IMOP : a source-to-source compiler framework for OpenMP C programs

PART B : Code Review Document

AMAN NOUGRAHIYA, Indian Institute of Technology Madras
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras

This code-review document provides a class-by-class walk-through of the code for some of the most important and elementary portions of IMOP. It has been written with an aim to help the users of IMOP understand the intent of each method and class of importance, so that they can use, modify, and extend them with ease.

This document is, by no means, exhaustive in nature. The code of IMOP is ever-evolving. Periodically, while attempts are made to keep this document up-to-date, many portions of it might not reflect the current state of the code, and many new portions of the code might not have any corresponding review section in this document.

It is suggested to read this document alongside the code, mainly because it liberally refers to various identifiers (class, field, method or variