intrinsic as an attribute of ScalarType rather than subclassing. So, for example, a 4-byte real scalar is defined like this:

```
>>> scalar_type = ScalarType(ScalarType.Intrinsic.REAL, 4)
```

and has the following pre-defined shortcut

```
scalar_type = REAL4_TYPE
```

If we were to subclass, it would have looked something like this:

```
scalar_type = RealType(4)
```

where `RealType` subclasses `ScalarType`. It may be that the latter would have provided a better interface, but both approaches have the same functionality.

7.2 Symbols

At the moment, nodes that represent a scope (all *Schedules* and *Containers*) have a symbol table which contains the symbols used by their descendant nodes. Nested scopes with their associated symbol table are allowed in the PSyIR.

The *new_symbol* method is provided to create new symbols while avoiding name clashes:

`SymbolTable.`**`new_symbol`**`(`*root_name=None*, *tag=None*, *shadowing=False*, *symbol_type=None*, ***symbol_init_args*`)`

Create a new symbol. Optional root_name and shadowing arguments can be given to choose the name following the rules of next_available_name(). An optional tag can also be given. By default it creates a generic symbol but a symbol_type argument and any additional initialization keyword arguments of this symbol_type can be provided to refine the created Symbol.

Parameters

- **root_name** (*str or NoneType*) – optional name to use when creating a new symbol name. This will be appended with an integer if the name clashes with an existing symbol name.
- **tag** (*str*) – optional tag identifier for the new symbol.
- **shadowing** (*bool*) – optional logical flag indicating whether the name can be overlapping with a symbol in any of the ancestors symbol tables. Defaults to False.
- **symbol_type** (type object of class (or subclasses) of psyclone.psyir.symbols.Symbol) – class type of the new symbol.
- **symbol_init_args** (*unwrapped Dict[str] = object*) – arguments to create a new symbol.

Raises

TypeError – if the type_symbol argument is not the type of a Symbol object class or one of its subclasses.

However, if this symbol needs to be retrieved later on, one must keep track of the symbol or the returned name. As this is not always feasible when accessed from different routines, there is also the option to provide a tag to uniquely identify the symbol internally (the tag is not displayed in the generated code). Therefore, to create a new symbol and associate it with a tag, the following code can be used:

```
variable = node.symbol_table.new_symbol("variable_name",
                                        tag="variable_with_the_result_x"
                                        symbol_type=DataSymbol,
                                        datatype=DataType.INTEGER)
```

There are two ways to retrieve the symbol from a symbol table. Using the *name* or using the *tag* as lookup keys. This is done with the two following methods:

SymbolTable.**lookup**(*name*, *visibility=None*, *scope_limit=None*)

Look up a symbol in the symbol table. The lookup can be limited by visibility (e.g. just show public methods) or by scope_limit (e.g. just show symbols up to a certain scope).

Parameters

- **name** (*str*) – name of the symbol.
- **visibilty** – the visibility or list of visibilities that the symbol must have.
- **scope_limit** (psyclone.psyir.nodes.Node or *NoneType*) – optional Node which limits the symbol search space to the symbol tables of the nodes within the given scope. If it is None (the default), the whole scope (all symbol tables in ancestor nodes) is searched otherwise ancestors of the scope_limit node are not searched.

Returns

the symbol with the given name and, if specified, visibility.

Return type

psyclone.psyir.symbols.Symbol

Raises

- **TypeError** – if the name argument is not a string.
- **SymbolError** – if the name exists in the Symbol Table but does not have the specified visibility.
- **TypeError** – if the visibility argument has the wrong type.
- **KeyError** – if the given name is not in the Symbol Table.

SymbolTable.**lookup_with_tag**(*tag*, *scope_limit=None*)

Look up a symbol by its tag. The lookup can be limited by scope_limit (e.g. just show symbols up to a certain scope).

Parameters

- **tag** (*str*) – tag identifier.
- **scope_limit** – optional Node which limits the symbol search space to the symbol tables of the nodes within the given scope. If it is None (the default), the whole scope (all symbol tables in ancestor nodes) is searched otherwise ancestors of the scope_limit node are not searched.

Returns

symbol with the given tag.

Return type

psyclone.psyir.symbols.Symbol

Raises

- **TypeError** – if the tag argument is not a string.
- **KeyError** – if the given tag is not in the Symbol Table.

Sometimes, we have no way of knowing if a symbol we need has already been defined. In this case we can use a try/catch around the *lookup_with_tag* method and if a KeyError is raised (the tag was not found), then proceed to declare the symbol. As this situation occurs frequently the Symbol Table provides the *find_or_create_tag* helper method that encapsulates the described behaviour and declares symbols when needed.

SymbolTable.**find_or_create_tag**(*tag*, *root_name=None*, ***new_symbol_args*)

Lookup a tag, if it doesn't exist create a new symbol with the given tag. By default it creates a generic Symbol with the tag as the root of the symbol name. Optionally, a different root_name or any of the arguments available in the new_symbol() method can be given to refine the name and the type of the created Symbol.

Parameters

- **tag** (*str*) – tag identifier.
- **root_name** (*str*) – optional name of the new symbol if it needs to be created. Otherwise it is ignored.
- **new_symbol_args** (*unwrapped Dict[str, object]*) – arguments to create a new symbol.

Returns

symbol associated with the given tag.

Return type

psyclone.psyir.symbols.Symbol

Raises

SymbolError – if the symbol already exists but the type_symbol argument does not match the type of the symbol found.

By default the *get_symbol*, *new_symbol*, *add*, *lookup*, *lookup_with_tag*, and *find_or_create_tag* methods in a symbol table will also take into account the symbols in any ancestor symbol tables. Ancestor symbol tables are symbol tables attached to nodes that are ancestors of the node that the current symbol table is attached to. These are found in order with the *parent_symbol_table* method. This method provides a *scope_limit* argument to limit the extend of the upwards recursion provided to each method that uses it.

Sibling symbol tables are currently not checked. The argument for doing this is that a symbol in a sibling scope should not be visible in the current scope so can be ignored. However, it may turn out to make sense to check both in some circumstances. One result of this is that names and tags do not need to be unique in the symbol table hierarchy (just with their ancestor symbols). It makes sense for symbol names to not be unique in a hierarchy as names can be re-used within different scopes. However this may not be true for all names and it may even make sense to have a separate global symbol table in the future, as well as the existing nested ones. It is less clear whether tags should be unique or not.

All other methods act only on symbols in the local symbol table. In particular *__contains__*, *remove*, *get_unresolved_data_symbols*, *symbols*, *datasymbols*, *local_datasymbols*, *argument_datasymbols*, *imported_symbols*, *precision_datasymbols* and *containersymbols*. It is currently not clear whether this is the best solution and it is possible that these should reflect a global view. One issue is that the *__contains__* method has no mechanism to pass a *scope_limit* optional argument. This would probably require a separate *setter* and *getter* to specify whether to check ancestors or not.

7.3 Specialising Symbols

When code is translated into PSyIR there may be symbols with unknown types, perhaps due to symbols being declared in different files. For example, in the following declaration it is not possible to know the type of symbol *fred* without knowing the contents of the *my_module* module:

```
use my_module, only : fred
```

In such cases a generic *Symbol* is created and added to the symbol table.

Later on in the code translation it may be that *fred* is used as the name of a subroutine call:

```
call fred()
```

It is now known that *fred* is a *RoutineSymbol* so the original *Symbol* should be replaced by a *RoutineSymbol*.

A simple way to do this would be to remove the original symbol for *fred* from the symbol table and replace it with a new one that is a *RoutineSymbol*. However, the problem with this is that there may be separate references to this symbol from other parts of the PSyIR and these references would continue to reference the original symbol.

One solution would be to search through all places where references could occur and update them accordingly. Another would be to modify the current implementation so that either a) references went in both directions or b) references were replaced with names and lookups. Each of these solutions has their benefits and disadvantages.

A third solution would be to have a single, non-hierarchical Symbol class that has only a name and a symbol-type attribute. Then we could replace the symbol_type attribute when we discover more information without modifying the thinner Symbol class and therefore not affecting the references to it.

What is currently done is to specialise the symbol in place (so that any references to it do not need to change). This is implemented by the *specialise* method in the *Symbol* class. It takes a subclass of a *Symbol* as an argument and modifies the instance so that it becomes the subclass. For example:

```
>>> sym = Symbol("a")
>>> # sym is an instance of the Symbol class
```

(continues on next page)

(continued from previous page)

```
>>> sym.specialise(RoutineSymbol)
>>> # sym is now an instance of the RoutineSymbol class
```

Sometimes providing additional properties of the new sub-class is desirable, and sometimes even mandatory (e.g. a *DataSymbol* must always have a datatype and optionally a constant_value parameter). For this reason the specialise method implementation provides the same interface as the constructor of the symbol type in order to provide the same behaviour and default values as the constructor. For instance, in the *DataSymbol* case the following specialisations are possible:

```
>>> sym = Symbol("a")
>>> # The following statement would fail because it doesn't have a datatype
>>> # sym.specialise(DataSymbol)
>>> # The following statement is valid and constant_value is set to None
>>> sym.specialise(DataSymbol, datatype=INTEGER_TYPE)

>>> sym2 = Symbol("b")
>>> # The following statement would fail because the constant_value doesn't
>>> # match the datatype of the symbol
>>> # sym2.specialise(DataSymbol, datatype=INTEGER_TYPE, constant_value=3.14)
>>> # The following statement is valid and constant_value is set to 3
>>> sym2.specialise(DataSymbol, datatype=INTEGER_TYPE, constant_value=3)
```

CHAPTER

EIGHT

PSYIR BACK-ENDS

PSyIR back-ends translate PSyIR into another form (such as Fortran, C or OpenCL). Until recently this back-end support has been implemented within the PSyIR *Node* classes themselves via various *gen** methods. However, this approach is getting a little unwieldy.

Therefore PSyclone is transitioning into a *Visitor* pattern approach. Visitor backends are already being used in the back-end implementations that translate PSyIR kernel code. This approach separates the code to traverse a tree from the tree being visited. It is expected that the existing back-ends (used in the PSy-layer) will migrate to this new approach over time (more information about the PSy-layer migration can be found in *Back-ends for the PSy-layer*). The back-end visitor code is stored in *psyclone/psyir/backend*.

8.1 Visitor Base code

visitor.py in *psyclone/psyir/backend* provides a base class - *PSyIRVisitor* - that implements the visitor pattern and is designed to be subclassed by each back-end.

PSyIRVisitor is implemented in such a way that the PSyIR classes do not need to be modified. This is achieved by translating the class name of the object being visited in the PSyIR tree into the method name that the visitor attempts to call (using the Python *eval* function). *_node* is postfixed to the method name to avoid name clashes with Python keywords.

For example, an instance of the *Loop* PSyIR class would result in *PSyIRVisitor* attempting to call a *loop_node* method with the PSyIR instance as an argument. Note the names are always translated to lower case. Therefore, a particular back-end needs to subclass *PSyIRVisitor*, provide a *loop_node* method (in this particular example) and this method would then be called when the visitor finds an instance of *Loop*. For example:

```
from __future__ import print_function
from psyclone.psyir.visitor import PSyIRVisitor
class TestVisitor(PSyIRVisitor):
    ''' Example implementation of a back-end visitor. '''

    def loop_node(self, node):
        ''' This method is called if the visitor finds a loop. '''
        print("Found a loop node")

test_visitor = TestVisitor()
test_visitor._visit(psyir_tree)
```

It is up to the sub-class to call any children of the particular node. This approach was chosen as it allows the sub-class to control when and how to call children. For example:

```
from __future__ import print_function
from psyclone.psyir.visitor import PSyIRVisitor
class TestVisitor(PSyIRVisitor):
    ''' Example implementation of a back-end visitor. '''

    def loop_node(self, node):
        ''' This method is called if the visitor finds a loop. '''
        print("Found a loop node")
        for child in node.children:
            self._visit(child)

test_visitor = TestVisitor()
test_visitor._visit(psyir_tree)
```

If a *node* is called that does not have an associated method defined then *PSyIRVisitor* will raise a *VisitorError* exception. This behaviour can be changed by setting the *skip_nodes* option to *True* when initialising the visitor i.e.

```
test_visitor = TestVisitor(skip_nodes=True)
```

Any unsupported nodes will then be ignored and their children will be called in the order that they appear in the tree.

PSyIR nodes might not be direct subclasses of *Node*. For example, *GOKernelSchedule* subclasses *KernelSchedule* which subclasses *Routine* which subclasses *Schedule* which subclasses *Node*. This can cause a problem as a back-end would need to have a different method for each class e.g. both a *gokernelschedule_node* and a *kernelschedule_node* method, even if the required behaviour is the same. Even worse, expecting someone to have to implement a new method in all back-ends when they subclass a node (if they don't require the back-end output to change) is overly restrictive.

To get round the above problem, if the attempt to call a method with the name of the PSyIR class (with *_node* appended) fails, then the *PSyIRVisitor* will subsequently call the method name of its parent (with *_node* appended). This will continue with the *PSyIRVisitor* working its way through the class hierarchy in method resolution order until it is successful (or fails for all names and raises an exception).

This implementation gives the behaviour one would expect from standard inheritance rules. For example, if a *kernelschedule_node* method is implemented in the back-end and a *GOKernelSchedule* is found then a *gokernelschedule_node* method is first tried which fails, then a *kernelschedule_node* method is called which succeeds. Therefore all subclasses of *KernelSchedule* will call the *kernelschedule_node* method (if their particular specialisation has not been added).

One example of the power of this approach makes use of the fact that all PSyIR nodes have *Node* as a parent class. Therefore, some base functionality can be added there and all nodes that do not have a specific method implemented will call this. To see the class hierarchy, the following code can be written:

```
from __future__ import print_function
class PrintHierarchy(PSyIRVisitor):
    ''' Example of a visitor that prints the PSyIR node hierarchy. '''

    def node_node(self, node):
    ''' This method is called if no specific methods have been
        written. '''
        print("[ {0} start]".format(type(node).__name__))
        for child in node.children:
            self._visit(child)
        print("[ {0} end]".format(type(node).__name__))

print_hierarchy = PrintHierarchy()
```

(continues on next page)

(continued from previous page)

```
print_hierarchy._visit(psyir_tree)
```

In the examples presented up to now, the information from a back-end has been printed. However, a back-end will generally not want to use print statements. Output from a *PSyIRVisitor* is supported by allowing each method call to return a string. Reimplementing the previous example using strings would give the following:

```
from __future__ import print_function class
class PrintHierarchy(PSyIRVisitor):
    ''' Example of a visitor that prints the PSyIR node hierarchy'''

    def node_node(self, node):
        ''' This method is called if the visitor finds a loop '''
        result = "[ {0} start ]".format(type(node).__name__)
        for child in node.children:
            result += self._visit(child)
        result += "[ {0} end ]".format(type(node).__name__)
        return result

print_hierarchy = PrintHierarchy()
result = print_hierarchy._visit(psyir_tree)
print(result)
```

As most back-ends are expected to indent their output based in some way on the PSyIR node hierarchy, the *PSyIRVisitor* provides support for this. The *self._nindent* variable contains the current indentation as a string and the indentation can be increased by increasing the value of the *self._depth* variable. The initial depth defaults to 0 and the initial indentation defaults to two spaces. These defaults can be changed when creating the back-end instance. For example:

```
print_hierarchy = PrintHierarchy(initial_indent_depth=2,
                                 indent_string="***")
```

The *PrintHierarchy* example can be modified to support indenting by writing the following:

```
from __future__ import print_function
class PrintHierarchy(PSyIRVisitor):
    ''' Example of a visitor that prints the PSyIR node hierarchy
    with indentation'''

    def node_node(self, node):
        ''' This method is called if the visitor finds a loop '''
        result = "{0}[ {1} start ]\n".format(self._nindent,
                                             type(node).__name__)
    self._depth += 1
    for child in node.children:
        result += self._visit(child)
    self._depth -= 1
    result += "{0}[ {1} end ]\n".format(self._nindent,
                                        type(node).__name__)
    return result

print_hierarchy = PrintHierarchy()
result = print_hierarchy._visit(psyir_tree)
print(result)
```

As a visitor instance always calls the *_visit* method, an alternative (functor) implementation is provided via the *__call__* method in the base class. This allows the above example to be called in the following simplified way (as if it were a function):

```
print_hierarchy = PrintHierarchy()
result = print_hierarchy(psyir_tree)
print(result)
```

The primary reason for providing the above (functor) interface is to hide users from the use of the visitor pattern. This is the interface to expose to users (which is why *_visit* is used for the visitor method, rather than *visit*). An important characteristic of the *__call__* method is that it will manage the lowering of DSL-concepts because the backends should not provide specific visitors for concepts that do not relate directly to the language domain (more information about the lowering step is provided in the *Back-ends for the PSy-layer* section below). This step is done internally without exposing side effects (e.g. modifications to the provided tree). This is important because it permits the generation of backend code without altering the existing PSyIR tree, thus simplifying debugging and development. For instance the walk statement in the following example will return the same nodes, regardless of whether or not the print statement is commented out:

```
print_hierarchy = PrintHierarchy()
# print(print_hierarchy(psyir_tree))
psyir_tree.walk(APIHaloExchange)
```

Note: The property of not having side effects is implemented by making a copy of the whole tree provided as an argument to the visitor functor. An alternative that was explored was modifying the lowering implementation so that it returned a new sub-tree instead of modifying the current one in-place. This turned out to be complicated as the lowering method doesn't have a well defined region where the modification can happen (e.g. a DSL concept could need the addition of imports and new symbols defined in an ancestor symbol table).

8.2 PSyIR Validation

Although the validity of parent-child relationships is checked during the construction of a PSyIR tree (see e.g. *The parent-child relationship*), there are often constraints that can only be checked once the tree is complete i.e. at the point that a backend is used to generate code. One such example is that an OpenMP *do* directive must appear within an OpenMP *parallel* region.

The base PSyVisitor class provides support for this validation by calling the *validate_global_constraints()* method of each Node that it visits. The *Node* base class contains an empty implementation of this method. Therefore, if a subclass of *Node* is subject to certain global constraints then it must override this method and implement the required checks. If those checks fail then the method should raise a *GenerationError*.

Note that, if required, this validation may be disabled by passing *check_global_constraints=False* when constructing the PSyIRVisitor instance:

```
print_hierarchy = PrintHierarchy(check_global_constraints=False)
```

8.3 Available back-ends

Currently, there are two back-ends capable of generating Kernel code (a KernelSchedule with all its children), these are:

- *FortranWriter()* in *psyclone.psyir.backend.fortran*
- *OpenCLWriter()* in *psyclone.psyir.backend.opencl*

Additionally, there are two partially-implemented back-ends

- *psyclone.psyir.backend.c* which is currently limited to processing partial PSyIR expressions.
- *SIRWriter()* in *psyclone.psyir.backend.sir* which can generate valid SIR from simple Fortran code conforming to the NEMO API.

8.4 SIR back-end

The SIR back-end is limited in a number of ways:

- only Fortran code containing 3 dimensional directly addressed arrays, with simple stencil accesses, iterated with triply nested loops is supported. Imperfectly nested loops, doubly nested loops, etc will cause a `VisitorError` exception.
- anything other than real arrays (integer, logical etc.) will cause incorrect SIR code to be produced (see issue #468).
- calls are not supported (and will cause a VisitorError exception).
- loop bounds are not analysed so it is not possible to add in offset and loop ordering for the vertical. This also means that the ordering of loops (lat/lon/levels) is currently assumed.
- Fortran literals such as *0.0d0* are output directly in the generated code (but this could also be a frontend issue).
- the only unary operator currently supported is '-'.

The current implementation also outputs text rather than running Dawn directly. This text needs to be pasted into another script in order to run Dawn, see Example 4: Transforming Fortran code to the SIR the NEMO API example 4.

Currently there is no way to tell PSyclone to output SIR. Outputting SIR is achieved by writing a script which creates an SIRWriter and outputs the SIR (for kernels) from the PSyIR. Whilst the main 'psyclone' program could have a '-backend' option added it is not clear this would be useful here as it is expected that the SIR will be output only for certain parts of the PSyIR and (an)other back-end(s) used for the rest. It is not yet clear how best to do this - perhaps mark regions using a transformation.

It is unlikely that the SIR will be able to accept full NEMO code due to its complexities (hence the comment about using different back-ends in the previous paragraph). Therefore the approach that will be taken is to use PSyclone to transform NEMO to make regions that conform to the SIR constraints and to make these as large as possible. Once this is done then PSyclone will be used to generate and optimise the code that the SIR is not able to optimise and will let the SIR generate code for the bits that it is able to do. This approach seems a robust one but would require interface code between the Dawn generated cuda (or other) code and the PSyclone generated Fortran. In theory PSyclone could translate the remaining code to C but this would require no codeblocks in the PSyIR when parsing NEMO (which is a difficult thing to achieve), or interface code between codeblocks and the rest of the PSyIR.

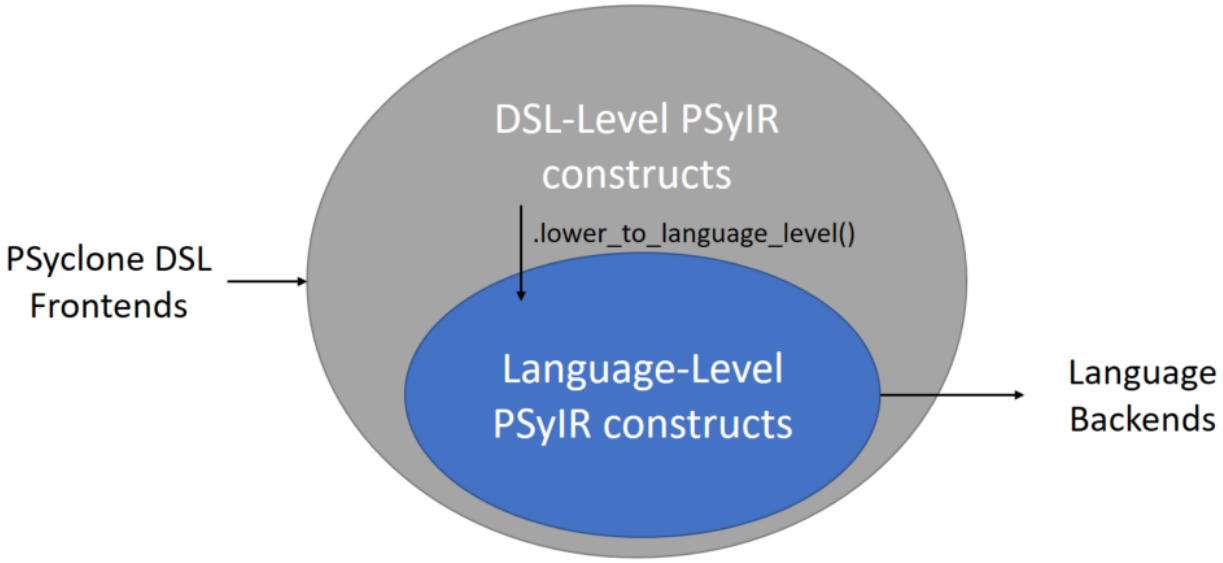
As suggested by the Dawn developers, PSyIR local scalar variables are translated into temporary SIR fields (which are 3D arrays by default). The reason for doing this is that it is easy to specify variables in the SIR this way (whereas I did not manage to get scalar declarations working) and Dawn optimises a temporary field, reducing it to its required dimensionality (so PSyIR local scalar variables are output as scalars by the Dawn back end even though they are specified as fields). A limitation of the current translation from PSyIR to SIR is that all PSyIR scalars are assumed to be local and all PSyIR arrays are assumed to be global, which may not be the case. This limitation is captured in issue #521.

8.5 Back-ends for the PSy-layer

The additional complexity of the PSy-layer comes from the fact that it contains multiple domain-specific concepts and parallel concepts that are not part of the target languages. Instead of dealing with these concepts in the visitors we require that any domain-specific concept introduced on top of the core PSyIR constructs contains the logic to lower this concept into language level constructs. The reasons for choosing a method instead of a visitor for this transformation are:

- Each concept introduced by the API-developer will need lowering instructions, and this is better implied by an abstract class in the node that needs to be filled.
- The lowering is done in-place. A method fits better with modifying the AST in-place because it can use and modify the nodes private fields.

The current proposed solution is to create a 2-phase generation workflow where a domain-specific PSyIR is first lowered to a language-level version of the PSyIR using the `lower_to_language_level` node method and then processed by the Visitor to generate the target language. The language-level PSyIR is still the same IR but restricted to the subset of Nodes that have a direct translation into target language concepts.



Note: Using the language backends to generate the PSy-layer code is supported by the Nemo and GOcean APIs. For the GOcean API the algorithm-layer is also generated using the language backends. LFRic support is still under development, see #1010.

CHAPTER

NINE

PARSING CODE

The original way to parse code was to use the PSyclone *parse* module which is responsible for parsing science (algorithm and kernel) code and extracting the required information for the algorithm translation and PSy generation phases. This is gradually being replaced by the use of the PSyIR and its front-ends and back-ends.

The current status is that the GOcean API uses PSyIR to capture and output algorithm code. This is achieved by first reading the algorithm file into generic PSyIR, then specialising the PSyIR (raising) to have GOcean-specific classes for invoke and kernel calls, then applying any transformations if required, then lowering the GOcean-specific classes back down to generic PSyIR (which also translates invokes and kernel calls to an appropriate call to the PSy-layer) and finally using the Fortran back-end to output the tranformed code. The same approach is in development for algorithm-layer code for the LFRic API, but currently the original approach is used.

In the original approach the *parse* module contains modules for parsing algorithm (*algorithm.py*) and kernel (*kernel.py*) code as well as a utility module (*utils.py*) for common functionality. This approach is discussed further in the following sections.

9.1 Parsing Algorithm Code

The first thing PSyclone typically does is parse an input file. This input file is expected to contain Fortran source code conforming to the particular API in question. For the *nemo* API this is standard code, for the other API's this is algorithm code (conforming to the PSyKAl separation of concerns). The PSyclone code to do this is found in *parse/algorithm.py*.

An input file can be parsed via the *parse* function or via an instance of the *Parser* class. In practice the *parse* function simply calls the *Parser* class so we will concentrate on the latter in this section. The *parse* function could be removed from PSyclone but it is simple and is used in existing PSyclone scripts and examples.

The *Parser* class is initialised with a number of optional arguments. A particular *api* can be specified (this is required so the parser knows what sort of code and metadata to expect, how to parse it and which *built-ins* are supported). The name used to specify an invoke (defaulting to *invoke*) can be changed, a path to where to look for associated kernel files can be provided and a particular maximum line length can be specified.

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

Supports the parsing of PSyclone conformant algorithm code within a file and extraction of relevant information for any 'invoke' calls contained within the code.

Parameters

- **api** (*str*) – the PSyclone API to use when parsing the code.
- **invoke_name** (*str*) – the expected name of the invocation calls in the algorithm code.
- **kernel_paths** (*list of str or NoneType*) – the paths to search for kernel source files (if different from the location of the algorithm source). Defaults to None.

- **line_length** (*bool*) – a logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms and an error raised if not. The default is False.

For example:

```
>>> from psyclone.parse.algorithm import Parser
>>> parser = Parser(api="gocean1.0")
>>> ast, info = parser.parse(SOURCE_FILE)
```

Once an instance of the *Parser()* class is created and configured with required values for the optional arguments, then the parse method can be called. This takes the name of the input code as its argument and returns a parse tree of the input code and a FileInfo object that captures the required invoke information found in the input code and in the associated kernel codes.

If the *nemo* API is specified then the *parse* method of the *Parser* instance simply parses the code with *fparser2* and returns the resultant *fparser2* parse tree.

For all other APIs the *parse* method of the *Parser* instance returns the resultant *fparser2* parse tree and a *FileInfo* instance which captures the invoke and kernel metadata in a hierarchy of classes.

When the *Parser* instance parses the code it expects to find Fortran code containing a program, module, subroutine or function (and it aborts if not). Currently the first of these (there may be more than one subroutine for example) is assumed to be the one that is required. This limititation is captured in issue #307.

The native *fparser2* tree of the Fortran code is then walked and all use statement names are captured and stored in a map (called *_arg_name_to_module_name*). This map allows the module name to be found given a particular argument name. Also, if an *invoke* call is found then an *InvokeCall* object is created and added to a list of such instances. Once all invokes have been found the *FileInfo* instance is created.

Note: In the future we want to be able to simply replace one parse tree with another. So how should we do this? One option would be to try to minimise the parser-specific parts and create some form of interface. The question is really what level of interface we should use.

An *InvokeCall* object is created in the *create_invoke_call* method by first parsing each of the kernels specified in an invoke call and creating a list of *kernel* objects which are then used to create an *InvokeCall*. These objects capture information on the way each kernel is being called from the Algorithm layer.

A *kernel* object is created in the *create_kernel_call* method which extracts the kernel name and kernel arguments, then creates either an *algorithm.BuiltInCall* instance (via the *create_builtin_kernel_call* method) or an *algorithm.KernelCall* instance (via the *create_coded_kernel_call* method). *BuiltInCalls* are created if the kernel name is the same as one of those specified in the built-in names for this particular API (see the variable *_builtin_name_map* which is initialised by the *get_builtin_defs* function).

The *create_kernel_call* method uses the *get_kernel* function to find out the kernel name and create a list of *Arg* instances representing the arguments. The *get_kernel* function parses each kernel argument using the fparser2 AST and determines the required argument information. An advantage of *fparser2* when compared with *fparser1* is that it parses all of a code, so we can use the parse tree to determine the type of each kernel argument appearing in the *invoke* call (e.g. scalar variable, array reference, literal constant) and create the appropriate *Arg* instance. Previously we relied on the *expression* module to do this (which has limitations).

Note: the analysis in the *get_kernel* function is the place to extend if we were to support arithmetic operations in an invoke call.

9.2 Mixed Precision

Support for mixed precision kernels has been added to PSyclone. The approach being taken is for the user to provide kernels with a generic interface and precision-specific implementations. As the PSyclone kernel metadata does not specify precision, this does not need to change. The actual precision being used will be specified by the scientist in the algorithm layer (by declaring variables with appropriate precision).

For example:

```
module kern_mod
  type kern_type
    ! metadata description
  contains
    procedure, nopass :: kern_code_32, kern_code_64
    generic :: kern_code => kern_code_32, kern_code_64
  end type kern_type
contains
  subroutine kern_code_32(arg)
    real*4 :: arg
    print *, "kern_code_32"
  end subroutine kern_code_32
  subroutine kern_code_64(arg)
    real*8 :: arg
    print *, "kern_code_64"
  end subroutine kern_code_64
end module kern_mod

program alg
  use kern_mod, only : kern_type
  real*4 :: a
  real*8 :: b
  call invoke(kern_type(a), kern_type(b))
end program alg
```

In the above example, the first call to kern will call *kern_code_32* and the second will call *kern_code_64*. Note, the actual code will use types for arguments in the algorithm layer, but for clarity precision has been used.

In order to support the above mixed precision kernel implementation, the PSy-layer generated by PSyclone must declare data passed into an invoke call with the same precision as is declared in algorithm layer, i.e. in the above example variable *a* must be declared as *real*4* and variable *b* as *real*8*. The only way for PSyclone to determine the required precision for a variable is by extracting the information from the algorithm layer.

To support the extraction of precision information, the name and precision of a variable are stored in a dictionary when the algorithm layer is parsed (see *psyclone.parser.algorithm.py*). This allows PSyclone to look up the precision of a variable when it is specified in an *invoke* call. The reason for implementing support in this way is because *fparser2* does not currently support a symbol table, therefore there is no link between variables names and their types. In the future, the algorithm layer will be translated into PSyIR, which does have a symbol table, so this dictionary will no longer be required (see issue #753).

All declarations that specify variables or types are stored in the dictionary, including those specified within locally defined types. Variables may also be single valued or arrays.

For example:

```
program alg
  use my_mod, only : my_type
  real(kind=r_def) :: rscalar
  type(my_type) :: field(10)
  type fred
    integer(kind=i_def) :: iscalar
  end type fred
end program alg
```

As the current implementation only stores variable names and does not know about variable scope there is a restriction that any variables within an algorithm code with the same name must have the same precision/type. This restriction will be removed when the algorithm layer is translated to PSyIR (see issue #753). The current implementation could be improved but in practice the lfric code does not fall foul of this restriction.

There is also a constraint that invoke arguments cannot be expressions (involving variables) or functions as it is then difficult to determine the datatype of the argument. However, arbitrary structures and arrays are supported, as are literal expressions.

For example the following are supported (where *b*, *f*, *g* and *i* are arrays, not functions):

```
program alg
  ! declare vars
  call invoke(kern(a, b(c), d%e, f(10)%g(4)%h, self%i(j), 1, 1.0*2.0))
end program alg
```

But the following are not:

```
program alg
  ! delare vars
  call invoke(kern(a%b(), c*d, e+1.0))
end program alg
```

The other issue is that, in general, it may not be possible to determine the type/precision of a variable from the algorithm layer code. In particular, the variable may be included from another module via use association. Potential solutions to this problem are 1) disallow this in the algorithm layer, 2) use a naming convention for the module and/or variable to determine its precision, or 3) search the modules for datatype information. At the moment only 1) or 2) will be feasible solutions. When we move to using the PSyIR (see issue #753), it may be possible to support 3).

9.3 Parsing Kernel Code (Metadata)

An *algorithm.BuiltInCall* instance is created by being passed a *kernel.BuiltinKernelType* instance for the particular API via the *BuiltInKernelTypeFactory* class which is found in the *parse.kernels* module. This class parses the Fortran module file which specifies built-in description metadata. Currently *fparser1* is used but we will be migrating to *fparser2* in the future. The built-in metadata is specified in the same form as coded kernel metadata so the same logic can be used (i.e. the *KernelTypeFactory.create* method is called) which is why *BuiltInKernelTypeFactory* subclasses *KernelTypeFactory*.

An *algorithm.KernelCall* instance is created by being passed the module name of the kernel and a *kernel.KernelType* instance for the particular API via the *KernelTypeFactory* class which is also found in the *parse.kernel* module. This class is given the parsed kernel module (via the *get_kernel_ast* function - which searches for the kernel file using the kernel path information introduced earlier). Again, currently *fparser1* is used but we will be migrating to *fparser2* in the future.

The *KernelTypeFactory create* method is used for both coded kernels and built-in kernels to specify the API-specific class to use. As an example, in the case of the *dynamo0.3* API, the class is *DynKernMetadata* which is found in *psyclone.dynamo0p3*. Once this instance has been created (by passing it an *fparser1* parse tree) it can return information about the metadata contained therein. Moving from *fparser1* to *fparser2* would required changing the parse code logic in each of the API-specific classes.

CHAPTER

TEN

GENERIC CODE

PSyclone is designed to be configurable so that new front-ends (called APIs) can be built, re-using as much existing code as possible. The generic code is kept in the `psyGen.py` file for psy-code generation.

CHAPTER

ELEVEN

NEW APIS

TBD

CHAPTER

TWELVE

EXISTING APIS

12.1 LFRic (Dynamo0.3)

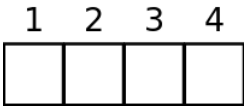
12.1.1 Mesh

The LFRic API supports meshes that are unstructured in the horizontal and structured in the vertical. This is often thought of as a horizontal 2D unstructured mesh which is extruded into the vertical. The LFRic infrastructure represents this mesh as a list of 2D cells with a scalar value capturing the number of levels in the vertical "column".

12.1.2 Cells

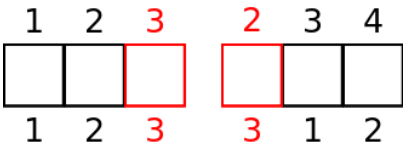
In the LFRic API, kernels which have metadata which specifies `operates_on = CELL_COLUMN` work internally on a column of cells. This means that PSyclone need only be concerned with iterating over cell-columns in the horizontal. (Kernels which have `operates_on = DOMAIN` work internally on all available columns of cells and therefore this iteration is not required.) As a result, the LFRic infrastructure presents the mesh information to PSyclone as if the mesh were 2-dimensional. From now on this 2D view will be assumed i.e. a cell will actually be a column of cells. The LFRic infrastracture provides a global 2D cell index from 1 to the number of cells.

For example, a simple quadrilateral element mesh with 4 cells might be indexed in the following way.



When the distributed memory option is switched on in the Dynamo0.3 API (see the Distributed Memory Section) the cells in the model are partitioned amongst processors and halo cells are added at the boundaries to a depth determined by the LFRic infrastructure. In this case the LFRic infrastructure maintains the global cell index and adds a unique local cell index from 1 to the number of cells in each partition, including any halo cells.

An example for a depth-1 halo implementation with the earlier mesh split into 2 partitions is given below, with the halo cells being coloured red. An example local indexing scheme is also provided below the cells. Notice the local indexing scheme is set up such that owned cells have lower indices than halo cells.



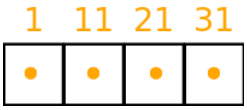
12.1.3 Dofs

In the LFRic infrastructure the degrees-of-freedom (dofs) are indexed from 1 to the total number of dofs. The infrastructure also indexes dofs so that the values in a column are contiguous and their values increase in the vertical. Thus, given the dof indices for the "bottom" cell, the rest of the dof indices can be determined for the column. This set of dof indices for the bottom cell is called a dofmap.

Dofs represent a field's values at various locations in the mesh. Fields can either be continuous or discontinuous. Continuous fields are so named because their values are continuous across cell boundaries. Dofs that represent continuous fields are shared between neighbouring cells. Discontinuous fields have values that are not necessarily related between neighbouring cells (there can be discontinuities across cell boundaries). Dofs that represent discontinuous fields are local to a cell.

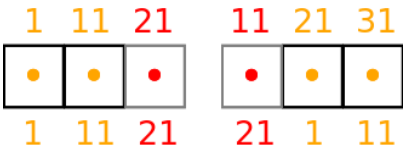
12.1.4 Discontinuous Dofs

A simple example of discontinuous dofs is given below. In this case each cell contains 1 dof and there are 10 cells in a column. We only show the bottom cells and their corresponding dof indices. As explained earlier, the dof indices increase contiguously up the column, so the cell above the cell containing dof index 1 contains dof index 2 and the cell above that contains dof index 3 etc.



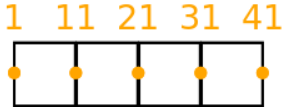
As discussed in the previous section, when the distributed memory option is switched on in the Dynamo0.3 API (see the Distributed Memory Section) the cells in the model are partitioned amongst processors and halo cells are added at the boundaries to a depth determined by the LFRic infrastructure. This results in the dofs being replicated in the halo cells, leading to a dof halo. As for cells, the LFRic infrastructure maintains the global dof indexing scheme and adds a local dof indexing scheme from 1 to the number of dofs in each partition, including any halo dofs.

An example for a depth-1 halo implementation with the earlier mesh split into 2 partitions is given below, with the halo cells drawn in grey and halo dofs coloured red. An example local partition indexing scheme is also provided below the dofs. As with cells, notice the local indexing scheme ensures that owned dofs have lower indices than halo dofs.



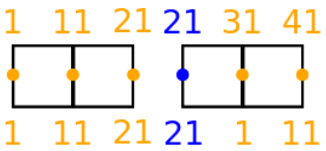
12.1.5 Continuous Dofs

A simple continuous dof example is given below for the same mesh as before. In this case dofs are on cell edges in the horizontal and there are 10 cells in a column. Again we only show the bottom cells and their corresponding dof indices. As explained earlier, the dof indices increase contiguously up the column, so the cell above the cell containing dof index 1 contains dof index 2 and the cell above that contains dof index 3 etc.

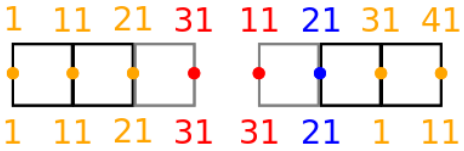


As already explained, when the distributed memory option is switched on in the Dynamo0.3 API (see the Distributed Memory Section) the cells in the model are partitioned amongst processors and halo cells are added at the boundaries.

In the example below we ignore the additional halo cells and just look at the partitioning of cells amongst processors (with the same mesh and 2 partitions as shown earlier). It can be seen that the dofs shared between cells which are on different partitions now need to be replicated if fields on continuous dofs are going to be able to be computed locally on each partition. This concept is different to halos as there are no halo cells here, the fact that the cells are partitioned has meant that continuous dofs on the edge of the partition are replicated. The convention used in Dynamo0.3 is that the cell with the lowest global id determines which partition owns a dof and which has the copy. Dofs which are copies are called `annexed`. Annexed dofs are coloured blue in the example:



If we now extend the above example to include the halo cells (coloured grey) then we get:



An example for a depth-1 halo implementation with the earlier mesh split into 2 partitions is given below, with the halo cells drawn in grey and halo dofs coloured red. An example local indexing scheme is also provided below the dofs. Notice the local indexing scheme ensures that owned dofs have lower indices than annexed dofs, which in turn have lower indices than halo dofs.

12.1.6 Cell and Dof Ordering

Cells in a partition are sequentially indexed by the LFRic infrastructure, starting at 1, so that local cells occur first, then level-1 halo cells, then level-2 halo cells etc. A benefit of this layout is that it makes it easy for PSyclone to specify the required iteration space for cells as a single range, allowing a single Fortran do loop (or other language construct as required) to be generated. The LFRic infrastructure provides an API that returns the index of the last owned cell, the index of the last halo cell at a particular depth and the index of the last halo cell, to support PSyclone code generation.

Dofs on a partition are also sequentially indexed by the LFRic infrastructure, starting at 1, so that local dofs occur first, then annexed dofs (if the field is continuous), then level-1 halo dofs, then level-2 halo dofs etc. Again, this layout makes it easy for PSyclone to specify the required iteration space for dofs as a single range. As before, the LFRic infrastructure provides an API that returns the index of the last owned dof, the index of the last annexed dof, the index of the last halo dof at a particular depth and the index of the last halo dof, to support PSyclone code generation.

12.1.7 Multi-grid

The Dynamo 0.3 API supports kernels that map fields between meshes of different horizontal resolutions; these are termed "inter-grid" kernels. As indicated in Fig. 12.1 below, the change in resolution between each level is always a factor of two in both the `x` and `y` dimensions.

Inter-grid kernels are only permitted to deal with fields on two, neighbouring levels of the mesh hierarchy. In the context of a single inter-grid kernel we term the coarser of these meshes the "coarse" mesh and the other the "fine" mesh.

There are two types of inter-grid operation; the first is "prolongation" where a field on a coarse mesh is mapped onto a fine mesh. The second is "restriction" where a field on a fine mesh is mapped onto a coarse mesh. Given the factor of two difference in resolution between the fine and coarse meshes, the depth of any halo accesses for the field on the fine mesh must automatically be double that of those on the coarse mesh.

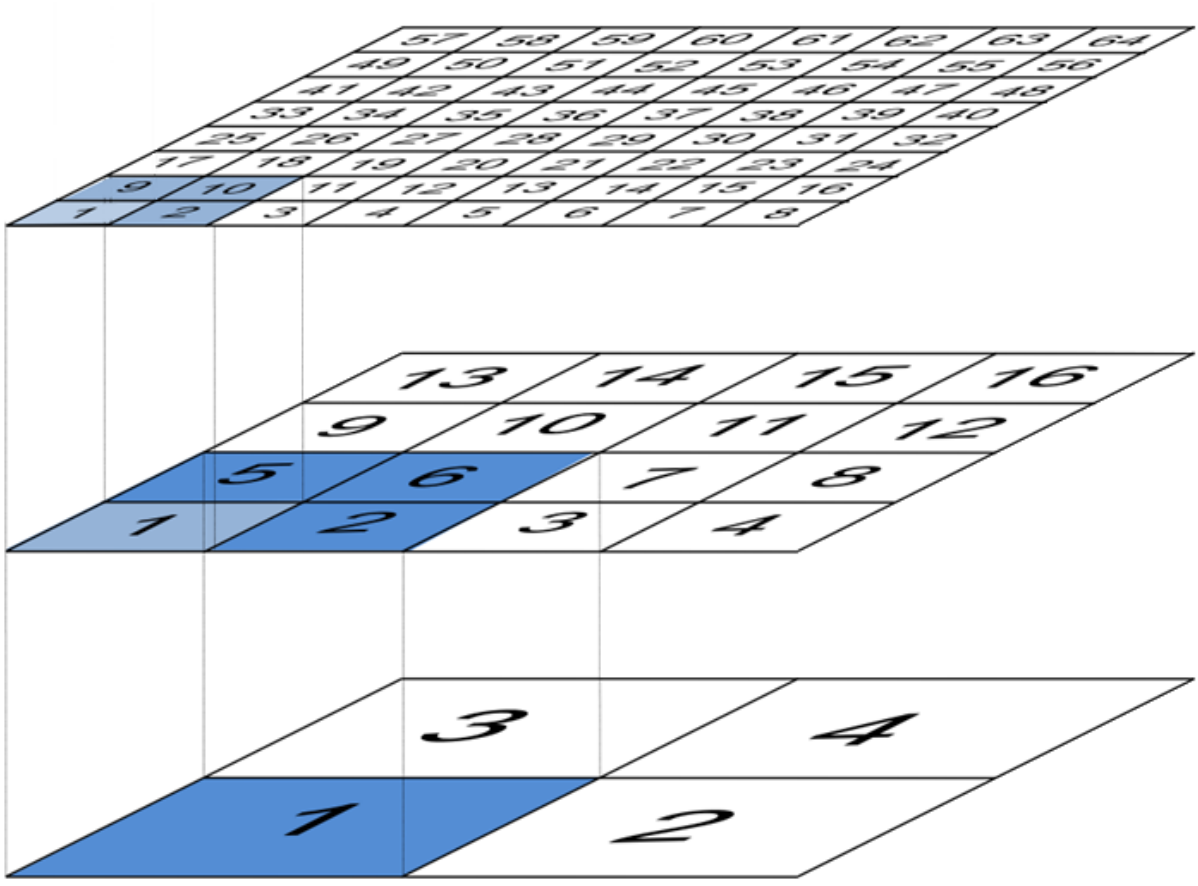


Fig. 12.1: The arrangement of cells in the multi-grid hierarchy used by LFRic. (Courtesy of R. Wong, Met Office.)

12.1.8 Loop iterators

In the current implementation of the Dynamo0.3 API it is possible to iterate (loop) either over cells or dofs. At the moment all coded kernels are written to iterate over cells and all Built-in kernels are written to iterate over dofs, but that does not have to be the case.

The loop iteration information is specified in the kernel metadata. In the case of Built-ins there is kernel metadata but it is part of PSyclone and is specified in `src/psyclone/parse/lfric_builtins_mod.f90`.

For inter-grid kernels, it is the coarse mesh that provides the iteration space. (The kernel is passed a list of the cells in the fine mesh that are associated with the current coarse cell.)

12.1.9 Cell iterators: Continuous

Note that if PSyclone does not know whether a modified field is discontinuous or continuous (because e.g. its function space is given as `ANY_SPACE_*` in kernel metadata) then it must assume it is continuous.

When a kernel is written to iterate over cells and modify a continuous field, PSyclone always (with the exception of `GH_WRITE` access - see below) computes dofs on owned cells and redundantly computes dofs in the level-1 halo (or to depth 2 if the field is on the fine mesh of an inter-grid kernel - see *Multi-grid*). Users can apply a redundant computation transformation to increase the halo depth for additional redundant computation but it must always at least compute the level-1 halo. The reason for this is to ensure that the shared dofs on cells on the edge of the partition (both owned and annexed) are always correctly computed. Note that the outermost halo dofs are not correctly computed and therefore the outermost halo of the modified field is dirty after redundant computation. Since shared dofs for a field with `GH_WRITE` access are guaranteed to have the same, correct value written to them, independent of whether or not the current cell "owns" them, there is no need to perform redundant computation in this case.

An alternative solution could have been adopted in Dynamo0.3 whereby no redundant computation is performed and partial-sum results are shared between processors in a communication pattern similar to halo exchanges. However, a decision was made to always perform redundant computation.

A downside of performing redundant computation in the level-1 halo is that any fields being read by the kernel must have their level-1 halo clean (up-to-date), which can result in halo exchanges.

This is also the case for a modified field with `GH_READINC` access as `readinc` captures a kernel field whose data is read (into the level-1 halo) and then incremented. However, the level-1 halo does not need to be clean for a modified field with `GH_INC` access, as an increment does not require the halo to be clean.

Whilst the level-1 halo does not need to be clean for a field with a `GH_INC` access, the data in the level-1 halo will be read and written. The data in the level-1 halo must therefore not cause any exceptions, which can be the case with some compilers where the values in the halo have not yet been written to (i.e. there will be an access to uninitialised data).

To avoid this problem the user guide currently recommends that all `setval_c` and `setval_x` Built-in calls (see Built-ins for more details) compute to the level-1 halo (by using the redundant computation transformation). This will guarantee that all modified halo data has been initialised with a value. If redundant computation transformations have been added then it is the outermost modified halo that will not require a halo exchange i.e. a loop iterating to the level-`n` halo will result in a halo exchange to the level-(`n-1`) halo being added before the loop, so the above Built-in calls would need to compute redundantly to the appropriate depth. In the future it may be that we should require fields with halos to have all of their data initialised to a set value when they are created, add an option to PSyclone, default to computing redundantly for the above Built-ins, or generate code that sets the halo to a specific value locally before the loop is called.

12.1.10 Cell iterators: Discontinuous

When a kernel is written to iterate over cells and modify a discontinuous field, PSyclone only needs to compute dofs on owned cells. Users can apply a redundant computation transformation (see the Transformations section) to redundantly compute into the halo but this is not done by default.

12.1.11 Dof iterators

When a kernel that is written to iterate over dofs modifies a field, PSyclone must ensure that all dofs in that field are updated. If the distributed memory flag is set to `false` then PSyclone must iterate over all dofs. PSyclone simply needs to create a loop that iterates from 1 to the total number of dofs. The latter value is provided by the LFRic API.

If the distributed memory flag is set to `true` then PSyclone must ensure that each partition only iterates over owned dofs. Again PSyclone just needs to create a loop that iterates from 1 to the total number of owned dofs on that partition. The latter value is provided by the LFRic API.

When the distributed memory flag is set to `true` an aditional configuration option can be set which makes PSyclone always create loops which iterate over both owned and annexed dofs. Whilst this is not necessary for correctness, it can improve performance by reducing the number of halo exchanges required (at the expense of computing annexed dofs redundantly). The only change for PSyclone is that it calls a different LFRic routine which returns the index of the last annexed dof. This iteration space will necessarily also include all owned dofs due to the ordering of dof indices discussed earlier.

The configuration variable is called `COMPUTE_ANNEXED_DOFS` and is found in the `dynamo0.3` section of the `psyclone.cfg` configuration file (see Configuration). If it is `true` then annexed dofs are always computed in loops that iterate over dofs and if it is `false` then annexed dofs are not computed. The default in PSyclone is `false`.

The computation of annexed dofs could have been added as a transformation optimisation. The reason for using a configuration switch is that it is then guaranteed that annexed dofs are always computed for loops that iterate over dofs which then allows us to always remove certain halo exchanges without needing to add any new ones.

If we first take the situation where annexed dofs are not computed for loops that iterate over dofs i.e. (`COMPUTE_ANNEXED_DOFS` is `false`), then a field's annexed dofs will be dirty (out-of-date) after the loop has completed. If a following kernel needs to read the field's annexed dofs, then PSyclone will need to add a halo exchange to make them clean.

There are five cases to consider:

1) the field is read in a loop that iterates over dofs,

2) the field is read in a loop that iterates over owned cells and level-1 halo cells,

3) the field is incremented in a loop that iterates over owned cells and level-1 halo cells,

4) the field is read in a loop that iterates over owned cells, and

5) the field is written in a loop that iterates over owned cells.

In case 1) the annexed dofs will not be read as the loop only iterates over owned dofs so a halo exchange is not required. In case 2) the full level-1 halo will be read (including annexed dofs) so a halo exchange is required. In case 3) the annexed dofs will be updated so a halo exchange is required. In case 4) the annexed dofs will be read so a halo exchange will be required. In case 5) the annexed dofs will be written with correct values (a condition of a kernel with `GH_WRITE` for a continuous field) so no halo exchange is required.

If we now take the case where annexed dofs are computed for loops that iterate over dofs (`COMPUTE_ANNEXED_DOFS` is `true`) then a field's annexed dofs will be clean after the loop has completed. If a following kernel needs to read the field's annexed dofs, then PSyclone will no longer need a halo exchange.

We can now guarantee that annexed dofs will always be clean after a continuous field has been modified by a kernel. This is because loops that iterate over either dofs or cells now compute annexed dofs and there are no other ways for a continuous field to be updated.

We now consider the same four cases. In case 1) the annexed dofs will now be read, but annexed dofs are guaranteed to be clean, so no halo exchange is required. In case 2) the full level-1 halo is read so a halo exchange is still required. Note, as part of this halo exchange we will update annexed dofs that are already clean. In case 3) the annexed dofs will be updated but a halo exchange is not required as the annexed dofs are guaranteed to be clean. In case 4) the annexed dofs will be read but a halo exchange is not required as the annexed dofs are guaranteed to be clean.

Furthermore, in the 3rd and 4th cases (in which annexed dofs are read or updated but the rest of the halo does not have to be clean), where the previous writer is unknown (as it comes from a different invoke call) we need to add a speculative halo exchange (one that makes use of the runtime clean and dirty flags) when `COMPUTE_ANNEXED_DOFS` is `False`, as the previous writer *may* have iterated over dofs, leaving the annexed dofs dirty. In contrast, when `COMPUTE_ANNEXED_DOFS` is `True`, we do not require a speculative halo exchange as we know that annexed dofs are always clean.

Therefore no additional halo exchanges are required when `COMPUTE_ANNEXED_DOFS` is changed from `false` to `true` i.e. case 1) does not require a halo exchange in either situation and case 2) requires a halo exchange in both situations. We also remove halo exchanges for cases 3) and 4) so the number of halo exchanges may be reduced.

If a switch were not used and it were possible to use a transformation to selectively perform computation over annexed dofs for loops that iterate over dofs, then we would no longer be able to guarantee that annexed dofs would always be clean. In this situation, if the dofs were known to be dirty then PSyclone would need to add a halo exchange and if it were unknown whether the dofs were dirty or not, then a halo exchange would need to be added that uses the run-time flags to determine whether a halo exchange is required. As run-time flags are based on whether the halo is dirty or not (not annexed dofs) then a halo exchange would be performed if the halo were dirty, even if the annexed dofs were clean, potentially resulting in more halo exchanges than are necessary.

12.1.12 Halo Exchange Logic

Halo exchanges are required when the `DISTRIBUTED_MEMORY` flag is set to `true` in order to make sure any accesses to a field's halo or to its annexed dofs receive the correct value.

Operators and Halo Exchanges

Halo Exchanges are only created for fields. This causes an issue for operators. If a loop iterates over halos to a given depth and the loop includes a kernel that reads from an operator then the operator must have valid values in the halos to that depth. In the current implementation of PSyclone all loops which write to, or update an operator are computed redundantly in the halo up to depth-1 (see the `load()` method in the `DynLoop` class). This implementation therefore requires a check that any loop which includes a kernel that reads from an operator is limited to iterating in the halo up to depth-1. PSyclone will raise an exception if an optimisation attempts to increase the iteration space beyond this (see the `gen_code()` method in the `DynKern` class).

To alleviate the above restriction one could add a configurable depth with which to compute operators e.g. operators are always computed up to depth-2, or perhaps up to the maximum halo depth. An alternative would be to halo exchange operators as required in the same way that halo exchanges are used for fields.

First Creation

When first run, PSyclone creates a separate InvokeSchedule for each of the invokes found in the algorithm layer. This schedule includes all required loops and kernel calls that need to be generated in the PSy layer for the particular invoke call. Once the loops and kernel calls have been created then (if the `DISTRIBUTED_MEMORY` flag is set to `true`) PSyclone adds any required halo exchanges and global sums. This work is all performed in the `DynInvoke` constructor (`__init__`) method.

In PSyclone we apply a lazy halo exchange approach (as opposed to an eager one), adding a halo exchange just before it is required.

It is simple to determine where halo exchanges should be added for the initial schedule. There are four cases:

1) loops that iterate over cells and modify a continuous field will access the level-1 halo. This means that any field that is read within such a loop must have its level-1 halo clean (up-to-date) and therefore requires a halo exchange. A modified field (specified as `GH_INC` which involves a read before a write) will require a halo exchange if its annexed dofs are not clean, or if their status is unknown. Whilst it is only the annexed dofs that need to be made clean in this case, the only way to acheive this is via a halo exchange (which updates the halo i.e. more than is required). Note, if the `COMPUTE_ANNEXED_DOFS` configuration variable is set to `true` then no halo exchange is required as annexed dofs will always be clean.

2) loops that iterate over cells and modify a continuous field but specify *GH_WRITE* access are a special case since the value to be written to any dof location is guaranteed to be independent of loop iteration (i.e. the current cell column). As such, annexed dofs are not required to be clean since they are not accessed.

3) continuous fields that are read from within loops that iterate over cells and modify a discontinuous field will access their annexed dofs. If the annexed dofs are known to be dirty (because the previous modification of the field is known to be from within a loop over dofs) or their status is unknown (because the previous modification to the field is outside of the current invoke) then a halo exchange will be required (As already mentioned, currently the only way to make annexed dofs clean is to perform a halo swap. Note, if the `COMPUTE_ANNEXED_DOFS` configuration variable is set to `true` then no halo exchange is required as annexed dofs will always be clean.

4) fields that have a stencil access will access the halo and need halo exchange calls added.

Halo exchanges are created separately (for fields with halo reads) for each loop by calling the `create_halo_exchanges()` method within the `DynLoop` class.

In the situation where a field's halo is read in more than one kernel in different loops, we do not want to add too many halo exchanges - one will be enough as long as it is placed correctly. To avoid this problem we add halo exchange calls for loops in the order in which they occur in the schedule. A halo exchange will be added before the first loop for a field but the same field in the second loop will find that there is a dependence on the previously inserted halo exchange so no additional halo exchange will be added.

The algorithm for adding the necessary halo exchanges is as follows: For each loop in the schedule, the `create_halo_exchanges()` method iterates over each field that reads from its halo (determined by the `unique_fields_with_halo_reads()` method in the `DynLoop` class).

For each field we then look for its previous dependencies (the previous writer(s) to that field) using PSyclone's dependence analysis. Three cases can occur: 1) there is no dependence, 2) there are multiple dependencies and 3) there is one dependence.

1) If no previous dependence is found then we add a halo exchange call before the loop (using the internal helper method `_add_halo_exchange()`). If the field is a vector field then a halo exchange is added for each component. The internal helper method `_add_halo_exchange` itself uses the internal helper method `_add_halo_exchange_code()`. This method creates an instance of the `DynHaloExchange` class for the field in question and adds it to the schedule before the loop. You might notice that this method then checks that the halo exchange is actually required and removes it again if not. In our current situation the halo exchange will always be needed so this check is not required but in more complex situations after transformations have been applied to the schedule this may not be the case. We discuss this type of situation later.

2) If multiple previous dependencies are found then the field must be a vector field as this is the only case where this can occur. We then choose the closest one and treat it as a single previous dependency (see 3).

3) If a single previous dependency is found and it is a halo exchange then we do nothing, as it is already covered. This will only happen when more than one reader depends on a writer, as discussed earlier. If the dependence is not a halo exchange then we add one.

After completing the above we have all the halo exchanges required for correct execution.

Note that we do not need to worry about halo depth or whether a halo is definitely required, or whether it might be required, as this is determined by the halo exchange itself at code generation time. The reason for deferring this information is that it can change as transformations are applied.

Halo Exchanges and Loop transformations

When a transformation (such as redundant computation) is applied to a loop containing a kernel, it can affect the surrounding halo exchanges. Consider the following example:

```
0: Loop[type='dofs', upper_bound='nannexed']
    0: BuiltIn setval_x(f2,f1)
1: HaloExchange[field='f1', check_dirty=True]
2: Loop[type='cells', upper_bound='cell_halo(1)']
    0: CodedKern testkern_code(f2,f1)
```

A potential halo exchange for field `f1` is required as the `testkern_code` kernel reads field `f1` in its level 1 halo.

If we transform the code so that the `setval_c` kernel computes redundantly to the level 1 halo:

```
0: HaloExchange[field='f1', check_dirty=True]
1: Loop[type='dofs', upper_bound='dof_halo(1)']
    0: BuiltIn setval_x(f2,f1)
2: Loop[type='cells', upper_bound='cell_halo(1)']
    0: CodedKern testkern_code(f2,f1)
```

then a potential halo exchange for field `f1` is now required before the `setval_c` kernel and the halo exchange before the `testkern_code` kernel is not required (as it is covered by the first halo exchange).

After such a transformation is applied then halo exchanges are managed by checking whether 1) any fields in the modified kernel now require a halo exchange before the kernel and adding them if so and 2) any existing halo exchanges after the loop, that were added due to fields being modified in the loop, are still required and removing them if not. Performing these 2 steps maintains halo exchange correctness and continues to minimise the number of required halo exchanges.

Note, the actual halo exchange extents (their depths) are computed dynamically so are not a concern at this point.

A general rule is that a halo exchange must not have a dependence with another halo exchange, as this would mean that one of them is not required. PSyclone should raise an exception if it finds this situation.

However, due to the two step halo exchange management process, there can be a transient situation after the first step where a halo exchange can temporarily depend on another halo exchange. If we revisit the previous example and consider the state of the system once the first step (checking whether halo exchanges are required *before* the modified kernel) has completed:

```
0: HaloExchange[field='f1', check_dirty=True]
1: Loop[type='dofs', upper_bound='dof_halo(1)']
    0: BuiltIn setval_x(f2,f1)
2: HaloExchange[field='f1', check_dirty=True]
```

(continues on next page)

(continued from previous page)

```
3: Loop[type='cells', upper_bound='cell_halo(1)']
    0: CodedKern testkern_code(f2,f1)
```

we have a transient situation where a halo exchange has been added before the `setval_x` kernel but the halo exchange before the `testkern_code` kernel has not yet been removed. Therefore, when adding, updating or removing halo exchanges the test for whether halo exchanges have a dependence between each other must be temporarily disabled. This is achieved by the `ignore_hex_dep` argument being set to `True` in the `_add_halo_exchange_code` function within the `DynLoop` class and the actual check that is skipped is implemented in the `_compute_halo_read_info` function within the `DynHaloExchange` class.

Asynchronous Halo Exchanges

The Dynamo0p3AsynchronousHaloExchange transformation allows the default synchronous halo exchange to be split into a halo exchange start and a halo exhange end which are represented separately as nodes in the schedule. These can then be moved in the schedule to allow overlapping of communication and computation, as long as data dependencies are honoured.

A halo exchange both reads and modifies a field so has a readwrite access for dependence analysis purposes. An asynchronous halo exchange start reads the field and an asynchronous halo exchange end writes to the field. Therefore the obvious thing to do would be to have the associated field set to read and write access respectively. However, the way the halo exchange logic works means that it is simplest to set the halo exchange end access to readwrite. The reason for this is that the logic to determine whether a halo exchange is required (`_required()`) needs information from all fields that read from the halo after the halo exchange has been called (and therefore must be treated as a write with following reads for dependence analysis) and it needs information from all fields that write to the field before the halo exchange has been called (and therefore must be treated as a read with previous writes for dependence analysis). An alternative would be to make the `_required()` method use the halo exchange start for previous writes and the halo exchange end for following reads. However, it was decided that this would be more complicated than the solution chosen.

Both halo exchange start and halo exchange end inherit from halo exchange. However, the halo exchange start and end are really two parts of the same thing and need to have consistent properties including after transformations have been performed. This is achieved by having the halo exchange start find and use the methods from the halo exchange end, rather than implement them independently. The actual methods needed are `_compute_stencil_type()`, `_compute_halo_depth()` and `_required()`. It is unclear how much halo exhange start really benefits from inheriting from halo exchange and this could probably be removed at the expense of returning appropriate names for the dag, colourmap, declaration etc.

Note: The dependence analysis for halo exchanges for field vectors is currently over zealous. It does not allow halo exchanges for independent vector components to be moved past one another. For example, a halo exchange for vector component 2, if placed after a halo exchange for component 1 could not be moved before the halo exchange for component 1, even though the accesses are independent of each other. This is also the case for asynchronous halo exchanges. See issue #220.

12.1.13 Evaluators

Evaluators consist of basis and/or differential basis functions for a given function space, evaluated at the nodes of another, 'target', function space. A kernel can request evaluators on multiple target spaces through the use of the `gh_evaluator_targets` metadata entry. Every evaluator used by that kernel will then be provided on all of the target spaces.

When constructing a `DynKernMetadata` object from the parsed kernel metadata, the list of target function-space names (as they appear in the meta-data) is stored in `DynKernMetadata._eval_targets`. This information is then used in the `DynKern._setup()` method which populates `DynKern._eval_targets`. This is an `OrderedDict` which has the (mangled) names of the target function spaces as keys and 2-tuples consisting of `FunctionSpace` and `DynKernelArgument` objects as values. The `DynKernelArgument` object provides the kernel argument from which to extract the function space and the `FunctionSpace` object holds full information on the target function space.

The `DynInvokeBasisFunctions` class is responsible for managing the evaluators required by all of the kernels called from an Invoke. `DynInvokeBasisFunctions._eval_targets` collects all of the unique target function spaces from the `DynKern._eval_targets` of each kernel.

`DynInvokeBasisFunctions._basis_fns` is a list holding information on each basis/differential basis function required by a kernel within the invoke. Each entry in this list is a `dict` with keys:

Key	Entry	Type
shape	Shape of the evaluator	*str*
type	Whether basis or differential basis	*str*
fspace	Function space	*FunctionSpace*
arg	Associated kernel argument	*DynKernelArgument*
qr_var	Quadrature argument name	*str*
nodal_fspaces	Target function spaces	list of *(FunctionSpace, DynKernelArgument)*

12.1.14 Precision

The different types (kinds) of precision for scalar, fields and operators are specified in the `lfric_constants.py` file. Adding a new precision (kind) name to PSyclone for the LFRic (Dynamo0.3) API should simply be a case of adding the appropriate entry to this file. Doing this will provide a working version, but, of course, it ignores any additional tests, an example and updating the documentation. The suggested approach here would be to search for an existing precision type e.g. `r_tran` and see where and how this is used in tests and examples and where it is mentioned in the documentation.

12.1.15 Modifying the Schedule

Transformations modify the schedule. At the moment only one of these transformations - the `Dynamo0p3RedundantComputationTrans` class in `transformations.py` - affects halo exchanges. This transformation can mean there is a requirement for new halo exchanges, it can mean existing halo exchanges are no longer required and it can mean that the properties of a halo exchange (e.g. depth) can change.

The redundant computation transformation is applied to a loop in a schedule. When this is done the `update_halo_exchanges()` method for that loop is called - see the `apply()` method in `Dynamo0p3RedundantComputationTrans`.

The first thing that the `update_halo_exchanges()` method does is call the `create_halo_exchanges()` method to add in any new halo exchanges that are required before this loop, due to any fields that now have a read access to their halo when they previously did not. For example, a loop containing a kernel that writes to a certain field might previously have iterated up to the number of owned cells in a partition (`ncells`) but now iterates up to halo depth 1.

However, a field that has its halo read no longer guarantees that a halo exchange is required, as the previous dependence may now compute redundantly to halo depth 2, for example. The solution employed in `create_halo_exchanges()` is to add a halo exchange speculatively and then remove it if it is not required. The halo exchange itself determines whether it is required or not via the `required()` method. The removal code is found at the end of the `_add_halo_exchange_code()` method in the `DynLoop()` class.

The second thing that the `update_halo_exchanges()` method does is check that any halo exchanges after this loop are still required. It finds all relevant halo exchanges, asks them if they are required and if they are not it removes them.

We only need to consider adding halo exchanges before the loop and removing halo exchanges after the loop. This is because redundant computation can only increase the depth of halo to which a loop computes so can not remove existing halo exchanges before a loop (as an increase in depth will only increase the depth of an existing halo exchange before the loop) or add existing halo exchanges after a loop (as an increase in depth will only make it more likely that a halo exchange is no longer required after the loop).

Kernel Transformations

Since PSyclone is invoked separately for each Algorithm file in an application, the naming of the new, transformed kernels is done with reference to the kernel output directory. All transformed kernels (and the modules that contain them) are re-named following the PSyclone Fortran naming conventions (Fortran Naming Conventions). This enables the reliable identification of transformed versions of any given kernel within the output directory.

If the "multiple" kernel-renaming scheme is in use, PSyclone simply appends an integer to the original kernel name, checks whether such a kernel is present in the output directory and if not, creates it. If a kernel with the generated name is present then the integer is incremented and the process repeated. If the "single" kernel-renaming scheme is in use, the same procedure is followed but if a matching kernel is already present in the output directory then the new kernel is not written (and we check that the contents of the existing kernel are the same as the one we would create).

If an application is being built in parallel then it is possible that different invocations of PSyclone will happen simultaneously and therefore we must take care to avoid race conditions when querying the filesystem. For this reason we use `os.open`:

```
fd = os.open(<filename>, os.O_CREAT | os.O_WRONLY | os.O_EXCL)
```

The `os.O_CREATE` and `os.O_EXCL` flags in combination mean that `open()` raises an error if the file in question already exists.

Colouring

If a loop contains one or more kernels that write to a field on a continuous function space then it cannot be safely executed in parallel on a shared-memory device. This is because fields on a continuous function space share dofs between neighbouring cells. One solution to this is to 'colour' the cells in a mesh so that all cells of a given colour may be safely updated in parallel (Fig. 12.2).

The loop over colours must then be performed sequentially but the loop over cells of a given colour may be done in parallel. A loop that requires colouring may be transformed using the `Dynamo0p3ColourTrans` transformation.

Each mesh in the multi-grid hierarchy is coloured separately (https://code.metoffice.gov.uk/trac/lfric/wiki/LFRicInfrastructure/MeshColouring) and therefore we cannot assume any relationship between the colour maps of meshes of differing resolution.

However, the iteration space for inter-grid kernels (that map a field from one mesh to another) is always determined by the coarser of the two meshes. Consequently, it is always the colouring of this mesh that must be used. Due to the set-up of the mesh hierarchy (see Fig. 12.1), this guarantees that there will not be any race conditions when updating shared quantities on either the fine or coarse mesh.

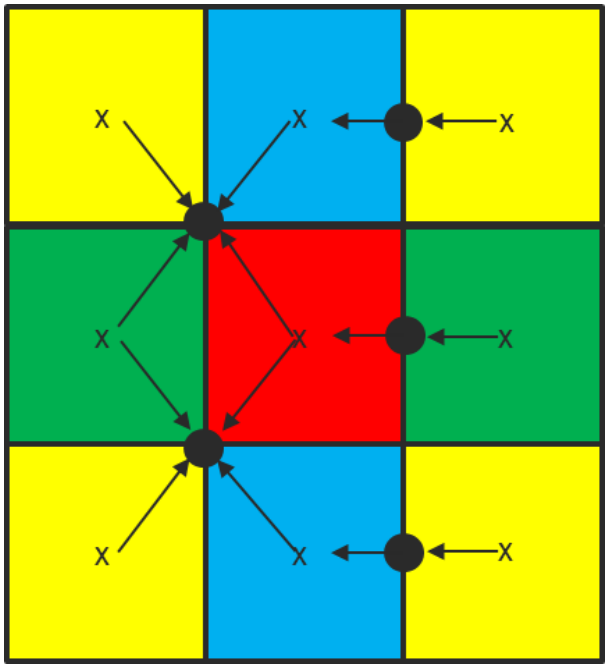


Fig. 12.2: Example of the colouring of the horizontal cells used to ensure the thread-safe update of shared dofs (black circles). (Courtesy of S. Mullerworth, Met Office.)

12.1.16 Lowering

As described in *Back-ends for the PSy-layer*, the use of a PSyIR backend to generate code for the LFRic PSy layer requires that each LFRic-specific node be lowered to 'language-level' PSyIR. Although this is work in progress (see e.g. https://github.com/stfc/PSyclone/issues/1010), some nodes already have the `lower_to_language_level()` method implemented. These are described in the sub-sections below.

BuiltIns

In the LFRic PSyIR, calls to BuiltIn kernels are represented by a single Node which is a subclass of LFRicBuiltIn. The `lower_to_language_level()` methods of these BuiltIn nodes must therefore replace that single Node with the PSyIR for the arithmetic operations required by the particular BuiltIn. This PSyIR forms the new body of the dof loop containing the original BuiltIn node.

In constructing this PSyIR, suitable Symbols for the loop variable and the various kernel arguments must be looked up. Since the migration to the use of language-level PSyIR for the LFRic PSy layer is at an early stage, in practise this often requires that suitable Symbols be constructed and inserted into the symbol table of the PSy layer routine. A lot of this work is currently performed in the `DynKernelArgument.infer_datatype()` method but ultimately (see https://github.com/stfc/PSyclone/issues/1258) much of this will be removed.

To date, all the LFRic BuiltIns have had `lower_to_language_level()` methods implemented except for the following:

- `LFRicXInnerproductYKern`,
- `LFRicXInnerproductXKern`,
- `LFRicSumXKern`,
- `LFRicIntXKern`,

- `LFRicRealXKern`.

The sum and inner product BuiltIns require extending PSyIR to handle reductions in the `GlobalSum` class in `psyGen.py`. Conversions from `real` to `int` and vice-versa require the target precisions be available as symbols, which is being implemented as a part of the mixed precision support.

12.2 GOcean1.0

TBD

12.3 NEMO

12.3.1 Usage

In general, the details of how PSyclone is used when building a particular model (such as LFRic) are left to the build system of that model. However, PSyclone support for the NEMO model is still evolving very rapidly and is not yet a part of the official NEMO repository. Consequently, the PSyclone repository contains two example scripts that are used when building the NEMO model. These scripts may be found in `examples/nemo/scripts` and their use is described in the `README.md` file in that directory.

12.3.2 PSyIR Construction

Since NEMO is an existing code, the way it is handled in PSyclone is slightly different from those APIs that mandate a PSyKAl separation of concerns (LFRic and GOcean1.0). As with the other APIs, fparser2 is used to parse the supplied Fortran source file and construct a parse tree (in `psyclone.generator.generate`). This parse tree is then passed to the `NemoPSy` constructor which uses the `fparser2` PSyIR frontend to construct the equivalent PSyIR. (This PSyIR is 'language-level' in that it does not contain any domain-specific constructs.) Finally, the PSyIR is passed to the `NemoInvokes` constructor which applies various 'raising' transformations which raise the level of abstraction by introducing domain-specific information.

12.3.3 Implicit Loops

When constructing the PSyIR of NEMO source code, PSyclone currently only considers explicit loops as candidates for being raised/transformed into `NemoLoop` instances. However, many of the loops in NEMO are written using Fortran array notation. Such use of array notation is encouraged in the NEMO Coding Conventions [nem13] and identifying these loops can be important when introducing, e.g. OpenMP. Currently these implicit loops are not automatically 'raised' into `NemoLoop` instances but can be done separately using the `NemoAllArrayRange2LoopTrans` transformation.

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

```
ascalar = afunc(twodarray1(:,:))
```

means that the function `afunc` is passed the (whole of the) `twodarray1` and returns a scalar value. (The requirement for explicit array shapes in the NEMO Coding Convention means that any quantity without such a shape must therefore be a scalar.)

Alternatively, a statement that assigns to an array must imply a loop:

```
twodarray2(:,:) = bfunc(twodarray1(:,:))
```

but it can only be converted into an explicit loop by PSyclone if the function `bfunc` returns a scalar. Since PSyclone does not currently attempt to fully resolve all symbols when parsing NEMO code, this information is not available and therefore such statements are not identified as loops (issue https://github.com/stfc/PSyclone/issues/286). This may then mean that opportunities for optimisation are missed.

CHAPTER

THIRTEEN

MODULES

This section describes the functionality of the various Python modules that make up PSyclone.

13.1 Module: f2pygen

f2pygen provides functionality for generating Fortran code from scratch and supports the addition of a use statement to an existing parse tree.

13.1.1 Variable Declarations

Three different classes are provided to support the creation of variable declarations (for intrinsic, character and derived-type variables). An example of their use might be:

```
>>> from psyclone.f2pygen import (ModuleGen, SubroutineGen, DeclGen,
... CharDeclGen, TypeDeclGen)
>>> module = ModuleGen(name="testmodule")
>>> sub = SubroutineGen(module, name="testsubroutine")
>>> module.add(sub)
>>> sub.add(DeclGen(sub, datatype="integer", entity_decls=["my_int"]))
>>> sub.add(CharDeclGen(sub, length="10", entity_decls=["my_char"]))
>>> sub.add(TypeDeclGen(sub, datatype="field_type", entity_decls=["ufld"]))
>>> gen = str(module.root)
>>> print(gen)
  MODULE testmodule
    IMPLICIT NONE
    CONTAINS
    SUBROUTINE testsubroutine()
      TYPE(field_type) ufld
      CHARACTER(LEN=10) my_char
      INTEGER my_int
    END SUBROUTINE testsubroutine
  END MODULE testmodule
```

The full interface to each of these classes is detailed below:

class psyclone.f2pygen.**DeclGen**(*parent*, *datatype=''*, *entity_decls=None*, *intent=''*, *pointer=False*, *kind=''*, *dimension=''*, *allocatable=False*, *save=False*, *target=False*, *initial_values=None*, *private=False*)

Generates a Fortran declaration for variables of various intrinsic types (integer, real and logical). For character variables CharDeclGen should be used.

Parameters

- **parent** (psyclone.f2pygen.BaseGen) – node to which to add this declaration as a child.
- **datatype** (*str*) – the (intrinsic) type for this declaration.
- **entity_decls** (*list*) – list of variable names to declare.
- **intent** (*str*) – the INTENT attribute of this declaration.
- **pointer** (*bool*) – whether or not this is a pointer declaration.
- **kind** (*str*) – the KIND attribute to use for this declaration.
- **dimension** (*str*) – the DIMENSION specifier (i.e. the xx in DIMENSION(xx)).
- **allocatable** (*bool*) – whether this declaration is for an ALLOCATABLE quantity.
- **save** (*bool*) – whether this declaration has the SAVE attribute.
- **target** (*bool*) – whether this declaration has the TARGET attribute.
- **initial_values** (*list of str with same no. of elements as entity_decls*) – initial value to give each variable.
- **private** (*bool*) – whether this declaration has the PRIVATE attribute (default is False).

Raises

RuntimeError – if datatype is not one of DeclGen.SUPPORTED_TYPES.

class psyclone.f2pygen.**CharDeclGen**(*parent*, *entity_decls=None*, *intent=''*, *pointer=False*, *kind=''*, *dimension=''*, *allocatable=False*, *save=False*, *target=False*, *length=''*, *initial_values=None*, *private=False*)

Generates a Fortran declaration for character variables.

Parameters

- **parent** (psyclone.f2pygen.BaseGen.) – node to which to add this declaration as a child.
- **entity_decls** (*list*) – list of variable names to declare.
- **intent** (*str*) – the INTENT attribute of this declaration.
- **pointer** (*bool*) – whether or not this is a pointer declaration.
- **kind** (*str*) – the KIND attribute to use for this declaration.
- **dimension** (*str*) – the DIMENSION specifier (i.e. the xx in DIMENSION(xx)).
- **allocatable** (*bool*) – whether this declaration is for an ALLOCATABLE quantity.
- **save** (*bool*) – whether this declaration has the SAVE attribute.
- **target** (*bool*) – whether this declaration has the TARGET attribute.
- **length** (*str*) – expression to use for the (len=xx) selector.
- **initial_values** (*list of str with same no. of elements as entity_decls*) – list of initial values, one for each variable. Each of these can be either a variable name or a literal, quoted string (e.g. "'hello'"). Default is None.
- **private** (*bool*) – whether this declaration has the PRIVATE attribute.

class psyclone.f2pygen.**TypeDeclGen**(*parent*, *datatype=''*, *entity_decls=None*, *intent=''*, *pointer=False*, *dimension=''*, *allocatable=False*, *save=False*, *target=False*, *is_class=False*, *private=False*)

Generates a Fortran declaration for variables of a derived type.

Parameters

- **parent** (psyclone.f2pygen.BaseGen) – node to which to add this declaration as a child.
- **datatype** (*str*) – the type for this declaration.
- **entity_decls** (*list*) – list of variable names to declare.
- **intent** (*str*) – the INTENT attribute of this declaration.
- **pointer** (*bool*) – whether or not this is a pointer declaration.
- **dimension** (*str*) – the DIMENSION specifier (i.e. the xx in DIMENSION(xx)).
- **allocatable** (*bool*) – whether this declaration is for an ALLOCATABLE quantity.
- **save** (*bool*) – whether this declaration has the SAVE attribute.
- **target** (*bool*) – whether this declaration has the TARGET attribute.
- **is_class** (*bool*) – whether this is a class rather than type declaration.
- **private** (*bool*) – whether or not this declaration has the PRIVATE attribute. (Defaults to False.)

13.1.2 Adding code

f2pygen supports the addition of use statements to an existing *fparser1* parse tree:

psyclone.f2pygen.**adduse**(*name*, *parent*, *only=False*, *funcnames=None*)

Adds a use statement with the specified name to the supplied object. This routine is required when modifying an existing AST (e.g. when modifying a kernel). The classes are used when creating an AST from scratch (for the PSy layer).

Parameters

- **name** (*str*) – name of module to USE
- **parent** (fparser.one.block_statements.*) – node in fparser1 AST to which to add this USE as a child
- **only** (*bool*) – whether this USE has an "ONLY" clause
- **funcnames** (*list*) – list of quantities to follow the "ONLY" clause

Returns
an fparser1 Use object

Return type
fparser.one.block_statements.Use

The PSyclone code where the *adduse* function was used has recently been migrated from using *fparser1* to using *fparser2*. In recognition of this change a new version of *adduse* has been developed which adds use statements to an existing *fparser2* parse tree. For the timebeing this new version is located in the same file it is used - *alg_gen.py* - but will be migrated to *f2pygen* (or equivalent) in the future:

psyclone.alg_gen.**adduse**(*location*, *name*, *only=None*, *funcnames=None*)

Add a Fortran 'use' statement to an existing fparser2 parse tree. This will be added at the first valid location before the current location.

This function should be part of the fparser2 replacement for f2pygen (which uses fparser1) but is kept here until this is developed, see issue #240.

Parameters

- **location** (fparser.two.utils.Base) – the current location (node) in the parse tree to which to add a USE.
- **name** (*str*) – the name of the use statement.
- **only** (*bool*) – whether to include the 'only' clause in the use statement or not. Defaults to None which will result in only being added if funcnames has content and not being added otherwise.
- **funcnames** (*list of str*) – a list of names to include in the use statement's only list. If the list is empty or None then nothing is added. Defaults to None.

Raises

- **GenerationError** – if no suitable enclosing program unit is found for the provided location.
- **NotImplementedError** – if the type of parent node is not supported.
- **InternalError** – if the parent node does not have the expected structure.

13.2 Module: configuration

PSyclone uses the Python ConfigParser class (https://docs.python.org/3/library/configparser.html) for reading the configuration file. This is managed by the psyclone.configuration module which provides a Config class. This class is a singleton, which can be (created and) accessed using Config.get(). Only one such instance will ever exist:

class psyclone.configuration.**Config**

Handles all configuration management. It is implemented as a singleton using a class _instance variable and a get() function.

property api

Getter for the API selected by the user.

Returns
The name of the selected API.

Return type
str

api_conf(*api=None*)

Getter for the object holding API-specific configuration options.

Parameters
api (*str*) – Optional, the API for which configuration details are required. If none is specified, returns the config for the default API.

Returns
object containing API-specific configuration

Return type
One of psyclone.configuration.LFRicConfig, psyclone.configuration.GOceanConfig or None.

Raises

- **ConfigurationError** – if api is not in the list of supported APIs.
- **ConfigurationError** – if the config file did not contain a section for the requested API.

property default_api

Getter for the default API used by PSyclone.

Returns
default PSyclone API

Return type
str

property default_stub_api

Getter for the default API used by the stub generator.

Returns
default API for the stub generator

Return type
str

property distributed_memory

Getter for whether or not distributed memory is enabled

Returns
True if DM is enabled, False otherwise

Return type
bool

property filename

Getter for the full path and name of the configuration file used to initialise this configuration object.

Returns
full path and name of configuration file

Return type
str

static find_file()

Static method that searches various locations for a configuration file. If the full path to an existing file has been provided in the PSYCLONE_CONFIG environment variable then that is returned. Otherwise, we search the following locations, in order:

- ${PWD}/.psyclone/
- **if inside-a-virtual-environment:**
  <base-dir-of-virtual-env>/share/psyclone/
- ${HOME}/.local/share/psyclone/
- <system-install-prefix>/share/psyclone/

Returns
the fully-qualified path to the configuration file

Return type
str

Raises
ConfigurationError – if no config file is found

static get(*do_not_load_file=False*)

Static function that if necessary creates and returns the singleton config instance.

Parameters
do_not_load_file (*bool*) – If set it will not load the default config file. This is used when handling the command line so that the user can specify the file to load.

get_constants()

Returns
the constants instance of the current API.

Return type
either `psyclone.domain.lfric.LFRicConstants`, `psyclone.domain.gocean.GOceanConstants`, or `psyclone.domain.nemo.NemoConstants`

get_default_keys()

Returns all keys from the default section. :returns list: List of all keys of the default section as strings.

static get_repository_config_file()

This function returns the absolute path to the config file included in the PSyclone repository. It is used by the testing framework to make sure all tests get the same config file (see tests/config_tests for the only exception). :return str: Absolute path to the config file included in the PSyclone repository.

property include_paths

Returns
the list of paths to search for Fortran include files.

Return type
list of str.

property kernel_naming

Returns
what naming scheme to use when writing transformed kernels to file.

Return type
str

property kernel_output_dir

Returns
the directory to which to write transformed kernels.

Return type
str

load(*config_file=None*)

Loads a configuration file.

Parameters
config_file (*str*) – Override default configuration file to read.

Raises
ConfigurationError – if there are errors or inconsistencies in the specified config file.

property ocl_devices_per_node

Returns
The number of OpenCL devices per node.

Return type
int

property psyir_root_name

Getter for the root name to use when creating PSyIR names.

Returns
the PSyIR root name.

Return type
str

property reprod_pad_size

Getter for the amount of padding to use for the array required for reproducible OpenMP reductions

Returns
padding size (no. of array elements)

Return type
int

property reproducible_reductions

Getter for whether reproducible reductions are enabled.

Returns
True if reproducible reductions are enabled, False otherwise.

Return type
bool

property supported_apis

Getter for the list of APIs supported by PSyclone.

Returns
list of supported APIs

Return type
list of str

property supported_stub_apis

Getter for the list of APIs supported by the stub generator.

Returns
list of supported APIs.

Return type
list of str

property valid_psy_data_prefixes

Returns
The list of all valid class prefixes.

Return type
list of str

The `Config` class is responsible for finding the configuration file (if no filename is passed to the constructor), parsing it and then storing the various configuration options. If PSyclone is started via `pytest`, the environment variable `PSYCLONE_CONFIG` is set to <`PSYCLONEHOME/config`>. This will guarantee that all tests use the config file provided in the PSyclone repository, and not a (potentially modified) user installed version.

The `Config` class also stores the list of supported APIs (`Config._supported_api_list`) and the default API to use if none is specified in either a config file or the command line (`Config._default_api`). Additionally, it performs some basic consistency checks on the values it obtains from the configuration file.

Since the PSyclone API to use can be read from the configuration file, it is not possible to have API-specifc sub-classes of `Config` as we don't know which API is in use before we read the file. However, the configuration file can contain API-specific settings. These are placed in separate sections, named for the API to which they apply, e.g.:

```
[dynamo0.3]
COMPUTE_ANNEXED_DOFS = false
```

Having parsed and stored the options from the default section of the configuration file, the `Config` constructor then creates a dictionary using the list of supported APIs to provide the keys. The configuration file is then checked for API-specific sections (again using the API names from the default section) and, if any are found, an API-specifc sub-class is created using the parsed entries from the corresponding section. The resulting object is stored in the dictionary under the appropriate key. The API-specific values may then be accessed as, e.g.:

```
Config.get().api_conf("dynamo0.3").compute_annexed_dofs
```

The API-specific sub-classes exist to provide validation/type-checking and encapsulation for API-specific options. They do not sub-class `Config` directly but store a reference back to the `Config` object to which they belong.

13.2.1 Constants Objects

Each API provides a specific object that stores required constants. Most of these constants are hard-coded in the object, but some are taken from a section of the configuration file. The constants are provided as class variables, but an instance of it needs to be created (at least once) in order to make sure all class variables are initialised. It is therefore recommended to always use an instance of the corresponding constant class to access these constants. The constant objects make sure that this initialisation only happens the very first time - creating an instance is therefore very cheap.

There three constant objects can be imported as follows:

- `from psyclone.domain.gocean import GOceanConstants`
- `from psyclone.domain.lfric import LFRicConstants`
- `from psyclone.domain.nemo import NemoConstants`

These objects can be used in two different ways:

1) If the API is known, e.g. because the constant is used in an API-specific file, an instance can simply be created and used, e.g.:

```
from psyclone.domain.lfric import LFRicConstants

const = LFRicConstants()

if var is in const.VALID_LOOP_BOUNDS_NAMES:
    ...
```

   This usage pattern can be seen in many API-specific files.

2) In some cases a value of an API-specific constant is required in a generic function. In this case the API-specific constant object can be accessed using the config file as follows:

```
from psyclone.configuration import Config

```

(continues on next page)

(continued from previous page)

```
const = Config.get().api_conf().get_constants()

if some_variable is in const.VALID_INTRINSIC_TYPES:
    ...
```

This pattern is used in some functions that can be called with different APIs. The following constants are used this way:

- VALID_ARG_TYPE_NAMES
- VALID_INTRINSIC_TYPES
- VALID_SCALAR_NAMES
- VALID_LOOP_TYPES

These are the only variables that are defined across all constant objects.

13.3 Module: transformations

As one might expect, the transformations module holds the various transformation classes that may be used to modify the Schedule of an Invoke and/or the kernels called from within it.

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

The base class for any transformation must be the class `Transformation`:

class psyclone.psyGen.**Transformation**(*writer=None*)

Abstract baseclass for a transformation. Uses the abc module so it can not be instantiated.

Parameters

writer (psyclone.psyir.backend.visitor.PSyIRVisitor) – optional argument to set the type of writer to provide to a transformation for use when constructing error messages. Defaults to FortranWriter().

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

Abstract method that applies the transformation. This function must be implemented by each transform. As a minimum each apply function must take a node to which the transform is applied, and a dictionary of additional options, which will also be passed on to the validate functions. This dictionary is used to provide optional parameters, and also to modify the behaviour of validation of transformations: for example, if the user knows that a transformation can correctly be applied in a specific case, but the more generic code validation would not allow this. Validation functions should check for a key in the options dictionary to disable certain tests. Those keys will be documented in each apply() and validate() function.

Note that some apply() functions might take a slightly different set of parameters.

Parameters

- **node** (*depends on actual transformation*) – The node (or list of nodes) for the transformation - specific to the actual transform used.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.

property name

Returns
the transformation's class name.

Return type
str

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

Method that validates that the input data is correct. It will raise exceptions if the input data is incorrect. This function needs to be implemented by each transformation.

The validate function can be called by the user independent of the apply() function, but it will automatically be executed as part of an apply() call.

As minimum each validate function must take a node to which the transform is applied and a dictionary of additional options. This dictionary is used to provide optional parameters and also to modify the behaviour of validation: for example, if the user knows that a transformation can correctly be applied in a specific case but the more generic code validation would not allow this. Validation functions should check for particular keys in the options dict in order to disable certain tests. Those keys will be documented in each apply() and validate() function as 'options["option-name"]'.

Note that some validate functions might take a slightly different set of parameters.

Parameters

- **node** (*depends on actual transformation*) – The node (or list of nodes) for the transformation - specific to the actual transform used.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.

Those transformations that work on a region of code (e.g. enclosing multiple kernel calls within an OpenMP region) must sub-class the `RegionTrans` class:

class psyclone.psyir.transformations.**RegionTrans**(*writer=None*)

This abstract class is a base class for all transformations that act on a list of nodes. It gives access to a validate function that makes sure that the nodes in the list are in the same order as in the original AST, no node is duplicated, and that all nodes have the same parent. We also check that all nodes to be enclosed are valid for this transformation - this requires that the sub-class populate the *excluded_node_types* tuple.

get_node_list(*nodes*)

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

Parameters
nodes (psyclone.psyir.nodes.Node or psyclone.psyir.nodes.Schedule or a list of :py:obj:`psyclone.psyir.nodes.Node) – can be a single node, a schedule or a list of nodes.

Returns
a list of nodes.

Return type
list of psyclone.psyir.nodes.Node

Raises

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

`validate`(*nodes*, *options=None*)

Checks that the nodes in node_list are valid for a region transformation.

Parameters

- **nodes** (subclass of `psyclone.psyir.nodes.Node` or a list of subclasses of `psyclone.psyir.nodes.Node`) – list of PSyIR nodes or a single node.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.
- **options["node-type-check"]** (*bool*) – this flag controls if the type of the nodes enclosed in the region should be tested to avoid using unsupported nodes inside a region.

Raises

- **TransformationError** – if the nodes in the list are not in the original order in which they are in the AST, a node is duplicated or the nodes have different parents.
- **TransformationError** – if any of the nodes to be enclosed in the region are of an unsupported type.
- **TransformationError** – if the parent of the supplied Nodes is not a Schedule or a Directive.
- **TransformationError** – if the nodes are in a NEMO Schedule and the transformation acts on the child of a single-line If or Where statment.
- **TransformationError** – if the supplied options are not a dictionary.

Finally, those transformations that act on a Kernel must sub-class the `KernelTrans` class:

In all cases, the *apply* method of any sub-class *must* ensure that the *validate* method of the parent class is called.

13.4 Module: psyGen

Provides the base classes for PSy-layer code generation.

13.5 Module: dynamo0p3

Specialises various classes from the `psyclone.psyGen` module in order to support the Dynamo 0.3 API.

When constructing the Fortran subroutine for either an Invoke or Kernel stub (see Kernel-stub Generator), there are various groups of related quantities for which variables must be declared and (for Invokes) initialised. Each of these groupings is managed by a distinct sub-class of the `DynCollection` abstract class:

(A single base class is used for both Invokes and Kernel stubs since it allows the code dealing with variable declarations to be shared.) A concrete sub-class of `DynCollection` must provide an implementation of the `_invoke_declarations` method. If the quantities associated with the collection require initialisation within the PSy layer then the `initialise` method must also be implemented. If stub-generation is to be supported for kernels that make use of the collection type then an implementation must also be provided for `_stub_declarations.`

Although instances of (sub-classes of) `DynCollection` handle all declarations and initialisation, there remains the problem of constructing the list of arguments for a kernel (or kernel stub). The `psyclone.dynamo0p3.ArgOrdering` base class provides support for this:

This class is then sub-classed in order to support the generation of argument lists when *calling* kernels (`KernCallArgList`) and when *creating* kernel stubs (`KernStubArgList`). `KernCallArgList` is only used in `DynKernelArguments.raw_arg_list()`. `KernStubArgList` is only used in `DynKern.gen_stub()`. These classes make use of `DynCollection` sub-classes in order to ensure that argument naming is consistent.

CHAPTER

FOURTEEN

DEPENDENCY ANALYSIS FUNCTIONALITY IN PSYCLONE

There are two different dependency analysis methods implemented, the old *dependency analysis* one, and a new one based on a *variable access API*. There is a certain overlap between these two methods, and it is expected that the current dependency analysis, which does not support the NEMO API, will be integrated with the variable access API in the future (see #1148).

14.1 Dependence Analysis

Dependence Analysis in PSyclone produces ordering constraints between instances of the *Argument* class within a PSyIR tree.

The *Argument* class is used to specify the data being passed into and out of instances of the *Kern* class, *HaloExchange* class and *GlobalSum* class (and their subclasses).

As an illustration consider the following invoke:

```
invoke(          &
    kernel1(a,b), &
    kernel2(b,c))
```

where the metadata for *kernel1* specifies that the 2nd argument is written to and the metadata for *kernel2* specifies that the 1st argument is read.

In this case the PSyclone dependence analysis will determine that there is a flow dependence between the second argument of *Kernel1* and the first argument of *Kernel2* (a read after a write).

Information about arguments is aggregated to the PSyIR node level (*kernel1* and *kernel2* in this case) and then on to the parent *loop* node resulting in a flow dependence (a read after a write) between a loop containing *kernel1* and a loop containing *kernel2*. This dependence is used to ensure that a transformation is not able to move one loop before or after the other in the PSyIR schedule (as this would cause incorrect results).

Dependence analysis is implemented in PSyclone to support functionality such as adding and removing halo exchanges, parallelisation and moving nodes in a PSyIR schedule. Dependencies between nodes in a PSyIR schedule can be viewed as a DAG using the *dag()* method within the *Node* base class.

14.1.1 DataAccess Class

The *DataAccess* class is at the core of PSyclone data dependence analysis. It takes an instance of the *Argument* class on initialisation and provides methods to compare this instance with other instances of the *Argument* class. The class is used to determine 2 main things, called *overlap* and *covered*.

Overlap

Overlap specifies whether accesses specified by two instances of the *Argument* class access the same data or not. If they do access the same data their accesses are deemed to *overlap*. The best way to explain the meaning of *overlap* is with an example:

Consider a one dimensional array called *A* of size 4 (*A(4)*). If one instance of the *Argument* class accessed the first two elements of array *A* and another instance of the *Argument* class accessed the last two elements of array *A* then they would both be accessing array *A* but their accesses would *not overlap*. However, if one instance of the *Argument* class accessed the first three elements of array *A* and another instance of the *Argument* class accessed the last two elements of array *A* then their accesses would *overlap* as they are both accessing element *A(3)*.

Having explained the idea of *overlap* in its general sense, in practice PSyclone currently assumes that *any* two instances of the *Argument* class that access data with the same name will always *overlap* and does no further analysis (apart from halo exchanges and vectors, which are discussed below). The reason for this is that nearly all accesses to data, associated with an instance of the *Argument* class, start at index 1 and end at the number of elements, dofs or some halo depth. The exceptions to this are halo exchanges, which only access the halo and boundary conditions, which only access a subset of the data. However these subset accesses are currently not captured in metadata so PSyclone must assume subset accesses do not exist.

If there is a field vector associated with an instance of an *Argument* class then all of the data in its vector indices are assumed to be accessed when the argument is part of a *Kern* or a *GlobalSum*. However, in contrast, a *HaloExchange* only acts on a single index of a field vector. Therefore there is one halo exchange per field vector index. For example:

```
InvokeSchedule[invoke='invoke_0_testkern_stencil_vector_type', dm=True]
... HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
... HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
... HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
... Loop[type='',field_space='w0',it_space='cells', upper_bound='cell_halo(1)']
... ... CodedKern testkern_stencil_vector_code(f1,f2) [module_inline=False]
```

In the above PSyIR schedule, the field *f1* is a vector field and the *CodedKern testkern_stencil_vector_code* is assumed to access data in all of the vector components. However, there is a separate *HaloExchange* for each component. This means that halo exchanges accessing the same field but different components do not *overlap*, but each halo exchange does overlap with the loop node. The current implementation of the *overlaps()* method deals with field vectors correctly.

Coverage

The concept of *coverage* naturally follows from the discussion in the previous section.

Again consider a one dimensional array called *A* of size 4 (*A(4)*). If one instance (that we will call the *source*) of the *Argument* class accessed the first 3 elements of array *A* (i.e. elements 1 to 3) and another instance of the *Argument* class accessed the first two elements of array *A* then their accesses would *overlap* as they are both accessing elements *A(1) and A(2)* and elements *A(1) and A(2)* would be *covered*. However, access *A(3)* for the *source Argument* class would not yet be *covered*. If a subsequent instance of the *Argument* class accessed the 2nd and 3rd elements of array *A* then all of the accesses (*A(1), A(2) and A(3)*) would now be *covered* so the *source argument* would be deemed to be covered.

In PSyclone the above situation occurs when a vector field is accessed in a kernel and also requires halo exchanges e.g.:

```
InvokeSchedule[invoke='invoke_0_testkern_stencil_vector_type', dm=True]
   HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
   HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
   HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
   Loop[type='',field_space='w0',it_space='cells', upper_bound='cell_halo(1)']
      CodedKern testkern_stencil_vector_code(f1,f2) [module_inline=False]
```

In this case the PSyIR loop node needs to know about all 3 halo exchanges before its access is fully *covered*. This functionality is implemented by passing instances of the *Argument* class to the *DataAccess* class *update_coverage()* method and testing the access.covered property until it returns *True*.

```
# this example is for a field vector 'f1' of size 3
# f1_index[1,2,3] are halo exchange accesses to vector indices [1,2,3] respectively
access = DataAccess(f1_loop)
access.update_coverage(f1_index1)
result = access.covered  # will be False
access.update_coverage(f1_index2)
result = access.covered  # will be False
access.update_coverage(f1_index3)
result = access.covered  # will be True
access.reset_coverage()
```

Note the *reset_coverage()* method can be used to reset internal state so the instance can be re-used (but this is not used by PSyclone at the moment).

The way in which halo exchanges are placed means that it is not possible for two halo exchange with the same index to depend on each other in a schedule. As a result an exception is raised if this situation is found.

Notice there is no concept of read or write dependencies here. Read or write dependencies are handled by classes that make use of the DataAccess class i.e. the *_field_write_arguments()* and *_field_read_arguments()* methods, both of which are found in the *Arguments* class.

14.2 Variable Accesses

Especially in the NEMO API, it is not possible to rely on pre-defined kernel information to determine dependencies between loops. So an additional, somewhat lower-level API has been implemented that can be used to determine variable accesses (READ, WRITE etc.), which is based on the PSyIR information. The only exception to this is if a kernel is called, in which case the metadata for the kernel declaration will be used to determine the variable accesses for the call statement. The information about all variable usage of a PSyIR node or a list of nodes can be gathered by creating an object of type *psyclone.core.access_info.VariablesAccessInfo*. This class uses a *Signature* object to keep track of the variables used.

14.2.1 Signature

A signature can be thought of as a tuple that consists of the variable name and structure members used in an access - called components. For example, an access like *a(1)%b(k)%c(i,j)* would be stored with a signature *(a, b, c)*, giving three components *a*, *b*, and *c*. A simple variable such as *a* is stored as a one-element tuple *(a,)*, having a single component.

class psyclone.core.access_info.**Signature**(*variable*, *sub_sig=None*)

Given a variable access of the form `a(i,j)%b(k,l)%c`, the signature of this access is the tuple `(a,b,c)`. For a simple scalar variable `a` the signature would just be `(a,)`. The signature is the key used in *VariablesAccessInfo*. In order to make sure two different signature objects containing the same variable can be used as a key, this

class implements *__hash__* and other special functions. The constructor also supports appending an existing signature to this new signature using the *sub_sig* argument. This is used in StructureReference to assemble the overall signature of a structure access.

Parameters

- **variable** (*str or tuple of str or list of str*) – the variable that is accessed.
- **sub_sig** (psyclone.core.Signature) – a signature that is to be added to this new signature.

__eq__(*other*)

Required in order to use a Signature instance as a key. Compares two objects (one of which might not be a Signature).

__hash__()

This returns a hash value that is independent of the instance. I.e. two instances with the same signature will have the same hash key.

__lt__(*other*)

Required to sort signatures. It just compares the tuples.

property is_structure

Returns
True if this signature represents a structure.

Return type
bool

to_language(*component_indices*, *language_writer=None*)

Converts this signature with the provided indices to a string in the selected language.

TODO 1320 This subroutine can be removed when we stop supporting strings - then we can use a PSyIR writer for the ReferenceNode to provide the right string.

Parameters

- **component_indices** (*psyclone.core.component_indices.ComponentIndices*) – the indices for each component of the signature.
- **language_writer** (None (default is Fortran), or an instance of psyclone.psyir.backend.language_writer.LanguageWriter) – a backend visitor to convert PSyIR expressions to a representation in the selected language. This is used when creating error and warning messages.

Raises
InternalError – if the number of components in this signature is different from the number of indices in component_indices.

property var_name

Returns
the actual variable name, i.e. the first component of the signature.

Return type
str

14.2.2 VariablesAccessInfo

The *VariablesAccessInfo* class is used to store information about all accesses in a region of code. To collect access information, the function *reference_accesses()* for the code region must be called. It will add the accesses for the PSyIR subtree to the specified instance of *VariablesAccessInfo*.

`Node.`**`reference_accesses`**`(`*var_accesses*`)`

Get all variable access information. The default implementation just recurses down to all children.

Parameters

var_accesses (*psyclone.core.access_info.VariablesAccessInfo*) – Stores the output results.

class psyclone.core.access_info.**VariablesAccessInfo**(*nodes=None*)

This class stores all *SingleVariableAccessInfo* instances for all variables in the corresponding code section. It maintains 'location' information, which is an integer number that is increased for each new statement. It can be used to easily determine if one access is before another.

Parameters

nodes (None, psyclone.psyir.nodes.Node or List[psyclone.psyir.nodes.Node]) – optional, a single PSyIR node or list of nodes from which to initialise this object.

__str__()

Gives a shortened visual representation of all variables and their access mode. The output is one of: READ, WRITE, READ+WRITE, or READWRITE for each variable accessed. READ+WRITE is used if the statement (or set of statements) contain individual read and write accesses, e.g. 'a=a+1'. In this case two accesses to *a* will be recorded, but the summary displayed using this function will be 'READ+WRITE'. Same applies if this object stores variable access information about more than one statement, e.g. 'a=b; b=1'. There would be two different accesses to 'b' with two different locations, but the string representation would show this as READ+WRITE. If a variable is is passed to a kernel for which no individual variable information is available, and the metadata for this kernel indicates a READWRITE access, this is marked as READWRITE in the string output.

add_access(*signature*, *access_type*, *node*, *component_indices=None*)

Adds access information for the variable with the given signature. If the *component_indices* parameter is not an instance of *ComponentIndices*, it is used to construct an instance. Therefore it can be None, a list or a list of lists of PSyIR nodes. In the case of a list of lists, this will be used unmodified to construct the ComponentIndices structures. If it is a simple list, it is assumed that it contains the indices used in accessing the last component of the signature. For example, for *a%b* with *component_indices=[i,j]*, it will create *[[], [i,j]* as component indices, indicating that no index is used in the first component *a*. If the access is supposed to be for *a(i)%b(j)*, then the *component_indices* argument must be specified as a list of lists, i.e. *[[i], [j]]*.

Parameters

- **signature** (psyclone.core.Signature) – the signature of the variable.
- **access_type** (psyclone.core.access_type.AccessType) – the type of access (READ, WRITE, ...)
- **node** (psyclone.psyir.nodes.Node instance) – Node in PSyIR in which the access happens.
- **component_indices** (*psyclone.core.component_indices.ComponentIndices*, or any other type that can be used to construct a ComponentIndices instance (None, List[psyclone.psyir.nodes.Node] or List[List[psyclone.psyir.nodes.Node]])) – index information for the access.

property all_signatures

Returns
all signatures contained in this instance, sorted (in order to make test results reproducible).

Return type
List[psyclone.core.signature]

has_read_write(*signature*)

Checks if the specified variable signature has at least one READWRITE access (which is typically only used in a function call).

Parameters
signature (psyclone.core.Signature) – signature of the variable

Returns
True if the specified variable name has (at least one) READWRITE access.

Return type
bool

Raises
KeyError if the signature cannot be found.

is_read(*signature*)

Checks if the specified variable signature is at least read once.

Parameters
signature (psyclone.core.Signature) – signature of the variable

Returns
True if the specified variable name is read (at least once).

Return type
bool

Raises
KeyError if the signature cannot be found.

is_written(*signature*)

Checks if the specified variable signature is at least written once.

Parameters
signature (psyclone.core.Signature) – signature of the variable.

Returns
True if the specified variable is written (at least once).

Return type
bool

Raises
KeyError if the signature name cannot be found.

property location

Returns the current location of this instance, which is the location at which the next accesses will be stored. See the Developers' Guide for more information.

Returns
the current location of this object.

Return type
int

merge(*other_access_info*)

Merges data from a VariablesAccessInfo instance to the information in this instance.

Parameters
other_access_info (*psyclone.core.access_info.VariablesAccessInfo*) – the other VariablesAccessInfo instance.

next_location()

Increases the location number.

This class collects information for each variable used in the tree starting with the given node. A *VariablesAccessInfo* instance can store information about variables in high-level concepts such as a kernel, as well as for language-level PSyIR. You can pass a single instance to more than one call to *reference_accesses()* in order to add more variable access information, or use the *merge()* function to combine two *VariablesAccessInfo* objects into one. It is up to the user to keep track of which statements (PSyIR nodes) a given *VariablesAccessInfo* instance is holding information about.

14.2.3 SingleVariableAccessInfo

The class *VariablesAccessInfo* uses a dictionary of *psyclone.core.access_info.SingleVariableAccessInfo* instances to map from each variable to the accesses of that variable. When a new variable is detected when adding access information to a *VariablesAccessInfo* instance via *add_access()*, a new instance of *SingleVariableAccessInfo* is added, which in turn stores all access to the specified variable.

class psyclone.core.access_info.**SingleVariableAccessInfo**(*signature*)

This class stores a list with all accesses to one variable.

Parameters
signature (psyclone.core.Signature) – signature of the variable.

add_access_with_location(*access_type*, *location*, *node*, *component_indices*)

Adds access information to this variable.

Parameters

- **access_type** (psyclone.core.access_type.AccessType) – the type of access (READ, WRITE, ...)
- **location** (*int*) – location information
- **node** (psyclone.psyir.nodes.Node) – Node in PSyIR in which the access happens.
- **component_indices** (*psyclone.core.component_indices.ComponentIndices*) – indices used for each component of the access.

property all_accesses

Returns
a list with all AccessInfo data for this variable.

Return type
List[*psyclone.core.access_info.AccessInfo*]

property all_read_accesses

Returns
a list with all AccessInfo data for this variable that involve reading this variable.

Return type
List[*psyclone.core.access_info.AccessInfo*]

property all_write_accesses

Returns
a list with all AccessInfo data for this variable that involve writing this variable.

Return type
List[*psyclone.core.access_info.AccessInfo*]

change_read_to_write()

This function is only used when analysing an assignment statement. The LHS has first all variables identified, which will be READ. This function is then called to change the assigned-to variable on the LHS to from READ to WRITE. Since the LHS is stored in a separate SingleVariableAccessInfo class, it is guaranteed that there is only one entry for the variable.

has_read_write()

Checks if this variable has at least one READWRITE access.

Returns
True if this variable is read (at least once).

Return type
bool

is_accessed_before(*reference*)

Returns True if this variable is accessed before the specified reference, and False if not. This is equivalent to testing that 'reference' is the very first access, but this function will also verify that 'reference' is indeed in the list of accesses.

Parameters
reference (psyclone.psyir.nodes.Reference) – the reference at which to stop for access checks.

Returns
True if this variable is read before the specified reference, and False if not.

Return type
bool

Raises
ValueError – if the specified reference is not in the list of all accesses.

is_array(*index_variable=None*)

Checks if the variable is used as an array, i.e. if it has an index expression. If the optional *index_variable* is specified, this variable must be used in (at least one) index access in order for this variable to be considered as an array.

Parameters
index_variable (*str*) – only considers this variable to be used as array if there is at least one access using this index_variable.

Returns
true if there is at least one access to this variable that uses an index.

Return type
bool

is_read()

Returns
True if this variable is read (at least once).

Return type
bool

`is_read_before`(*reference*)

Returns True if this variable is read before the specified reference, and False if not.

Parameters
reference (`psyclone.psyir.nodes.Reference`) – the reference at which to stop for access checks.

Returns
True if this variable is read before the specified reference, and False if not.

Return type
bool

Raises
ValueError – if the specified reference is not in the list of all accesses.

`is_read_only`()

Checks if this variable is always read, and never written.

Returns
True if this variable is read only.

Return type
bool

`is_written`()

Returns
True if this variable is written (at least once).

Return type
bool

`is_written_before`(*reference*)

Returns True if this variable is written before the specified reference, and False if not.

Parameters
reference (`psyclone.psyir.nodes.Reference`) – the reference at which to stop for access checks.

Returns
True if this variable is written before the specified reference, and False if not.

Return type
bool

Raises
ValueError – if the specified reference is not in the list of all accesses.

`property signature`

Returns
the signature for which the accesses are stored.

Return type
`psyclone.core.Signature`

property var_name

Returns
the name of the variable whose access info is managed.

Return type
str

14.2.4 AccessInfo

The class *SingleVariableAccessInfo* uses a list of *psyclone.core.access_info.AccessInfo* instances to store all accesses to a single variable. A new instance of *AccessInfo* is appended to the list whenever *add_access_with_location()* is called.

class psyclone.core.access_info.**AccessInfo**(*access_type*, *location*, *node*, *component_indices=None*)

This class stores information about a single access pattern of one variable (e.g. variable is read at a certain location). A location is a number which can be used to compare different accesses (i.e. if one access happens before another). Each consecutive location will have an increasing location number, but read and write accesses in the same statement will have the same location number. If the variable accessed is an array, this class will also store the indices used in the access. Note that the name of the variable is not stored in this class. It is a helper class used in the *SingleVariableAccessInfo* class, which stores all *AccessInfo* objects for a variable, and it stores the name of the variable.

Parameters

- **access** – the access type.
- **location** (*int*) – a number used in ordering the accesses.
- **node** (psyclone.psyir.nodes.Node) – Node in PSyIR in which the access happens.
- **component_indices** (None, [], a list or a list of lists of psyclone.psyir.nodes.Node objects, or an object of type *psyclone.core.component_indices.ComponentIndices*) – indices used in the access, defaults to None.

property access_type

Returns
the access type.

Return type
psyclone.core.access_type.AccessType

change_read_to_write()

This changes the access mode from READ to WRITE. This is used for processing assignment statements, where the LHS is first considered to be READ, and which is then changed to be WRITE.

Raises
InternalError – if the variable originally does not have READ access.

property component_indices

This function returns the list of accesses used for each component as an instance of ComponentIndices. For example, *a(i)%b(j,k)%c* will return an instance of ComponentIndices representing *[[i], [j, k], []]*. In the case of a simple scalar variable such as *a*, the *component_indices* will represent *[[]]*.

Returns
the indices used in this access for each component.

Return type
psyclone.core.component_indices.ComponentIndices

`is_array()`

Test if any of the components has an index. E.g. an access like a(i)%b would still be considered an array.

Returns
if any of the variable components uses an index, i.e. the variable is an array.

Return type
bool

`property location`

Returns
the location information for this access. Please see the Developers' Guide for more information.

Return type
int

`property node`

Returns
the PSyIR node at which this access happens.

Return type
`psyclone.psyir.nodes.Node`

14.2.5 Indices

The *AccessInfo* class stores the original PSyIR node that contains the access, but it also stores the indices used in a simplified form, which makes it easier to analyse dependencies without having to analyse a PSyIR tree for details. The indices are stored in the ComponentIndices object that each access has, which can be accessed using the *component_indices* property of an *AccessInfo* object.

`class psyclone.core.access_info.ComponentIndices(indices=None)`

This class stores index information for variable accesses. It stores one index list for each component of a variable, e.g. for *a(i)%b(j)* it would store *[[i], [j]]*. Even for scalar accesses an empty list is stored, so *a* would have the component indices *[[]]*, and *a%b* would have *[[], []]*. Each member of this list of lists is the PSyIR node describing the array expression used.

As a shortcut, the *indices* parameter can be None or an empty list (which then creates the component indices as *[[]]*, i.e. indicating a scalar access), a list *l* (which will then create the component indices as *[l]*, i.e. a single component variable, which uses all the indices in the list *l* as array indices).

Parameters
indices (None, [], a list or a list of lists of `psyclone.psyir.nodes.Node`) – the indices from which to create this object.

Raises

- **InternalError** – if the indices parameter is not None, a list or a list of lists.
- **InternalError** – if the indices parameter is a list, and some but not all members are a list.

`__getitem__(indx)`

Allows to use this class as a dictionary. If *indx* is an integer, the list of indices for the specified component is returned. If *indx* is a tuple (as returned from *iterate*), it will return the PSyIR of the index for the specified component at the specified dimension.

Returns
either the list of indices for a component, or the index PSyIR node for the specified tuple.

Return type
list of psyclone.psyir.nodes.Node, or psyclone.psyir.nodes.Node

Raises
IndexError – if a tuple is given and one of the indices is outside of the valid range.

__len__()

Returns
the number of components in this class.

Return type
int

get_subscripts_of(*set_of_vars*)

This function returns a flat list of which variable from the given set of variables is used in each subscript. For example, the access *a(i+i2)%b(j*j+k,k)%c(l,5)* would have the component_indices *[[i+i2], [j*j+k,k], [l,5]]*. If the set of variables is *(i,j,k)*, then *get_subscripts_of* would return *[{i},{j,k},{k},{l},{}]*.

Parameters
set_of_vars (*Set[str]*) – set with name of all variables.

Returns
a list of sets with all variables used in the corresponding array subscripts as strings.

Return type
List[Set[str]]

property indices_lists

Returns
the component indices list of lists.

Return type
list of list of psyclone.psyir.nodes.Node

is_array()

Test whether there is an index used in any component. E.g. an access like *a(i)%b* with indices *[[i], []]* would still be considered an array.

Returns
whether any of the variable components uses an index, i.e. the variable is an array.

Return type
bool

iterate()

Allows iterating over all component indices. It returns a tuple with two elements, the first one indicating the component, the second the dimension for which the index is. The return tuple can be used in a dictionary access (see *__getitem__*) of this object.

Returns
a tuple of the component index and index.

Return type
tuple(int, int)

The *ComponentIndices* class provides an array-like accessor for the internal data structure, you can use *len(component_indices)* to get the number of components for which array indices are stored. The information can be accessed using array subscription syntax, e.g.: *component_index[0]* will return the list of array indices used in the first component. You can also use a 2-tuple to select a component and a dimension at the same time, e.g. *component_indices[(0,1)]*, which will return the index used in the second dimension of the first component.

ComponentIndices provides an easy way to iterate over all indices using its *iterate()* method, which returns all valid 2-tuples of component index and dimension index. For example:

```
# access_info is an AccessInfo instance and contains one access. This
# could be as simple as `a(i,j)`, but also something more complicated
# like `a(i+2*j)%b%c(k, 1)`.
for indx in access_info.component_indices.iterate():
    # indx is a 2-tuple of (component_index, dimension_index)
    psyir_index = access_info.component_indices[indx]

# Using enumerate:
for count, indx in enumerate(access_info.component_indices.iterate()):
    psyir_index = access_info.component_indices[indx]
    # fortran writer converts a PSyIR node to Fortran:
    print("Index-id {0} of 'a(i,j)': {1}"
          .format(count, fortran_writer(psyir_index)))
```

```
Index-id 0 of 'a(i,j)': i
Index-id 1 of 'a(i,j)': j
```

To find out details about an index expression, you can either analyse the tree (e.g. using *walk*), or use the variable access functionality again. Below is an example that shows how this is done to determine if an array expression contains a reference to a given variable specified as a signature in the variable *index_variable*. The variable *access_info* is an instance of *AccessInfo* and contains the information about one reference. The function *reference_accesses* is used to analyse the index expression. Typically, this code would be wrapped in an outer loop over all accesses.

```
index_variable = Signature("i")
# access_info contains the access information for a single
# reference, e.g. `a(i+2*j)%b%c(k, 1)`. Loop over all
# individual index expressions ("i+2*j", then "k" and "l"
# in the example above).
for indx in access_info.component_indices.iterate():
    index_expression = access_info.component_indices[indx]

    # Create an access info object to collect the accesses
    # in the index expression
    accesses = VariablesAccessInfo(index_expression)

    # Then test if the index variable is used. Note that
    # the key of `access` is a signature, as is the `index_variable`
    if index_variable in accesses:
        # The index variable is used as an index
        # at the specified location.
        print("Index '{0}' is used.".format(str(index_variable)))
        break
else:
    print("Index '{0}' is not used.".format(str(index_variable)))
```

14.2.6 Access Location

Variable accesses are stored in the order in which they happen. For example, an assignment *a=a+1* will store two access for the variable *a*, the first one being a READ access, followed by a WRITE access, since this is the order in which the accesses are executed. Additionally, the function *reference_accessess()* keeps track of the location at which the accesses happen. A location is an integer number, starting with 0, which is increased for each new statement. This makes it possible to compare accesses to variables: if two accesses have the same location value, it means the accesses happen in the same statement, for example *a=a+1*: the READ and WRITE access to *a* will have the same location number. If on the other hand the accesses happen in two separate statements, e.g. *a=b+1; c=a+1* then the first access to *a* (and the access to *b*) will have a smaller location number than the second access to *a* (and the access to *c*). If two statements have consecutive locations, this does not necessarily mean that the statements are executed one after another. For example in if-statements the statements in the if-body are counted first, then the statements in the else-body. It is the responsibility of the user to handle these cases - for example by creating separate *VariablesAccessInfo* for statements in the if-body and for the else-body.

Note: When using different instances for an if- and else-body, the first statement of the if-body will have the same location number as the first statement of the else-body. So you can only compare location numbers from the same *VariablesAccessInformation* instance. If you merge two instances together, the locations of the merged-in instance will be appropriately increased to follow the locations of the instance to which it is merged.

The location number is not exactly a line number - several statements can be on one line, which will get different location numbers. And certain lines will not have a location number (e.g. comment lines).

As stated above, one instance of *VariablesAccessInfo* can be extended by adding additional variable information. It is the responsibility of the user to make sure the accesses are added in the right order - the *VariablesAccessInfo* object will always assume accesses happen at the current location, and a call to *next_location()* is required (internally) to increase the location number.

Note: It is not possible to add access information about an earlier usage to an existing *VariablesAccessInfo* object.

14.2.7 Access Examples

Below we show a simple example of how to use this API. This is from the *psyclone.psyir.nodes.OMPParallelDirective*, and it is used to determine a list of all the scalar variables that must be declared as thread-private. Note that this code does not handle the usage of *first-private* declarations.

```
result = set()
var_accesses = VariablesAccessInfo()
omp_directive.reference_accesses(var_accesses)
for signature in var_accesses.all_signatures:
    if signature.is_structure:
        # A lookup in the symbol table for structures are
        # more complicated, so ignore them for this example.
        continue
    var_name = str(signature)
    symbol = symbol_table.lookup(var_name)
    # Ignore variables that are arrays, we only look at scalar ones.
    # The `is_array_access` function will take information from
    # the access information as well as from the symbol table
    # into account.
```

(continues on next page)

(continued from previous page)

```
    access_info = var_accesses[signature]
    if symbol.is_array_access(access_info=access_info):
        # It's not a scalar variable, so it will not be private
        continue

    # If a scalar variable is only accessed once, it is either a coding
    # error or a shared variable - anyway it is not private
    accesses = access_info.all_accesses
    if len(accesses) == 1:
        continue

    # We have at least two accesses. If the first one is a write,
    # assume the variable should be private:
    if accesses[0].access_type == AccessType.WRITE:
        print("Private variable", var_name)
        result.add(var_name.lower())
```

The next, hypothetical example shows how the *VariablesAccessInfo* class can be used iteratively. Assume that you have a function *can_be_parallelised* that determines if the given variable accesses can be parallelised, and the aim is to determine the largest consecutive block of statements that can be executed in parallel. The accesses of one statement at a time are added until we find accesses that would prevent parallelisation:

```
# Create an empty instance to store accesses
accesses = VariablesAccessInfo()
list_of_parallelisable_statements = []
for next_statement in statements:
    # Add the variable accesses of the next statement to
    # the existing accesses:
    next_statement.reference_accesses(accesses)
    # Stop when the next statement can not be parallelised
    # together with the previous accesses:
    if not can_be_parallelised(accesses):
        break
    list_of_parallelisable_statements.append(next_statement)

print("The first {0} statements can be parallelised."
      .format(len(list_of_parallelisable_statements)))
```

Note: There is a certain overlap in the dependency analysis code and the variable access API. More work on unifying those two approaches will be undertaken in the future. Also, when calling *reference_accesses()* for a Dynamo or GOcean kernel, the variable access mode for parameters is taken from the kernel metadata, not from the actual kernel source code.

14.3 Dependency Tools

PSyclone contains a class that builds upon the data-dependency functionality to provide useful tools for dependency analaysis. It especially provides messages for the user to indicate why parallelisation was not possible. It uses *SymPy* internally to compare expressions symbolically.

class psyclone.psyir.tools.dependency_tools.**DependencyTools**(*loop_types_to_parallelise=None*, *language_writer=None*)

This class provides some useful dependency tools, allowing a user to overwrite/modify functions depending on the application. It includes a messaging system where functions can store messages that might be useful for the user to see.

Parameters

- **loop_types_to_parallelise** (*Optional[List[str]]*) – A list of loop types that will be considered for parallelisation. An example loop type might be 'lat', indicating that only loops over latitudes should be parallelised. The actually supported list of loop types is specified in the PSyclone config file. This can be used to exclude for example 1-dimensional loops.
- **language_writer** (Optional[psyclone.psyir.backend.visitor.PSyIRVisitor]) – a backend visitor to convert PSyIR expressions to a representation in the selected language. This is used for creating error and warning messages.

Raises

TypeError – if an invalid loop type is specified.

can_loop_be_parallelised(*loop*, *only_nested_loops=True*, *test_all_variables=False*, *signatures_to_ignore=None*)

This function analyses a loop in the PsyIR to see if it can be safely parallelised over the specified variable.

Parameters

- **loop** (psyclone.psyir.nodes.Loop) – the loop node to be analysed.
- **only_nested_loops** (*bool*) – if True, a loop must have an inner loop in order to be considered parallelisable (default: True).
- **test_all_variables** (*bool*) – if True, it will test if all variable accesses can be parallelised, otherwise it will stop after the first variable is found that can not be parallelised.
- **signatures_to_ignore** (list of psyclone.core.Signature) – list of signatures for which to skip the access checks.

Returns

True if the loop can be parallelised.

Return type

bool

Raises

TypeError – if the supplied node is not a Loop.

get_all_messages()

Returns all messages that have been stored by the last function the user has called.

Returns

a list of all messages.

Return type

List[str]

get_in_out_parameters(*node_list*)

Return a 2-tuple of lists that contains all variables that are input parameters (first entry) and output parameters (second entry). This function calls get_input_parameter and get_output_parameter, but avoids the repeated computation of the variable usage.

Parameters
node_list (List[psyclone.psyir.nodes.Node]) – list of PSyIR nodes to be analysed.

Returns
a 2-tuple of two lists, the first one containing the input parameters, the second the output parameters.

Return type
Tuple[List[psyclone.core.Signature], List[psyclone.core.Signature])

get_input_parameters(*node_list*, *variables_info=None*)

Return all variables that are input parameters, i.e. are read (before potentially being written).

Parameters

- **node_list** (List[psyclone.psyir.nodes.Node]) – list of PSyIR nodes to be analysed.
- **variables_info** (psyclone.core.variables_info.VariablesAccessInfo) – optional variable usage information, can be used to avoid repeatedly collecting this information.

Returns
a list of all variable signatures that are read.

Return type
List[psyclone.core.Signature]

get_output_parameters(*node_list*, *variables_info=None*)

Return all variables that are output parameters, i.e. are written.

Parameters

- **node_list** (List[psyclone.psyir.nodes.Node]) – list of PSyIR nodes to be analysed.
- **variables_info** (Optional[psyclone.core.variables_info.VariablesAccessInfo]) – optional variable usage information, can be used to avoid repeatedly collecting this information.

Returns
a list of all variable signatures that are written.

Return type
List[psyclone.core.Signature]

An example of how to use this class is shown below. It takes a list of statements (i.e. nodes in the PSyIR), and adds 'OMP DO' directives around loops that can be parallelised:

```
parallel_loop = OMPLoopTrans()
# The loops in the Fortran functions that must be parallelised
# are over the 'lat' domain. Note that the psyclone config
# file specifies the mapping of loop variable to type, e.g.:
#
#   mapping-lat = var: jj, start: 1, stop: jpj
#
# This means any loop using the variable 'jj' is considered a
# loop of type 'lat'
```

(continues on next page)

(continued from previous page)

```
dt = DependencyTools(["lat"])

for statement in loop_statements:
    if isinstance(statement, Loop):
        # Check if there is a variable dependency that might
        # prevent this loop from being parallelised:
        if dt.can_loop_be_parallelised(statement):
            parallel_loop.apply(statement)
        else:
            # Print all messages from the dependency analysis
            # as feedback for the user:
            for message in dt.get_all_messages():
                print(message)
```

CHAPTER

FIFTEEN

SYMPY

PSyclone uses the symbolic maths package "SymPy" for comparing expressions symbolically, e.g. a comparison like `i+j > i+j-1` will be evaluated to be true.

The SymPy package is wrapped in the PSyclone class `SymbolicMaths`:

class psyclone.core.**SymbolicMaths**

A wrapper around the symbolic maths package 'sympy'. It provides convenience functions for PSyclone. It has a Singleton access, e.g.:

```
>>> from psyclone.psyir.backend.fortran import FortranWriter
>>> from psyclone.core import SymbolicMaths
>>> sympy = SymbolicMaths.get()
>>> # Assume lhs is the PSyIR of 'i+j', and rhs is 'j+i'
>>> if sympy.equal(lhs, rhs):
...     writer = FortranWriter()
...     print("'{0}' and '{1}' are equal."
...           .format(writer(lhs), writer(rhs)))
'i + j' and 'j + i' are equal.
```

static equal(*exp1*, *exp2*)

Test if the two PSyIR expressions are symbolically equivalent.

Parameters

- **exp1** (Optional[psyclone.psyir.nodes.Node]) – the first expression to be compared.
- **exp2** (Optional[psyclone.psyir.nodes.Node]) – the first expression to be compared.

Returns

whether the two expressions are mathematically identical.

Return type

bool

static expand(*expr*)

Expand a PSyIR expression. This is done by converting the PSyIR expression to a sympy expression, applying the expansion operation and then converting the resultant output back into PSyIR.

Currently does not work if the PSyIR expression contains Range nodes, see issue #1655.

Parameters

expr (py:class:*psyclone.psyir.nodes.Node*) – the expression to be expanded.

static get()

Static function that creates (if necessary) and returns the singleton SymbolicMaths instance.

Returns
the instance of the symbolic maths class.

Return type
psyclone.core.SymbolicMaths.

static never_equal(*exp1*, *exp2*)

Returns if the given SymPy expressions are guaranteed to be different regardless of the values of symbolic variables. E.g. *n-1* and *n* are always different, but *5* and *n* are not always different.

Parameters

- **exp1** (py:class:*psyclone.psyir.nodes.Node*) – the first expression to be compared.
- **exp2** (py:class:*psyclone.psyir.nodes.Node*) – the first expression to be compared.

Returns
whether or not the expressions are never equal.

Return type
bool

static solve_equal_for(*exp1*, *exp2*, *symbol*)

Returns all solutions of exp1==exp2, solved for the specified symbol. It restricts the solution domain to integer values. If there is an infinite number of solutions, it returns the string 'independent', indicating that the solution of exp1==exp2 does not depend on the specified symbol. This is done to avoid that the SymPy instance representing an infinite set is used elsewhere in PSyclone (i.e. creating a dependency in other modules to SymPy). Otherwise a standard Python set is returned that stores the solutions.

Parameters

- **exp1** (sympy.core.basic.Basic) – the first expression.
- **exp2** (sympy.core.basic.Basic) – the second expression.
- **symbol** (sympy.core.symbol.Symbol) – the symbol for which to solve.

Returns
a set of solutions, or the string "independent".

Return type
Union[set, str]

This can be used for tests of nodes in the PSyIR. For example, the NEMO loop fuse transformation checks that the loops to be fused have the same loop boundaries using code like this:

```
from psyclone.core import SymbolicMaths
from psyclone.psyir.backend.fortran import FortranWriter

# Assume loop1 is ``do i=1, k`` and loop2 ``do i=5+k-4-k, 2*k-k-1``.

writer = FortranWriter()
sym_maths = SymbolicMaths.get()
if sym_maths.equal(loop1.start_expr, loop2.start_expr):
    print("'{0}' equals '{1}'".format(writer(loop1.start_expr),
                                      writer(loop2.start_expr)))
if not sym_maths.equal(loop1.stop_expr, loop2.stop_expr):
    print("'{0}' does not equal '{1}'".format(writer(loop1.stop_expr),
                                              writer(loop2.stop_expr)))
```

```
'1' equals '5 + k - 4 - k'
'k' does not equal '2 * k - k - 1'
```

15.1 Handling of PSyIR Structures and Arrays

SymPy has no concept of structure references or array syntax like `a(i)%b` in Fortran. But this case is not handled especially, the PSyIR is converted to Fortran syntax and is provided unmodified to SymPy. SymPy interprets the `%` symbol as modulo function, so the expression above is read as `Mod(a(i), b)`. This interpretation achieves the expected outcome when comparing structures and array references. For example, `a(i+2*j-1)%b(k-i)` and `a(j*2-1+i)%b(-i+k)` will be considered to be equal:

1. Converting the two expressions to SymPy internally results in `Mod(a(i+2*j-1), b(k-i))` and `Mod(a(j*2-1+i, b(-i+k))`.
2. Since nothing is known about the arguments of any of the `Mod` functions, SymPy will first detect that the same function is called in both expression, and then continue to compare the arguments of this function.
3. The first arguments are `a(i+2*j-1)` and `a(j*2-1+i)`. The name `a` is considered an unknown function. SymPy detects that both expressions appear to call the same function, and it will therefore compare the arguments.
4. SymPy compares `i+2*j-1` and `j*2-1+i` symbolically, and evaluate these expressions to be identical. Therefore, the two expressions `a(...)` are identical, so the first arguments of the `Mod` function are identical.
5. Similarly, it will then continue to evaluate the second argument of the `Mod` function (`b(...)`), and evaluate them to be identical.
6. Since all arguments of the `Mod` function are identical, SymPy will report these two functions to be the same, which is the expected outcome.

15.2 Converting PSyIR to Sympy - SymPyWriter

The method `equal` of the SymbolicMaths class expects two PSyIR nodes. It converts these expression first into strings before parsing them as SymPy expressions. The conversion is done with the SymPyWriter class. As described in the previous section, a member of a structure in Fortran becomes a stand alone symbol or function in sympy. The SymPy writer will rename members to better indicate that they are members: an expression like `a%b%c` will be written as `a%a_b%a_b_c`, which SymPy then parses as `MOD(a, MOD(a_b, a_b_c))`. This convention makes it easier to identify what the various expressions in SymPy are.

This handling of member variables can result in name clashes. Consider the expression `a%b + a_b + b`. The structure access will be using two symbols `a` and `a_b` - but now there are two different symbols with the same name. Note that the renaming of the member from `b` to `a_b` is not the reason for this - without renaming the same clash would happen with the symbol `b`.

The SymPy writer uses a symbol table to make sure it creates unique symbols. It first adds all References in the expression to the symbol table, which guarantees that no Reference to an existing symbol is renamed. The writer then renames all members and makes sure it uses a unique name. In the case of `a%b + a_b + b`, it would create `a%a_b_1 + a_b + b`, using the name `a_b_1` for the member to avoid the name clash with the reference `a_b` - so an existing Symbol Reference will not be renamed, only members.

The SymPy writer mostly uses the Fortran writer, but implements the following, SymPy specific features:

1. It will declare any array access as a SymPy unknown function, and any scalar access as a SymPy symbol. These declarations are stored in a dictionary, which can be queried. This dictionary is parsed into the SymPy writer to ensure the correct interpretation of any names found in the expression. Declaring arrays as functions results in

the correct behaviour of SymPy: in case of an unknown function SymPy will compare all arguments, which are the array indices.

2. It renames members as described above. So a structure reference like `a%b` (in Fortran syntax) will create two SymPy symbols: `a` and `a_b` (or a similar name if a name clash was detected).
3. No precision or kind information is added to a constant (e.g. a Fortran value like `2_4` will be written just as `2`).
4. The intrinsic functions `Max`, `Min`, `Mod` are returned with a capitalised first letter. The Fortran writer would write them as `MAX` etc., which SymPy does not recognise and would then handle as unknown functions.

class psyclone.psyir.backend.sympy_writer.**SymPyWriter**(*type_map=None*)

Implements a PSyIR-to-sympy writer, which is used to create a representation of the PSyIR tree that can be understood by SymPy. Most Fortran expressions work as expected without modification. This class implements special handling for constants (which can have a precision attached, e.g. 2_4) and some intrinsic functions (e.g. MAX, which SymPy expects to be Max). It additionally supports accesses to structure types. A full description can be found in the manual: https://psyclone-dev.readthedocs.io/en/latest/sympy.html#sympy

Parameters
type_map (*dict of str:Sympy-data-type values*) – Optional initial mapping that contains the SymPy data type of each reference in the expressions. This is the result of the static function `psyclone.core.sympy_writer.create_type_map()`.

static convert_to_sympy_expressions(*list_of_expressions*)

This function takes a list of PSyIR expressions, and converts them all into Sympy expressions using the SymPy parser. It takes care of all Fortran specific conversion required (e.g. constants with kind specification, ...), including the renaming of member accesses, as described in https://psyclone-dev.readthedocs.io/en/latest/sympy.html#sympy

Parameters
list_of_expressions (list of `psyclone.psyir.nodes.Node`) – the list of expressions which are to be converted into SymPy-parsable strings.

Returns
the converted PSyIR expressions.

Return type
list of SymPy expressions

static create_type_map(*list_of_expressions*)

This function creates a dictionary mapping each Reference in any of the expressions to either a Sympy Function (if the reference is an array reference) or a Symbol (if the reference is not an array reference).

Parameters
list_of_expressions (list of `psyclone.psyir.nodes.Node`) – the list of expressions from which all references are taken and added to the a symbol table to avoid renaming any symbols (so that only member names will be renamed).

Returns
the dictionary mapping each reference name to a Sympy data type (Function of Symbol).

Return type
dictionary of string:Sympy-data-type values

get_operator(*operator*)

Determine the operator that is equivalent to the provided PSyIR operator. This implementation checks for certain functions that SymPy supports: Max, Min, Mod. These functions must be spelled with a capital first letter, otherwise SymPy will handle them as unknown functions. If none of these special operators are given, the base implementation is called (which will return the Fortran syntax).

Parameters
operator (psyclone.psyir.nodes.Operation.Operator) – a PSyIR operator.

Returns
the operator as string.

Return type
str

Raises
KeyError – if the supplied operator is not known.

static get_sympy_expressions_and_symbol_map(*list_of_expressions*)

This function takes a list of PSyIR expressions, and converts them all into Sympy expressions using the SymPy parser. It takes care of all Fortran specific conversion required (e.g. constants with kind specification, ...), including the renaming of member accesses, as described in https://psyclone-dev.readthedocs.io/en/latest/sympy.html#sympy It also returns the symbol map, i.e. the mapping of Fortran symbol names to SymPy Symbols.

Parameters
list_of_expressions (list of psyclone.psyir.nodes.Node) – the list of expressions which are to be converted into SymPy-parsable strings.

Returns
a 2-tuple consisting of the the converted PSyIR expressions, followed by a dictionary mapping the symbol names to SymPy Symbols.

Return type
Tuple[List[sympy.core.basic.Basic], Dict[str, sympy.core.symbol.Symbol]]

Raises
VisitorError – if an invalid SymPy expression is found.

is_intrinsic(*operator*)

Determine whether the supplied operator is an intrinsic function (i.e. needs to be used as *f(a,b)*) or not (i.e. used as $a + b$). This tests for known SymPy names of these functions (e.g. Max), and otherwise calls the function in the base class.

Parameters
operator (*str*) – the supplied operator.

Returns
true if the supplied operator is an intrinsic and false otherwise.

literal_node(*node*)

This method is called when a Literal instance is found in the PSyIR tree. For SymPy we need to handle booleans (which are expected to be capitalised: True). Real values work by just ignoring any precision information (e.g. 2_4, 3.1_wp). Character constants are not supported and will raise an exception.

Parameters
node (psyclone.psyir.nodes.Literal) – a Literal PSyIR node.

Returns
the SymPy representation for the literal.

Return type
str

Raises
TypeError – if a character constant is found, which is not supported with SymPy.

`member_node`(*node*)

In SymPy an access to a member 'b' of a structure 'a' (i.e. a%b in Fortran) is handled as the 'MOD' function *MOD(a, b)*. We must therefore make sure that a member access is unique (e.g. *b* could already be a scalar variable). This is done by creating a new name, which replaces the % with an _. So *a%b* becomes *MOD(a, a_b)*. This makes it easier to see where the function names come from. Additionally, we still need to avoid a name clash, e.g. there could already be a variable *a_b*. This is done by using a symbol table, which was prefilled with all references (*a* in the example above) in the constructor. We use the string containing the '%' as a unique tag and get a new, unique symbol from the symbol table based on the new name using _. For example, the access to member *b* in *a(i)%b* would result in a new symbol with tag *a%b* and a name like *a_b*, *a_b_1*, ...

Parameters
node (`psyclone.psyir.nodes.Member`) – a Member PSyIR node.

Returns
the SymPy representation of this member access.

Return type
str

Note: The SymPyWriter class provides the static function `convert_to_sympy_expressions` which hides the complexities of the conversion from PSyIR expressions to SymPy expressions. It is strongly recommended to only use this function when this functionality is needed.

CHAPTER

SIXTEEN

TRANSFORMATIONS

16.1 Kernel Transformations

PSyclone is able to perform kernel transformations by obtaining the PSyIR representation of the kernel with:

`CodedKern.get_kernel_schedule()`

Returns a PSyIR Schedule representing the kernel code. The Schedule is just generated on first invocation, this allows us to retain transformations that may subsequently be applied to the Schedule.

Returns
Schedule representing the kernel code.

Return type
`psyclone.psyir.nodes.KernelSchedule`

The result of *psyclone.psyGen.Kern.get_kernel_schedule* is a *psyclone.psyir.nodes.KernelSchedule* which is a specialisation of the *Routine* class with the *is_program* and *return_type* properties set to *False* and *None*, respectively.

In addition to modifying the kernel PSyIR with the desired transformations, the *modified* flag of the *CodedKern* node has to be set. This will let PSyclone know which kernel files it may have to rename and rewrite during the code generation.

16.2 Raising Transformations

Whenever the PSyIR is created from existing source code using one of the frontends, the result is language-level PSyIR. That is, it contains only nodes that can be mapped directly into a language such as C or Fortran by one of the PSyIR backends. In order to utilise domain-specific knowledge, this language level PSyIR must be 'raised' to a domain-specific PSyIR. The resulting PSyIR will then contain nodes representing higher-level concepts such as kernels or halo exchanges. This raising is performed by means of the transformations listed in the sub-sections below.

16.2.1 Raising Transformations for the NEMO API

The top-level raising transformation creates NEMO PSy layer PSyIR:

This transformation is itself implemented using three separate transformations:

16.2.2 Raising Transformations for the LFRic API

16.3 Algorithm Transformations

In order to generate the transformed version of the algorithm with normal subroutine calls to PSy-layer routines, PSyclone provides a transformation that converts an individual `AlgorithmInvokeCall` into a `Call` to an appropriate subroutine:

class psyclone.domain.common.transformations.**AlgInvoke2PSyCallTrans**(*writer=None*)

Transforms an AlgorithmInvokeCall into a standard Call to a generated PSy-layer routine.

16.3.1 Algorithm Transformations for the LFRic API

Since the LFRic API has the concept of Builtin kernels, there is more work to do when transforming an invoke into a call to a PSy layer routine and therefore there is a specialised class for this:

16.4 OpenACC

PSyclone is able to generate code for execution on a GPU through the use of OpenACC. Support for generating OpenACC code is implemented via Transformations. The specification of parallel regions and loops is very similar to that in OpenMP and does not require any special treatment. However, a key feature of GPUs is the fact that they have their own, on-board memory which is separate from that of the host. Managing (i.e. minimising) data movement between host and GPU is then a very important part of obtaining good performance.

Since PSyclone operates at the level of Invokes, it has no information about when an application starts and thus no single place in which to initiate data transfers to a GPU. (We assume that the host is responsible for model I/O and therefore for populating fields with initial values.) Fortunately, OpenACC provides support for this kind of situation with the `enter data` directive. This may be used to "define scalars, arrays and subarrays to be allocated in the current device memory for the remaining duration of the program" [ope17]. The `ACCEnterDataTrans` transformation adds an `enter data` directive to an Invoke:

The resulting generated code will then contain an `enter data` directive.

Of course, a given field may already be on the device (and have been updated) due to a previous Invoke. In this case, the fact that the OpenACC run-time does not copy over the now out-dated host version of the field is essential for correctness.

In order to support the incremental porting and/or debugging of an application, PSyclone also supports the OpenACC `data` directive that creates a statically-scoped data region. See the description of the `ACCDataTrans` transformation in the Standard Functionality section for more details.

16.5 OpenCL

PSyclone is able to generate an OpenCL [ope18] version of PSy-layer code for the GOcean 1.0 API and its associated kernels. Such code may then be executed on devices such as GPUs and FPGAs (Field-Programmable Gate Arrays).

The PSyKAl model of calling kernels for pre-determined iteration spaces is a natural fit to OpenCL's concept of an `NDRangeKernel`. However, the kernels themselves must be created or loaded at runtime, their arguments explicitly set and any arrays copied to the compute device. All of this 'boilerplate' code is generated by PSyclone. In order to minimise the changes required, the generated code is still Fortran and makes use of the FortCL library (https://github.com/stfc/FortCL) to access OpenCL functionality. We could of course generate the PSy layer in C instead but this would require further extension of PSyclone.

Consider the following invoke:

```
call invoke( compute_cu(CU_fld, p_fld, u_fld) )
```

When creating the OpenCL PSy layer for this invoke, PSyclone creates three subroutines instead of the usual one. The first, `psy_init` is responsible for ensuring that a valid kernel object is created for each kernel called by the invoke, e.g.:

```
use fortcl, only: ocl_env_init, add_kernels
...
! Initialise the OpenCL environment/device
CALL ocl_env_init
! The kernels this PSy layer module requires
kernel_names(1) = "compute_cu_code"
! Create the OpenCL kernel objects. Expects to find all of the
! compiled kernels in PSYCLONE_KERNELS_FILE.
CALL add_kernels(1, kernel_names)
```

As indicated in the comment, the `FortCL::add_kernels` routine expects to find all kernels in a pre-compiled file pointed to by the PSYCLONE_KERNELS_FILE environment variable. (A pre-compiled file is used instead of run-time kernel compilation in order to support execution on FPGAs.)

The second routine created by PSyclone sets the kernel arguments, e.g.:

```
SUBROUTINE compute_cu_code_set_args(kernel_obj, nx, cu_fld, p_fld, u_fld)
  USE clfortran, ONLY: clSetKernelArg
  USE iso_c_binding, ONLY: c_sizeof, c_loc, c_intptr_t
  ...
  INTEGER(KIND=c_intptr_t), target :: cu_fld, p_fld, u_fld
  INTEGER(KIND=c_intptr_t), target :: kernel_obj
  INTEGER, target :: nx
  ! Set the arguments for the compute_cu_code OpenCL Kernel
  ierr = clSetKernelArg(kernel_obj, 0, C_SIZEOF(nx), C_LOC(nx))
  ierr = clSetKernelArg(kernel_obj, 1, C_SIZEOF(cu_fld), C_LOC(cu_fld))
  ...
END SUBROUTINE compute_cu_code_set_args
```

The third routine generated is the ususal psy-layer routine that is responsible for calling all of the kernels. However, it must now also call `psy_init`, create buffers on the compute device (if they are not already present) and copy data over:

```
SUBROUTINE invoke_compute_cu(...)
  ...
  IF (first_time) THEN
    first_time = .false.
    CALL psy_init
    num_cmd_queues = get_num_cmd_queues()
    cmd_queues => get_cmd_queues()
    kernel_compute_cu_code = get_kernel_by_name("compute_cu_code")
  END IF
  globalsize = (/p_fld%grid%nx, p_fld%grid%ny/)
  ! Ensure field data is on device
  IF (.NOT. cu_fld%data_on_device) THEN
    size_in_bytes = int(p_fld%grid%nx*p_fld%grid%ny, 8)* &
                    c_sizeof(cu_fld%data(1,1))
```

(continues on next page)

(continued from previous page)

```
    ! Create buffer on device
    cu_fld%device_ptr = create_rw_buffer(size_in_bytes)
    ierr = clEnqueueWriteBuffer(cmd_queues(1), cu_fld%device_ptr,  &
                                CL_TRUE, 0, size_in_bytes,       &
                                C_LOC(cu_fld%data), 0, C_NULL_PTR, &
                                C_LOC(write_event))
    cu_fld%data_on_device = .true.
  END IF
  ...
END SUBROUTINE
```

Note that we use the `data_on_device` member of the field derived type (implemented in github.com/stfc/dl_esm_inf) to keep track of whether a given field has been copied to the compute device. Once all of this setup is done, the kernel itself is launched by calling `clEnqueueNDRangeKernel`:

```
ierr = clEnqueueNDRangeKernel(cmd_queues(1), kernel_compute_cu_code, &
                              2, C_NULL_PTR, C_LOC(globalsize),      &
                              C_NULL_PTR, 0, C_NULL_PTR, C_NULL_PTR)
```

16.5.1 Limitations

The current implementation only supports the conversion of a single whole Invoke to use OpenCL. In the future we may refine this functionality so that it may be applied to just a subset of kernels within an Invoke and/or to multiple invokes.

Since PSyclone knows nothing about the I/O performed by a model, the task of ensuring that the correct data is written out by a model (including when doing halo exchanges for distributed memory) is left to the dl_esm_inf library since that has the information on whether field data is local or on a remote compute device. How the data is sent or retrieved from the OpenCL device is provided by the dl_esm_inf `read_from_device_*` and `write_to_device_*` function pointers. In the current implementation it does a just-when-is-needed synchronous data transfer using a single command queue which can bottleneck the OpenCL performance if there are many I/O operations.

16.6 ArrayRange2LoopTrans

The ArrayRange2LoopTrans transformation has the following known issues:

1) code in the NEMO API remains unchanged after this transformation is applied. This is the case for all transformations that manipulate the PSyIR as the NEMO API currently manipulates and outputs the underlying fparser2 tree. In the future the NEMO API will output code from the PSyIR representation using the back-ends provided.

2) if the indices of the ranges in different array accesses that are going to be modified to use a loop index are not the same then the transformation raises an exception. For example `a(1:2) = b(2:3)`. Issue #814 captures removing this limitation.

3) at the moment, to test whether two loop ranges are the same, we first check whether they both access the full bounds of the array. If so we assume that they are the same (otherwise the code will not run). If this is not the case, then PSyclone uses *SymPy* for comparing ranges, which will consider the two ranges `range(1:n+1:1)` and `range(1:1+n:1)` to be equal.

4) there is a test for non-elementwise operations on the rhs of an assignment as it is not possible to turn this into an explicit loop. At the moment, the type of data that a PSyIR Expression Node returns can not be determined, so it is not possible to check directly for a non-elementwise operation. Fixing this issue is the subject of #685.

For the moment the test just checks for MATMUL as that is currently the only non-elementwise operation in the PSyiR.

16.7 OpenMP Tasking

OpenMP tasking is supported in PSyclone, currently by the combination of the *OMPTaskloopTrans* and the *OMPTaskwaitTrans*. Dependency analysis and handling is done by the *OMPTaskwaitTrans*, which uses its own *get_forward_dependence* function to compute them.

16.7.1 get_forward_dependence

This function searches through the current section of the PSyIR tree for the given taskloop's next forward dependency, using the dependency analysis tools provided in *psyclone.psyir.tools.dependency_tools*. It searches through the tree for all *Loop*, *OMPDoDirective*, *OMPTaskloopDirective*, and *OMPTaskwaitDirective*. It then iterates forward through these until it finds:

1) A *Loop*, *OMPDoDirective*, or *OMPTaskloopDirective* which contains a Read-after-Write (RaW) or Write-after-Read (WaR) dependency, in which case that node is returned as the next dependence if it is contained within the same *OMPSerialDirective*. If it is not contained within the same *OMPSerialDirective*, the taskloop's parent *OMPSingleDirective* is returned instead, provided it has no *nowait* clause associated with it.
2) An *OMPTaskloopDirective* within the same *OMPSingleDirective* provided the single region has no *nowait* clause associated with it. If this criteria is satisfied the taskloop directive is returned.

The forward dependency will never be a child node of the provided taskloop, and the dependency's *abs_position* will always be great than *taskloop.abs_position*.

The RaW and WaR dependencies are computed by gathering all of the variable accesses contained inside the relevant directive's subtrees (other than loop variables which are ignored), and checking for collisions between the lists. If those collisions are not both read-only, then we know there must be a RaW or WaR dependency.

If no dependency is found, then *None* is returned.

If a taskloop has no *nogroup* clause associated, it will be skipped over during the *OMPTaskwaitTransformation.apply* call, as any solvable dependencies will be satisfied by the implicit taskgroup.

These structures are the only way to satisfy dependencies between taskloops, and any other structures of dependendent taskloops will be caught by the *OMPTaskwaitTransformation.validate* call, which will raise an Error explaining why the dependencies cannot be resolved.

CHAPTER

SEVENTEEN

PSYDATA API

PSyclone provides transformations that will insert callbacks to an external library at runtime. These callbacks allow third-party libraries to access data structures at specified locations in the code.

17.1 Introduction to PSyData Classes

The PSyData API imports a user-defined data type from a PSyData module and creates an instance of this data type for each code region. It then adds a sequence of calls to methods in that instance. A simplified example:

```
USE psy_data_mod, ONLY: PSyDataType
TYPE(PSyDataType), target, save :: psy_data

CALL psy_data%PreStart("update_field_mod", "update_field_code", 1, 1)
...
```

In order to allow several different callback libraries to be used at the same time, for example to allow in-situ visualisation at the same time as checking that read-only values are indeed not modified, different module names and data types must be used.

PSyData divides its application into different classes. For example, the class "profile" is used for all profiling tools (e.g. DrHook or the NVIDIA profiling tools). This class name is used as a prefix for the module name, the `PSyDataType` and functions. So if a profiling application is linked the above code will actually look like this:

```
USE profile_psy_data_mod, ONLY: profile_PSyDataType
TYPE(profile_PSyDataType), target, save :: profile_psy_data

CALL profile_psy_data%PreStart("update_field_mod", "update_field_code", 1, 1)
```

Note: While adding the class prefix to the name of the instance variable is not necessary, it helps improves the readability of the created code.

The list of valid class prefixes is specified in the configuration file. It can be extended by the user to support additional classes:

```
[DEFAULT]
...
VALID_PSY_DATA_PREFIXES = profile, extract
```

The class prefixes supported at the moment are:

Class Prefix	Description
profile	All libraries related to profiling tools like DrHook, NVIDIA's profiling tools etc. See Profiling for details.
extract	For libraries used for kernel data extraction. See PSy Kernel Extractor (PSyKE) for details.
read_only_verify	Use a checksum to verify that variables that are read-only are not modified in a subroutine.

In the following documentation the string `PREFIX_` is used to indicate the class prefix used (e.g. `profile`).

Note: The transformations for profiling or kernel extraction allow to overwrite the default-prefix (see *Passing Parameters From the User to the Node Constructor*). This can be used to link with two different libraries of the same class at the same time, e.g. you could use `drhook_profile` and `nvidia_profile` as class prefixes. However, this would also require that the corresponding wrapper libraries be modified to use this new prefix.

17.2 Full Example

The following example shows the full code created by PSyclone for a kernel extraction (excluding declarations for user data). This code is automatically inserted by the various transformations that are based on `PSyDataTrans`, like `ProfileTrans` and `GOCeanExtractTrans`. More details can be found in *PSyDataTrans*.

```
USE extract_psy_data_mod, ONLY: extract_PSyDataType
TYPE(extract_PSyDataType), target, save :: extract_psy_data

CALL extract_psy_data%PreStart("update_field_mod", "update_field_code", 1, 1)
CALL extract_psy_data%PreDeclareVariable("a_fld", a_fld)
CALL extract_psy_data%PreDeclareVariable("b_fld", b_fld)
CALL extract_psy_data%PreEndDeclaration
CALL extract_psy_data%ProvideVariable("a_fld", a_fld)
CALL extract_psy_data%PreEnd

DO j=1,jstop+1
  DO i=1,istop+1
    CALL update_field_code(i, j, a_fld%data, b_fld%data)
  END DO
END DO

CALL extract_psy_data%PostStart
CALL extract_psy_data%ProvideVariable("b_fld", b_fld)
CALL extract_psy_data%PostEnd
```

1. A user defined type `PREFIX_PSyDataType` is imported from the module `PREFIX_psy_data_mod`.
2. A static variable of type `PREFIX_PSyDataType` is declared.
3. `PreStart` is called to indicate that a new that a new instrumented code region is started.
4. `PreDeclareVariable` is called for each variable passed to the data API either before or after the instrumented region.
5. `PreEndDeclaration` is called to indicate the end of the variable declarations.
6. `ProvideVariable` is called for each variable to be passed to the wrapper library before the instrumented region.
7. `PreEnd` is called to signal the end of all PSyData activity before the instrumented region.

8. Then the actual instrumented code region is entered.
9. After the instrumented region, a call to `PostStart` is added to indicate that all further data output occurs after the instrumented region.
10. For each variable to be passed on to the wrapper library after the instrumented region a call to `ProvideVariable` is added.
11. A call to `PostEnd` is added once all variables have been provided.

Note: Depending on the options provided to the PSyData transformation, some of the calls might not be created. For example, for a performance profiling library no variables will be declared or provided.

17.3 API

This section describes the actual PSyData API in detail. It contains all functions, data types and methods that need to be implemented for a wrapper library.

The PSyData API includes two function calls that allow initialisation and shut down of a wrapper library. These two calls must be inserted manually into the program, and their location might depend on the wrapper library used - e.g. some libraries might need to be initialised before `MPI_Init` is called, others might need to be called after. Similarly the shutdown function might need to be called before or after `MPI_Finalize`.

> **Note:** Not all PSyData libraries require these calls (for example, the NVIDIA profiling library does not need it, the data extraction libraries do not need it, ...), but any PSyData library should include (potentially empty) subroutines in order to avoid linking problems. It is the responsibility of the user to decide if the initialisation and shutdown calls are unnecessary.

Similarly, the PSyData API also includes two function calls that allow programmatic control of whether or not the underlying data-capture mechanism is active. If this functionality is required then these calls must be inserted manually into the program. They are intended to be used e.g. when profiling only a part of a program's execution or perhaps to switch on/off output of data for on-line visualisation. As with the initialisation and shutdown subroutines, any PSyData library should include implementations of these routines, even if they are empty.

> **Note:** Currently only the profiling wrapper libraries and read-only-verification libraries implement the Start and Stop routines. Wider support for all PSyData-based APIs will be addressed in Issue #824.

17.3.1 Init and Shutdown Functions

`PREFIX_PSyDataInit()`

Initialises the wrapper library used. It takes no parameters, and must be called exactly once before any other PSyData-related function is invoked. Example:

```
use profile_psy_data_mod, only : profile_PSyDataInit
...
call profile_PSyDataInit()
```

PREFIX_PSyDataShutdown()

Cleanly shuts down the wrapper library used. It takes no parameters, and must be called exactly once. No more PSyData-related functions can be called after `PSyDataShutdown` has been executed. Example:

```
use profile_psy_data_mod, only : profile_PSyDataShutdown
...
call profile_PSyDataShutdown()
```

17.3.2 Start and Stop Functions

PREFIX_PSyDataStart()

Currently not implemented in the kernel extraction wrapper.

Starts or enables the PSyData library so that subsequent calls to the API cause data to be output. For instance, if we have a time-stepping application where `timestep` holds the value of the current time step then we could turn on profiling after the first 5 steps by doing:

```
use profile_psy_data_mod, only: profile_PSyDataStart
...
if(timestep == 6) call profile_PSyDataStart()
```

(Assuming that profiling was disabled at application start by the runtime environment or by a call to `profile_PSyDataStop` - see below.)

This routine may be called any number of times but must be after `PSyDataInit()` and before `PSyDataShutdown()` (if present).

PREFIX_PSyDataStop()

Currently not implemented in the kernel extraction wrapper.

Stops or disables the PSyData library so that subsequent calls to the PSyData API have no effect. Continuing the above time-stepping example, we could turn off profiling after time step 10 by doing:

```
use profile_psy_data_mod, only: profile_PSyDataStop
...
if(timestep == 11) call profile_PSyDataStop()
```

This routine may be called any number of times but must be after `PSyDataInit()` and before `PSyDataShutdown()` (if present).

17.3.3 PREFIX_PSyDataType

The library using the PSyData API must provide a user-defined data type called `PREFIX_PSyDataType`. It is up to the application how this variable is used. PSyclone will declare the variables to be static, meaning that they can be used to accumulate data from call to call. An example of the PSyDataType can be found in the NetCDF example extraction code (see `lib/extract/netcdf/dl_esm_inf`, or NetCDF Extraction Examples for a detailed description), any of the profiling wrapper libraries (all contained in `lib/profiling`) or the read_only wrappers (in `lib/read_only`).

PreStart(*this*, *module_name*, *kernel_name*, *num_pre_vars*, *num_post_vars*)

This is the first method called when the instrumented region is executed. It takes 4 parameters (besides the implicit `PREFIX_PSyDataType` instance):

module_name

This is the name of the module in which the original Fortran source code is contained. Together with `kernel_name` it can be used to create a unique name for each instrumented region.

kernel_name
The name of the kernel that is being executed.

num_pre_vars
This is the number of variables that will be supplied using `ProvideVar` before the instrumented region is executed.

num_post_vars
This is the number of variables that will be supplied using `ProvideVar` after the instrumented region is executed. The sum `num_pre_vars+num_post_vars` is the number of variable declarations that will follow.

Typically the static `PREFIX_PSyDataType` instance can be used to store the module and kernel names if they are required later, or to allocate arrays to store variable data. This call is always created, even if no variables are to be provided.

`PreDeclareVariable`(*this*, *name*, *value*)

This method is called for each variable that will be written before or after the user-instrumented region. If a variable is written both before and after the region, the transformations will add two calls to `PreDeclareVariable` (it can be useful to provide a variable using a different name before and after, see NetCDF Extraction Examples). If no variables are to be provided to the wrapper library, this call will not be created (and there is no need to implement this function in a wrapper library).

name
This is the name of the variable as a string.

value
This is the actual content of the variable.

The same call is used for different arguments, so a generic interface is recommended to distinguish between the data types provided. The netcdf kernel writer (see NetCDF Extraction Examples) uses the following declaration (with types defined in the dl_esm_inf library):

```
generic, public :: PreDeclareVariable => DeclareScalarInteger, &
                                         DeclareScalarReal,    &
                                         DeclareFieldDouble
...
subroutine DeclareScalarInteger(this, name, value)
    implicit none
    class(extract_PSyDataType), intent(inout), target :: this
    character(*), intent(in) :: name
    integer, intent(in) :: value
...
subroutine DeclareScalarReal(this, name, value)
    implicit none
    class(extract_PSyDataType), intent(inout), target :: this
    character(*), intent(in) :: name
    real, intent(in) :: value
...
subroutine DeclareFieldDouble(this, name, value)
    use field_mod, only : r2d_field
    implicit none
    class(extract_PSyDataType), intent(inout), target :: this
    character(*), intent(in) :: name
    type(r2d_field), intent(in) :: value
...
```

PreEndDeclaration(*this*)

Called once all variables have been declared. This call is only inserted if any variables are to be provided either before or after the instrumented region (thus this call is not created for performance profiling).

ProvideVariable(*this*, *name*, *value*)

This method is called for each variable to be provided to the runtime library.

name
This is the name of the variable as a string.

value
This is the actual content of the variable.

The same method `ProvideVariable` is called to provide variable name and content before and after the user-instrumented region. Again it is expected that a library using the API will provide a generic interface to distinguish between the various possible data types, which will be different for each infrastructure library:

```
generic, public :: ProvideVariable => WriteScalarInteger, &
                                      WriteScalarReal,    &
                                      WriteFieldDouble
```

PreEnd(*this*)

The method `PreEnd` is called after all variables have been provided before the instrumented region. This call is also not inserted if no variables are provided.

PostStart(*this*)

This is the first call after the instrumented region. It does not take any parameters, but the static `PREFIX_PSyDataType` instance can be used to store the name and number of variables if required. This will be followed by calls to `ProvideVariable`, which is described above. This call is not used if no variables are provided.

PostEnd(*this*)

This method is the last call after an instrumented region. It indicates that all variables have been provided. It will always be created, even if no variables are to be provided.

Note: Only the `PreDataStart` call takes the module- and region-name as parameters. If these names are required by the PSyData runtime library in different calls, they must be stored in the `PSyData` object so that they are available when required.

17.4 PSyDataTrans

Any transformation that uses the PSyData API works by inserting a special node into the PSyclone tree representation of the program. Only at program creation time is the actual code created that implements the API. The `PSyDataTrans` transformation contained in `psyir/transformations/psy_data_trans.py` is the base class for other transformations like profiling and kernel data extraction. All derived transformations mostly add specific validations, and provide parameters to `PSyDataTrans`, including the class of the node to insert. After passing validation, `PSyDataTrans` creates an instance of the class requested, and inserts it into the tree.

class psyclone.psyir.transformations.**PSyDataTrans**(*node_class=<class 'psyclone.psyir.nodes.psy_data_node.PSyDataNode'>*)

Create a PSyData region around a list of statements. For example:

```
>>> from psyclone.parse.algorithm import parse
>>> from psyclone.parse.utils import ParseError
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> ast, invoke_info = parse(SOURCE_FILE, api=api)
>>> psy = PSyFactory(api).create(invoke_info)
>>>
>>> from psyclone.psyir.transformations import PSyDataTrans
>>> data_trans = PSyDataTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>>
>>> # Enclose all children within a single PSyData region
>>> data_trans.apply(schedule.children)
>>> # Uncomment the following line to see a text view of the schedule
>>> # print(schedule.view())
>>> # Or to use custom region name:
>>> data_trans.apply(schedule.children,
...                  {"region_name": ("module","region")})
```

Parameters

node_class (psyclone.psyir.nodes.ExtractNode) – The Node class of which an instance will be inserted into the tree (defaults to PSyDataNode).

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

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

Parameters

- **nodes** (psyclone.psyir.nodes.Node or list of psyclone.psyir.nodes.Node) – can be a single node or a list of nodes.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (PREFIX_psy_data_mod) and the PSyDataType (PREFIX_PSYDATATYPE) - a "_" will be added automatically. It defaults to "".
- **options["region_name"]** (*(str,str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

get_unique_region_name(*nodes*, *options*)

This function returns the region and module name. If they are specified in the user options, these names will just be returned (it is then up to the user to guarantee uniqueness). Otherwise a name based on the module and invoke will be created using indices to make sure the name is unique.

Parameters

- **nodes** (list of psyclone.psyir.nodes.Node) – a list of nodes.

- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.
- **options["region_name"]** (*(str,str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

property name

This function returns the name of the transformation. It uses the Python 2/3 compatible way of returning the class name as a string, which means that the same function can be used for all derived classes.

Returns
the name of this transformation as a string.

Return type
str

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

Checks that the supplied list of nodes is valid, that the location for this node is valid (not between a loop-directive and its loop), that there aren't any name clashes with symbols that must be imported from the appropriate PSyData library and finally, calls the validate method of the base class.

Parameters

- **nodes** ((list of) psyclone.psyir.nodes.Loop) – a node or list of nodes to be instrumented with PSyData API calls.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (PREFIX_psy_data_mod) and the PSyDataType (PREFIX_PSYDATATYPE) - a "_" will be added automatically. It defaults to "".
- **options["region_name"]** (*(str,str)*) – an optional name to use for this PSyData area, provided as a 2-tuple containing a location name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

Raises

- **TransformationError** – if the supplied list of nodes is empty.
- **TransformationError** – if the PSyData node is inserted between an OpenMP/ACC directive and the loop(s) to which it applies.
- **TransformationError** – if the 'prefix' or 'region_name' options are not valid.
- **TransformationError** – if there will be a name clash between any existing symbols and those that must be imported from the appropriate PSyData library.

17.5 PSyDataNode

This is the base class for any node that is being inserted into PSyclone's program tree to use the PSyData API. The derived classes will typically control the behaviour of PSyDataNode by providing additional parameters.

class psyclone.psyir.nodes.**PSyDataNode**(*ast=None*, *children=None*, *parent=None*, *options=None*)

This class can be inserted into a schedule to instrument a set of nodes. Instrument means that calls to an external library using the PSyData API will be inserted before and after the child nodes, which will give that library access to fields and the fact that a region is executed. This can be used, for example, to add performance profiling calls, in-situ visualisation of data, or for writing fields to a file (e.g. for creating test cases, or using driver to run a certain kernel only). The node allows specification of a class string which is used as a prefix for the PSyData module name (prefix_psy_data_mod) and for the PSyDataType (prefix_PSyDataType).

Parameters

- **ast** (sub-class of fparser.two.Fortran2003.Base) – reference into the fparser2 parse tree corresponding to this node.
- **children** (list of psyclone.psyir.nodes.Node) – the PSyIR nodes that are children of this node. These will be made children of the child Schedule of this PSyDataNode.
- **parent** (psyclone.psyir.nodes.Node) – the parent of this node in the PSyIR tree.
- **options** (*dictionary of string:values or None*) – a dictionary with options for transformations.
- **options["prefix"]** (*str*) – a prefix to use for the PSyData module name (prefix_psy_data_mod) and the PSyDataType (prefix_PSyDataType) - a "_" will be added automatically. It defaults to "", which means the module name used will just be psy_data_mod, and the data type PSyDataType.
- **options["region_name"]** (*(str, str)*) – an optional name to use for this PSyDataNode, provided as a 2-tuple containing a module name followed by a local name. The pair of strings should uniquely identify a region unless aggregate information is required (and is supported by the runtime library).

Raises

InternalError – if a prefix is specified that is not listed in the configuration file.

gen_code(*parent*, *options=None*)

Creates the PSyData code before and after the children of this node.

TODO #1010: This method and the lower_to_language_level below contain duplicated logic, the gen_code method will be deleted when all APIs can use the PSyIR backends.

Parameters

- **parent** (psyclone.f2pygen.BaseGen) – the parent of this node in the f2pygen AST.
- **options** (*dict of str:value or None*) – a dictionary with options for transformations.
- **options["pre_var_list"]** (*list of str*) – a list of variables to be extracted before the first child.
- **options["post_var_list"]** (*list of str*) – a list of variables to be extracted after the last child.
- **options["pre_var_postfix"]** (*str*) – an optional postfix that will be added to each variable name in the pre_var_list.

- **options["post_var_postfix"]** (*str*) – an optional postfix that will be added to each variable name in the post_var_list.

There are two ways of passing options to the PSyDataNode. The first one is used to pass parameters from the user's transformation script to the constructor of the node inserted, the second for passing parameters from a derived node to the PSyDataNode base class.

17.5.1 Passing Parameters From the User to the Node Constructor

Options can be passed from the user via the transformation to the node that will create the code. This is done by using the options dictionary that is a standard parameter for all validate and apply calls of a transformation (see Application). Besides using this dictionary for validation and application parameters, PSyDataTrans passes it to the constructor of the node that is being inserted. An example of a parameter is the region_name, where the user can overwrite the default name given to a region (which can be somewhat cryptic due to the need to be unique). The region name is validated by PSyDataTrans, and then passed to the node constructor. The PSyDataNode stores the name as an instance attribute, so that they can be used at code creation time (when gen_code is being called). Below is the list of all options that the PSyData node supports in the option dictionary:

Parameter Name	Description
region_name	Overwrites the region name used by the PSyDataNode. It must be a pair of strings: the first one being the name of the module, the second the name of the region. The names are used e.g. by the ProfileNode to define a unique region name for a profiled code region, or by ExtractNode to define the file name for the output data- and driver-files.
prefix	A prefix to be used for the module name, the user-defined data type and the variables declared for the API.

17.5.2 Passing Parameter From a Derived Node to the PSyDataNode

The PSyDataTrans.gen_code function also accepts an option dictionary, which is used by derived nodes to control code creation. The gen_code function is called internally, not directly by the user. If the gen_code function of a node derived from PSyDataNode is called, it can define this option directory to pass the parameters to the PSyDataNode's gen_code function. Here are the options that are currently supported by PSyDataNode:

Parameter Name	Description
pre_var_list	A list of the variable names to be extracted before the instrumented region.
post_var_list	A list of variable names to be extracted after the instrumented region.
pre_var_postfix	An optional postfix that will be appended to each variable name in the pre_var_list.
post_var_postfix	An optional postfix that will be appended to each variable name in the post_var_list.

If there is no variable to be provided by the PSyData API (i.e both pre_variable_list and post_variable_list are empty), then the PSyDataNode will only create a call to PreStart and PostEnd. This is utilised by the profiling node to make the profiling API libraries (see Profiling) independent of the infrastructure library (since a call to ProvideVariable can contain API-specific variable types). It also reduces the number of calls required before and after the instrumented region which can affect overall performance and precision of any measurements; see Profiling for more details.

The kernel extraction node ExtractNode uses the dependency module to determine which variables are input- and output-parameters, and provides these two lists to the gen_code() function of its base class, a PSyDataNode node. It also uses the post_var_postfix option as described under gen_code() above (see also NetCDF Extraction Examples).

17.6 PSyData Base Class

PSyclone provides a base class for all PSyData wrapper libraries. The base class is independent of the API, but it can provide implementations for scalars and arrays for all native Fortran types that are required by the API-specific implementation. The base class does not have to be used, but it provides useful functionality:

Verbosity:
It will check the `PSYDATA_VERBOSE` environment flag. If it exists, it must have a value of either 0 (no messages), 1 (some messages, typically only `PSyDataStart` and `PSyDataEnd`), or 2 (detailed messages, depending on wrapper library). All other values will result in a warning message being printed (and verbosity will be disabled). The verbosity level is available as `this%verbosity`.

Module- and Region-Name Handling:
The module stores the module name in `this%module_name`, and the region name as `this%region_name`.

Variable Index:
It automatically sets `this%next_var_index` to 1 in `PSyDataPreStart` `PSyDataPreEndDeclaration` and `PostStart`. This variable will also be increased by one for each call to a Declare- or Provide-subroutine. It can be used to provide a reproducible index for declaring and providing a variable (and it also counts the number of declared variables, which can be used in e.g. `PSyDataPreEndDeclaration` to allocate arrays).

Start/Stop Handling:
The base class maintains a module variable `is_enabled`. This is set to true at startup, and gets enabled and disabled by calling the module functions `PREFIX_PSyDataStart()` and `PREFIX_PSyDataStop()` (see *Start and Stop Functions*). It is up to the derived classes to actually use this setting. Of course it is also possible to ignore `is_enabled` and use a different mechanism. For example, the NVIDIA profiling wrapper library calls corresponding start and stop functions in the NVIDIA profiler.

Jinja Support:
The base class is creating using the template language Jinja. It is therefore easy to automatically create the base functions for the argument types actually required by the wrapper library. See *Jinja Support in the Base Class* for details.

17.6.1 Jinja Support in the Base Class

Code written for a PSyData library is often very repetitive. For example, an implementation of `PreDeclareVariable` must be provided for each data type. For LFRic that can easily result in over 10 very similar subroutines (3 basic types integer, 4- and 8-byte reals; and 4- and 8-byte arrays of one to four dimensions). In order to simplify the creation of these subroutines the templating language Jinja is being used. Jinja creates code based on an template, which makes it possible to maintain just one template implementation of a subroutine, from which the various Fortran-type specific implementation will be generated.

Jinja is used in the generic base class `PSyDataBase`, and the base class for all `ReadOnly` libraries. It is not required that any library using one of these base classes itself uses Jinja. For example, the ReadOnly library for dl_esm_inf does not use Jinja (except for processing the base class templates of course), while the ReadOnly library for LFRic does. In case of dl_esm_inf, there were only 5 data types that need to be supported, so it was easy to just list these 5 functions in a generic interface. LFRic on the other hand uses many more Fortran basic types, so it uses Jinja to create the code that declares the generic interfaces. The additional advantage is that if new data types are required by LFRic (e.g. if 5-dimensional arrays are used), there will be no code change required (except for declaring the new types in the Makefile).

The PSyData base class `PSyDataBaseType` is contained in `lib/psy_data_base.jinja`. It is processed with the script `process.py`, which will print the processed file to stdout. The `Makefile` will automatically create the file `psy_data_base.f90` from the Jinja template and compile it. If you use the base class in a wrapper library, you have to process the template in your library directory with additional parameters to specify the required types and the prefix. Besides the name of the template file to process the `process.py` script takes the following parameters:

-types:
A comma-separated list of Fortran basic types (no spaces allowed). The following type names are supported:

`real`:
32-bit floating point value

`double`:
64-bit floating point value

`int`:
32-bit integer value

`long`:
64-bit integer value

Default value is `real,double,int`.

-dims:
A comma-separated list of dimensions (no spaces allowed). Default value is `1,2,3,4`.

-prefix:
The prefix to use for the PSyData type and functions. Default is emtpy (i.e. no prefix). If you specify a prefix, you have to add the _ between the prefix and name explicitly.

-generic-declare:
If this flag is specified, the processed template will also declare a generic subroutine with all `declareXXX` functions (see *Details About Generic Interfaces* for details).

-generic-provide:
If this flag is specified, the processed template will also declare a generic subroutine with all `provideXXX` functions (see *Details About Generic Interfaces* for details).

Details About Generic Interfaces

The Fortran standard requires that a generic subroutine is declared in the same program unit in which it is defined. Therefore, if a derived class falls back to the implementation of a function in the base class, the generic subroutine must be declared in the base class, not in the derived class (though some compilers, e.g. gfortran 9 accept it). The two options `-generic-declare` and `-generic-provide` are supported so that a derived class can control where the generic subroutines should be declared: if the derived class does not implement say the 'declaration' functions itself, the `-generic-declare` command line option guarantees that the base class defines the declaration functions all as one generic subroutine. The derived class can of course extend this generic subroutine with API-specific implementations. On the other hand, if the derived class implements the 'declaration' functions (potentially calling their implementation in the base class), then the derived class must declare the generic subroutine, and the base class must not declare them as well. The same approach is used for the 'provide' functions. As an example, the ReadOnly verification library for LFRic in `lib/read_only/lfric/` uses `-generic-declare` when processing the PSyData base class (i.e. all declaration functions are implemented in the PSyData base class), and uses `-generic-provide` when processing the ReadOnly base class (i.e. all provide functions are implemented in the ReadOnly base class). The LFRic-specific implementation extends the generic subroutine with two new subroutines: `DeclareFieldDouble` and `DeclareFieldVectorDouble` (and the same for the corresponding 'provide' functions).

Explanation of Jinja Use

For each specified type name the Jinja template will create methods called `DeclareScalar{{type}}` and `ProvideScalar{{type}}` for handling scalar parameters. For array parameters, the functions `DeclareArray{{dim}}d{{type}}` and `ProvideArray{{dim}}d{{type}}` will be created for each type and each specified number of dimensions.

Below is an example of using the `process.py` script in a Makefile for a read-only verification library (taken from `lib/read_only/lfric/Makefile`):

```
PROCESS_ARGS = -prefix=read_only_verification -types=real,int,double \
               -dims=1,2,3,4

psy_data_base.f90: ../../psy_data_base.jinja
    ../../process.py $(PROCESS_ARGS) $< > psy_data_base.f90
```

This will create the processed file `psy_data_base.f90` in the directory of the library, where it will be compiled. Having a separate pre-processed source code version of the base class for each library (as opposed to one compiled base class library that is used for all libraries) has the advantage that consistent compiler settings will be used in your library, and consistent parameters have been provided specifying the required types and dimensions.

The `process.py` script provides the two variables for a template (based on its command line parameters of course): `ALL_DIMS` stores the list of required dimensions, and `ALL_TYPES` stores the type information requested when invoking `process.py`. `ALL_TYPES` is a three-tuple that lists the name to use when creating subroutine names, the Fortran data type, and number of bits for the data type. While number of bits is not used in the base class, the read-only-verification base class (see *PSyData Read-Only-Verification Base Class*) uses it. If more types are required, they can be defined in `process.py`. If the additional types need different numbers of bits and are required in a read-only library, the read-only-verification base class (`lib/read_only/read_only_base.jinja`) needs to be adjusted as well.

Below is a short excerpt that shows how these variables are defined by default, and how they are used to create subroutines and declare their parameters in `lib/read_only/read_only_base.jinja`:

```
{% if ALL_DIMS is not defined %}
   {# Support 1 to 4 dimensional arrays if not specified #}
   {% set ALL_DIMS = [1, 2, 3, 4] %}
{% endif %}

{# The types that are supported. The first entry of each tuple
   is the name used when naming subroutines and in user messages.
   The second entry is the Fortran declaration. The third entry
   is the number of bits. There is slightly different code
   required for 32 and 64 bit values (due to the fact that the
   Fortran ``transfer(value, mould)`` function leaves undefined
   bits when mould is larger than value.) #}

{% if ALL_TYPES is not defined %}
   {% set ALL_TYPES = [ ("Double", "real(kind=real64)",   64),
                        ("Real",   "real(kind=real32)",   32),
                        ("Int",    "integer(kind=int32)", 32) ] %}
{% endif %}
...
{% for name, type, bits in ALL_TYPES %}
subroutine DeclareScalar{{name}}(this, name, value)
    {{type}}, intent(in) :: value
...
```

(continues on next page)

(continued from previous page)

```
  {# Now create the declarations of all array implementations #}
  {% for dim in ALL_DIMS %}
    {# Create the ':,:...' string - DIM-times
       We repeat the list [":"] DIM-times, which is then joined #}
    {% set DIMENSION=([":"]*dim)|join(",") %}
! ------------------------------------------------------------------------
subroutine DeclareArray{{dim}}d{{name}}(this, name, value)
    {{type}}, dimension({{DIMENSION}}), intent(in) :: value
  {% endfor %}
{% endfor %}
```

Since the PSyData API relies on a generic interface to automatically call the right subroutine depending on type, a library must declare these automatically created subroutines (together with additional, library-specific versions) as one generic interface. This can either be done by explicitly listing the subroutines (which is for example done in `lib/read_only/dl_esm_inf/read_only.f90`, since it uses only 5 different data types), or using Jinja as well. The code below shows how the base class and Jinja are used in `lib/read_only/lfric/read_only.f90`. The `FieldDouble` and `FieldVectorDouble` related functions are explicitly coded in the LFRic-specific library, the rest is taken from the base class. Jinja is then used to create the list of all automatically created subroutines, which allows the declaration of one generic interface for each `PreDeclareVariable` and `ProvideVariable` function. While the use of Jinja here is only very minimal (only the list of subroutine names in the generic interfaces is created), the advantage of using Jinja here is that if a new data type is added to LFRic (e.g. a 5-dimensional array), only the parameter to `process.py` need to be changed, and the correct subroutines will be created in the base class, and the corresponding generic interfaces will be declared without further code changes:

```
procedure :: DeclareFieldDouble,  ProvideFieldDouble
procedure :: DeclareFieldVectorDouble,  ProvideFieldVectorDouble

{# Collect the various procedures for the same generic interface #}
{# ------------------------------------------------------------ #}
{% set all_declares=[] %}
{% set all_provides=[] %}
{% for name, type, bits in ALL_TYPES %}
  {{ all_declares.append("DeclareScalar"~name) or "" -}}
  {{ all_provides.append("ProvideScalar"~name) or "" -}}
  {% for dim in ALL_DIMS %}
    {{ all_declares.append("DeclareArray"~dim~"d"~name) or "" -}}
    {{ all_provides.append("ProvideArray"~dim~"d"~name) or "" -}}
  {% endfor %}
{% endfor %}

{% set indent="            " %}
generic, public :: PreDeclareVariable => &
    DeclareFieldDouble, &
    DeclareFieldVectorDouble, &
    {{all_declares|join(", &\n"+indent) }}

generic, public :: ProvideVariable => &
    ProvideFieldDouble,       &
    ProvideFieldVectorDouble, &
    {{all_provides|join(", &\n"+indent) }}
```

Note: Ending the Jinja statements with '`or ""`' avoids having `None` added to the files (which would be the output of e.g. the `append` instruction). The '`-`' before the closing '`}}`' also prevents this line from creating any white-spaces. As a result of this the processed file will not have unusual empty lines or indentation.

17.6.2 Static Functions in the Base Class

The base class provides the four static functions defined in the API (see *API*). The subroutines `PREFIX_PSyDataInit()` and `PREFIX_PSyDataShutdown()` are empty, and `PREFIX_PSyDataStart()` and `PREFIX_PSyDataStop()` function just change the module variable `is_enabled` (which can then be used by a wrapper library to enable or disable its functionality).

If you don't need specific functionality for these functions, you can just declare them in your wrapper library:

```
module MyProfileWrapperLibrary
  use psydata_base_mod, only : profile_PSyDataInit, profilePSyDataShutdown, &
                               profile_PSyDataStart, profile_PSyDataStop
```

If you need to partially change the behaviour of these functions, you have to rename them in your `use` statement, and then call the renamed base class function like this:

```
module MyProfileWrapperLibrary
...
contains

subroutine profile_PSyDataStart()
  use psydata_base_mod, only : base_PSyDataStart => profile_PSyDataStart
  ! Do something
  call base_PSyDataStart()
  ! Do something else
end subroutine profile_PSyDataStart
```

Note: The `PSyDataBase` template will use the prefix that you have specified as parameter when using the `process.py` script for naming the static functions. So as an alternative to renaming the symbol when importing them you could also specify a different prefix when processing the base Jinja file, and use this name.

17.7 PSyData Read-Only-Verification Base Class

The ReadOnlyVerification transformation uses the PSyData API to verify that read-only arguments to a subroutine call are not changed. It does this by computing and storing a checksum of each read-only parameter to a subroutine before the call, and verifying that this checksum is not changed after the subroutine call. Since the API-specific instances share a significant part of code (all functions for the non API-specific Fortran types, e.g. scalar values and plain Fortran arrays), a common base class is implemented for these, based on the PSyData Base class (see *PSyData Base Class*). The `ReadOnlyBaseType` is provided as a Jinja template as well (see *Jinja Support in the Base Class*), see `lib/read_only/read_only_base.jinja`. It takes the same parameters as the `PSyDataBaseType`, which makes it easy to make sure the PSyDataBaseClass and ReadOnly base classes are created with the same settings (like supported Fortran types and number of dimensions).

This Read-Only-Verification base class uses the PSyData base class. It uses the Declaration functions to count how many variables will be provided. In `PreEndDeclaration` a checksum array will be allocated.

Note: The `PreStart` function gets the number of variables as a parameter. The decision not to use this value for allocating the array is that the LFRic read-only implementation stores several checksums for one variable of a certain type (VectorField). The `DeclareVariable` functions for VectorFields counts the right number of checksums required.

The ReadOnlyBase base class uses the information about the number of bits for the data types to implement the checksum functions. One complication is that the Fortran `transfer` function results in undefined bits when transferring e.g. a 32-bit value into a 64-bit variable. Therefore any 32-bit value is first transferred to a 32-bit integer value, which is then assigned to the 64-bit integer value, which is added to the overall checksum value.

The two API-specific ReadOnlyVerification libraries are both based on this base class. Therefore they need only implement the checksum functions for the API-specific types - `Field` and `VectorFields` in LFRic, and `Field` in GOcean.

17.8 Profiling

The command line options and transformations available to a user are described in the PSyclone User's guide (Profiling). This section describes how the PSyData API is used to implement the profiling wrapper API, and which functions must be provided in a wrapper library to allow other existing profiling tools to be used.

17.8.1 Profiling API

PSyclone uses the PSyData API to allow implementation of profile wrapper libraries that connect to various existing profiling tools. For each existing profiling tool a simple interface library needs to be implemented that maps the PSyclone PSyData calls to the corresponding call for the profiling tool.

Since the profiling API does not need access to any fields or variables, PSyclone will only create calls to `PreStart` and `PostEnd`. The profiling wrapper libraries also need the static initialisation and shutdown functions `profile_PSyDataInit` and `profile_PSyDataShutdown`. Details can be found in the section *API*.

The examples in the `lib/profiling` directory show various ways in which the opaque data type `profile_PSyDataType` can be used to interface with existing profiling tools - for example by storing an index used by the profiling tool in `profile_PSyDataType`, or by storing pointers to the profiling data to be able to print all results in a ProfileFinalise() subroutine. Some of the wrapper libraries use the PSyData base class (e.g. dl_timer, simple_timing, template), others do not (e.g. NVIDIA profiling, DrHook wrapper).

17.9 Kernel Extraction (PSyKE)

The PSyclone Kernel Extraction functionality (see PSy Kernel Extractor (PSyKE)) also relies on the PSyData API to write kernel input- and output-arguments to a file.

When an extraction transformation is applied, it will determine the input- and output-variable for the code region using the dependency analysis (see *Variable Accesses*). The code created by PSyclone's PSyData transformation will then call the PSyData extraction library as follows:

- For each input variable the variable with its original name will be stored in the file before the kernel is called.
- For each output variable the variable will be stored with in the file with the postfix `_post` added after the kernel has been executed.

As example, an input- and output-variable `my_field` will therefore have two values stored in the file: the input value using the name `my_field` and the output value using `my_field_post`. If the variable is a member of a structure, the key-name of the variable to be written to the file will contain the `%`, e.g. `my_field%whole%xstart`. It is therefore important that the output format supports special characters (e.g. NetCDF does allow the use of `%` in names).

Note: If a name clash is detected, e.g. the user has a variable called `my_field` and another variable called `my_field_post`, the output postfix `_post` will be changed to make sure unique names are created by appending a number to the postfix, e.g. `_post0`, `_post1`. The same postfix will be applied to all variables, not only to variables that have a name clash.

An excerpt of the created code (based on `examples/gocean/eg5/extract`):

```
CALL extract_psy_dataPreStart("main", "update", 11, 5)
...
CALL extract_psy_dataPreDeclareVariable("a_fld", a_fld)
CALL extract_psy_dataPreDeclareVariable("a_fld_post", a_fld)
CALL extract_psy_dataPreEndDeclaration
CALL extract_psy_dataProvideVariable("a_fld", a_fld)
...
CALL extract_psy_dataPreEnd
! Kernel execution
...
CALL extract_psy_dataPostStart
CALL extract_psy_dataProvideVariable("a_fld_post", a_fld)
...
```

The variable `a_fld` is declared twice, once with its unmodified name representing the input value, and once as `a_fld_post`, which will be used to store the output value. The values are provided once before the kernel to allow the PSyData library to capture the input values, and once using the name `a_fld_post` after the kernel execution to store the results.

Note: The following section on driver creation applies at this stage only to GOcean. Support for driver creation for LFRic is tracked in issue #1392.

The creation of a stand-alone driver can be requested when applying the extraction transformation, e.g.:

```
extract = GOceanExtractTrans()
extract.apply(schedule.children, {"create_driver": True})
```

When compiled and executed, this driver will read in the values of all input- and output-variables, execute the instrumented code region, and then compare the results of the output variables (see NetCDF Extraction Examples). This program does not depend on any infrastructure library (like 'dl_esm_inf`'), it only needs the PSyData wrapper library (e.g. `lib/extract/netcdf/dl_esm_inf`), plus any libraries the wrapper depends on (e.g. NetCDF).

The following changes are applied by the `ExtractionDriverCreator` in order to generate stand-alone code for GOcean:

1. The *dl_esm_inf* field type is replaced with 2d Fortran arrays. The structure name used is 'flattened', i.e. each `%` is replaced with a `_` to make a standard Fortran name, but the final `%data` is removed. So `my_field%data` becomes `my_field`.
2. Any other structure access is converted into a variable with the same intrinsic type and flattened name. E.g. `my_field%whole%xstart` becomes the variable `my_field_whole_xstart`.
3. For each input-only variable one variable (with a potentially flattened name) is created. It calls `ReadVariable` from the PSyData extraction library, which will allocate the variable if it is an array.
4. For each input+output variable two variables are created and initialised (especially allocated if the variable is an array) with a call to `ReadVariable`. The output variable will have a postfix appended (`_post` by default) to

distinguish it from the input value. The value of the input array will be provided to the kernel call. At the end, the newly computed values in the variable will be compared with the corresponding `_post` variable, which was read from the file.

5. An output only variable will be declared with the postfix attached, and allocated and initialised in `ReadData`. Then the variable without postfix will also be declared, and explicitly allocated to have the same shape as the output variable. This variable will be initialised to 0, and then provided to the kernel call. Again, at the end of the call these two variables should have the same value.

Here an example showing some of the driver code created:

```
integer :: a_fld_whole_ystart

real*8, allocatable, dimension(:,:) :: a_fld
real*8, allocatable, dimension(:,:) :: a_fld_post
real*8, allocatable, dimension(:,:) :: b_fld
real*8, allocatable, dimension(:,:) :: b_fld_post

call extract_psy_data%OpenRead('main', 'update')
call extract_psy_data%ReadVariable('a_fld', a_fld)
call extract_psy_data%ReadVariable('a_fld_post', a_fld_post)
call extract_psy_data%ReadVariable('a_fld%whole%ystart', a_fld_whole_ystart)

call extract_psy_data%ReadVariable('b_fld_post', b_fld_post)
ALLOCATE(b_fld(SIZE(b_fld_post, 1), SIZE(b_fld_post, 2)))
b_fld = 0

do j = a_fld_whole_ystart, a_fld_whole_ystop, 1
  do i = a_fld_whole_xstart, a_fld_whole_xstop, 1
      call update_field_code(i, j, a_fld, b_fld, ...)
  enddo
enddo

if (ALL(a_fld - a_fld_post == 0.0)) then
  PRINT *, "a_fld correct"
else
  PRINT *, "a_fld incorrect. Values are:"
  PRINT *, a_fld
  PRINT *, "a_fld values should be:"
  PRINT *, a_fld_post
end if
if (ALL(b_fld - b_fld_post == 0.0)) then
  PRINT *, "b_fld correct"
else
  PRINT *, "b_fld incorrect. Values are:"
  PRINT *, b_fld
  PRINT *, "b_fld values should be:"
  PRINT *, b_fld_post
end if
```

The variable `a_fld` is an input- and output-field, so both values for `a_fld` and `a_fld_post` are read in, and after the kernel execution the values are checked for correctness. The variable `b_fld` on the other hand is an output-variable only, no input values are stored in the file. The code created calls to `ReadVariable` for `b_fld_post`, which allocates the variable corresponding to the array size information stored in the data file. Then the driver allocates an array `b_fld` with the same shape as `b_fld_post`, which is initialised to 0. This `b_fld` is provided in the kernel call. After the

kernel call, `b_fld` should be equal to `b_fld_post`.

CHAPTER

EIGHTEEN

SYSTEM-SPECIFIC DEVELOPER SET-UP

Section System-specific Set-up for Users in the PSyclone User Guide describes the setup for a user of PSyclone. It includes all steps necessary to be able to use PSyclone. And while you could obviously do some development, none of the required tools for testing or documentation creation will be installed.

This section adds software that is used to develop and test PSyclone. It includes all packages for testing and creation of documentation in html and pdf. We assume you have already installed the software described in the System-specific Set-up for Users section.

It contains detailed instructions for Ubuntu 16.04.2 and OpenSUSE 42.2 - if you are working with a different Linux distribution some adjustments will be necessary.

18.1 Installing PSyclone From GitHub

For development it is recommended to get a copy of PSyclone using git to get access to the latest development version.

18.1.1 Installing git for Ubuntu

You need to install the git package:

```
> sudo apt-get install git
```

18.1.2 Installing git on OpenSUSE

You need to install the git package:

```
>> sudo zypper --no-recommends install git
```

18.1.3 Cloning PSyclone Using git

Cloning PSyclone from git and setting up your environment is done as follows:

```
> cd <PSYCLONEHOME>
> git clone --recursive https://github.com/stfc/PSyclone.git
> cd PSyclone
> pip install --user -e .
```

Note that the "-e" flag causes the project to be installed in 'editable' mode so that any changes to the PSyclone source take effect immediately. However, it also means that the PSyclone configuration file is not installed so you will have to do that manually (see the Configuration section of the User Guide).

Warning: On OpenSUSE it is necessary to add *$HOME/.local/bin* to your *$PATH* if you have done a user-local install.

18.2 Installing Documentation Tools

Install Sphinx along with bibtex support for creating PSyclone documentation:

```
> sudo pip install sphinx sphinxcontrib.bibtex
```

You can now build html documentation:

```
> cd doc
> make html
```

The latex package is required to create the pdf documentation for PSyclone. Installing the full dependencies can take up several GB, the instructions for Ubuntu and OpenSUSE only install a minimal subset.

18.2.1 Installing LaTeX on Ubuntu

The following four packages need to be installed to create the pdf documentation. It is recommended to install the packages in one `apt-get` command, since otherwise, depending on your filesystem, unnecessary snapshots might be created that take up additional space. The `--no-install-recommends` option significantly reduces the number of installed packages:

```
> sudo apt-get install --no-install-recommends texlive \
texlive-latex-extra latexmk tex-gyre
```

18.2.2 Installing LaTeX on OpenSUSE

The following command installs the minimum number of packages in order to create the pdf documentation - around 130 packages all in all, requiring approximately 300 MB.

Warning: It is important to install the packages in one `zypper` command, since otherwise, depending on your filesystem, unnecessary snapshots might be created after each package, which can add up to several GB of data.

```
> sudo zypper install --no-recommends texlive-latex texlive-latexmk \
texlive-babel-english texlive-cmap texlive-psnfss texlive-fncychap  \
texlive-fancyhdr texlive-titlesec texlive-tabulary texlive-varwidth \
texlive-framed texlive-fancyvrb texlive-float texlive-wrapfig       \
texlive-parskip texlive-upquote texlive-capt-of texlive-needspace   \
texlive-metafont texlive-makeindex texlive-times texlive-helvetic   \
texlive-courier texlive-dvips
```

18.2.3 Creating PDF Documentation

You can now build the pdf documentation using

```
> cd doc
> make latexpdf
```

18.2.4 Creating Markdown Documentation

The documentation for PSyclone's Examples and Tutorials is written in GitHub-flavoured Markdown. Although this requires no special tools to create, checking that it renders correctly means pushing your changes to GitHub and then visiting the file in GitHub's source browser. This may be avoided by using grip. Once installed, simply start *grip* in the directory containing the file you are working on and point your web browser at the URL it displays, e.g.:

```
> grip
* Serving Flask app "grip.app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://localhost:6419/ (Press CTRL+C to quit)
```

18.3 Installing Testing Tools

You can install the necessary dependencies to run the PSyclone tests with:

```
> pip install psyclone[test]
```

or when using the git version:

```
> pip install .[test]
```

The test dependencies are canonically documented in PSyclone's setup.py under the `extras_requires` section. This installs the recommended tools to get access to testing and formatting tools.

Warning: It appears that the 1.7 series of `pylint` has a bug (at least up to 1.7.2) and does not work properly with PSyclone - it aborts with a "maximum recursion depth exceeded" error message. It is therefore recommended to avoid pylint 1.7.

You can now run the PSyclone python tests:

```
> cd PSyclone.git
> pytest
```

In order to see whether the Python code conforms to the pep8 standards, use:

```
> pycodestyle code.py
```

Note: `pycodestyle` is a replacement for the older `pep8` program.

Verifying the pylint standards is done with:

```
> pylint code.py
```

OK, you're all set up.

18.4 Installing Tools for PSyData Wrapper Libraries

If you intend to compile the PSyData wrapper libraries or develop new libraries, you might need to install Jinja2 (most wrapper libraries require Jinja2 though some, like the NVIDIA GPU profiling wrapper, do not need it). You can install the necessary dependencies to create all PSyData wrapper libraries with:

```
> pip install psyclone[psydata]
```

or when using the git version:

```
> pip install .[psydata]
```

Check PSyData API and especially the section *Jinja Support in the Base Class* for more details.

CHAPTER

NINETEEN

CODING AND DOCUMENTATION STYLE

19.1 Documentation Style

When writing documentation, each reference to a PSyclone class or function should be set in italics (i.e. enclosed by single backticks) except in headings. The first time a class or function is mentioned, use the full Python path, e.g.: *psyclone.core.access_info.SingleVariableAccessInfo*. After that just use the class name (again in italics). File names and shell commands should be set in double back-ticks (``).

Any code examples provided in the documentation should be using the right syntax to allow for syntax highlighting. The option `.. highlight:: LANGUAGE` is used to define the language to be used for all text following the directive (until the end of the current file). It is recommended to have at most one `highlight` directive in a file and to avoid switching default languages several times within a file. Having a default language set in a file allows use of the very convenient shortcut notation "::". Note that the "::" will be replaced with a single ":" if the previous character is not a space:

```
.. highlight:: fortran

Code example::

    program this_is_highlighted_using_fortran_syntax

And if you don't want any ":" in the previous line, just
add a white space before the "::" ::

    program there_is_no_colon_in_the_previous_description
```

If some examples in a file need a different highlighting syntax, use the `.. code-block:: LANGUAGE` directive, which will only change the syntax for the following code example:

```
.. highlight:: fortran

Code example::

    program this_is_highlighted_using_fortran_syntax

This is a python example:

.. code-block:: python

    trans = self.get_transform()
```

It is also possible to use `code-block` inside a Python docstring that is pulled into a document:

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

    .. code-block:: fortran

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

    Some compilers are able to produce more efficient code if they are
    ...
```

The code block will be highlighted as Fortran code.

19.2 Coding Style

Any new PSyclone code must confirm to Python's pep8 specification, and must be pylint error and warning free. Installation of the tools is described in *System-specific Developer Set-up*. In some cases pylint errors and warnings can be suppressed using the pylint markup code:

```
# pylint: disable=too-many-branches
def _upper_bound(self):
    ....
```

It is up to the developer to decide if code should be refactored to avoid a warning. For example, a warning about too many local variables in a function might be better suppressed instead of removing one variable, or refactoring the code to create two functions.

Additional rules that apply:

1) All setter and getter functions (that do not do any significant work) must be declared as property and setter to allow them to be used without parentheses:

```
@property
def ast_end(self):
    '''
    :returns: a reference to the last node in the fparser2 parse tree \
              that represents a child of this PSyIR node or None.
    :rtype: sub-class of :py:class:`fparser.two.utils.Base`
    '''
    return self._ast_end

@ast.setter
def ast(self, ast):
    '''
    Set a reference to the fparser2 node associated with this Node.
           :param ast: fparser2 node associated with this Node.
    :type ast: :py:class:`fparser.two.utils.Base`
    '''
    self._ast = ast
```

2) Any function must contain an interface description, see *Interface Description* for full details..

3) Any new line of code must be covered by at least one test case, see *Test Suite* and especially *Coverage*.

4) Importing other modules should be done at the top of a file, and the import statements must be sorted alphabetically. If a module cannot be import at the top of a file due to a circular dependency, a comment must be added to the local import statement, and the corresponding pylint warning must be disabled, e.g.:

```
def my_function():
    ...
    # Avoid circular import
    # pylint: disable=import-outside-toplevel
    from psyclone.core.access_type import AccessType
```

19.2.1 Exceptions

When raising an exception, every effort must be made to ensure that the associated message is clear and provides as much information as possible. However, it should *not* contain the name of the current routine/class etc. as this is provided in the Python stack trace. Where it makes sense to do so, the past tense should be used, e.g. "expected a str but got an object of type 'blah'."

In the event that code that is handling an exception then needs to raise a new exception, the *raise XXX from YYY* form must be used. (This ensures that contextual information about the source of the error is retained.). For example:

```
try:
    something()
except KeyError as err:
    raise InternalError("Useful message here") from err
```

19.3 Interface Description

The interface to any new or modified routine in PSyclone must be fully documented using Sphinx mark-up. An example of how to do this is shown below:

```
def some_function(filename, kernel_path, node=None):
    "The description starts with a capital letter and must have
    proper punctuation. Use for example :func:`parse.algorithm.parse`
    to reference functions, or :py:class:`psyclone.psyir.nodes.Node` for
    references to other PSyclone classes. The description must be followed
    by an empty line before the parameters start, but it is not necessary
    to escape each new line with a backslash here.

    :param str filename: start lower case, but add full stop. This\
                         line also shows the type declarations part of\
                         the parameter declaration, which can be used for\
                         any standard Python data type like str, bool, \
                         int, float.
    :param str kernel_path: no empty line between different parameters.\
                            If you need more than one line, add a backslash\
                            to the end of each line, otherwise sphinx\
                            will not layout the text correctly.
    :param node: the parameter type can also be declared in a stand-alone\
                 line which follows immediately after the corresponding\
```

(continues on next page)

(continued from previous page)

```
                :param: line. The actual type should only contain the\
                type, no filler words like 'return type is integer'.\
                Type information should be specified according to PEP 483\
                (https://peps.python.org/pep-0483/).\
                Notice the empty line between parameter and return\
                documentation.
    :type node: Optional[:py:class:`psyclone.psyir.nodes.Node`]

    :returns: a new node in the PSyIR. The return type must always be\
              specified in a separate line with an :rtype: entry. An empty\
              line separates the return documentation and the exceptions.
    :rtype: :py:class:`psyclone.psyir.nodes.Node`

    :raises IOError: lower case start with punctuation at the end.
    :raises GenerationError: list the same exception more than once if\
                             it can be raised by different errors.
    :raises GenerationError: same exception, raised by a different error.

    For example:

    >>> from psyclone.generator import generate
    >>> API="gocean1.0"
    >>> alg, psy = generate(SOURCE_FILE, api=API)
    >>> alg, psy = generate(SOURCE_FILE, api=API, kernel_paths=[KERNEL_PATH])

    '''
```

Example layout of the interface description above:

`interface_example.`**`some_function`**(*filename*, *kernel_path*, *node=None*)

The description starts with a capital letter and must have proper punctuation. Use for example `parse.algorithm.parse()` to reference functions, or `psyclone.psyir.nodes.Node` for references to other PSyclone classes. The description must be followed by an empty line before the parameters start, but it is not necessary to escape each new line with a backslash here.

Parameters

- **`filename`** (*str*) – start lower case, but add full stop. This line also shows the type declarations part of the parameter declaration, which can be used for any standard Python data type like str, bool, int, float.
- **`kernel_path`** (*str*) – no empty line between different parameters. If you need more than one line, add a backslash to the end of each line, otherwise sphinx will not layout the text correctly.
- **`node`** (Optional[`psyclone.psyir.nodes.Node`]) – the parameter type can also be declared in a stand-alone line which follows immediately after the corresponding :param: line. The actual type should only contain the type, no filler words like 'return type is integer'. Type information should be specified according to PEP 483 (https://peps.python.org/pep-0483/). Notice the empty line between parameter and return documentation.

Returns

a new node in the PSyIR. The return type must always be specified in a separate line with an :rtype: entry. An empty line separates the return documentation and the exceptions.

Return type
psyclone.psyir.nodes.Node

Raises

- **IOError** – lower case start with punctuation at the end.
- **GenerationError** – list the same exception more than once if it can be raised by different errors.
- **GenerationError** – same exception, raised by a different error.

For example:

```
>>> from psyclone.generator import generate
>>> API="gocean1.0"
>>> alg, psy = generate(SOURCE_FILE, api=API)
>>> alg, psy = generate(SOURCE_FILE, api=API, kernel_paths=[KERNEL_PATH])
```

Some important details:

1) There are up to four major sections in each interface description: function description, parameter description and type, return value and type, and raises (exceptions). The function description is required, all other sections only need be provided if they are applicable for the code being documented. The formatting for each section is slightly different:

Section	Formatting
function description	The description of the function must start with a capital letter, and end with a full stop.
parameter description	Start the parameter description with a lowercase letter and end with a full stop. The parameter type declaration must follow PEP 483 with no punctuation at the end. References to other classes within PSyclone should be written as :py:class:`psyclone.filename.Class`.
return value	The description of the return value should start with a lowercase letter, and end with a full stop. The type must follow PEP 483.
exceptions	These must start with a lower case letter, and end with a full stop.

2) If a parameter description, type, return value or exception is continued to the next line, there must be a '\\' continuation symbol at the end of each line. Align each continued line with the column at which the description begins on the previous line. If this would create lines that are too short then the first continued line may be indented less, e.g.:

```
'''
:param some_very_long_variable_name: this is some argument that has \
       a very long name and therefore it does not make sense to indent \
       continued lines to align with the start of the description.
'''
```

3) If an argument type is a Python built-in (e.g. str, int or bool) then the type can be specified in-line with the argument description. However, if it is of a derived type then, for clarity, it should be specified in a separate :type my_arg: line.

4) The closing \'\'\' of the interface description can be at the end of a text line if the overall description is short. Otherwise it should be on a separate line. An optional empty line between interface description and code should be included in the comment section.

5) Standard Python functions like *__str__* etc. need only be documented with a simple informal comment.

6) Only document exceptions that are raised directly by a method, not exceptions that might be raised in base classes.

19.4 File Names and Directory Layout

Any file in PSyclone should only contain one main class (helper classes or functions specific to that class are of course possible). While the class name should start with a capital letter and be in camel-case (*ExtractTrans*), the corresponding file name should be derived from the class name by replacing all upper case letters with lower case, and adding a '_' to separate words. So the file containing the class *ExtractTrans* should be called *extract_trans.py*.

The directory structure of the PSyclone classes is as follows:

domain:
This directory contains the various API-specific classes.

domain/API_NAME:
The following domains are currently supported: *lfric*, *gocean*, *nemo*.

domain/API_NAME/transformations:
These directories contain the API-specific transformations, typically using one of the classes in psyir/transformations as a base class. Any transformation class should have the domain as prefix, and *Trans* as postfix (e.g. *GOceanExtractTrans*), and the corresponding file name should start with the API name and end with *_trans.py* (e.g. *gocean_extract_trans.py*).

psyir:
This directory contains all classes and functions related to the PSyIR (PSyclone Internal Representation). The directory itself does not contain any source files (except *__init__.py* to shorten the import paths).

psyir/transformations
This directory contains all basic transformations, i.e. all transformations that are either directly usable in any API, or are base classes for API-specific transformations. Any transformation class should have *Trans* as postfix (e.g. *ExtractTrans*), and the corresponding file name should end with *_trans.py* (e.g. *extract_trans.py*).

BIBLIOGRAPHY

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

[ope17] *The OpenACC Application Programming Interface, Version 2.6*. 2017. URL: http://www.openacc.org.

[ope18] *The OpenCL 2.2 Reference Guide*. 2018. URL: https://www.khronos.org/files/opencl22-reference-guide.pdf.

INDEX

Symbols

A

C

D

E

F

G

H

I

K

L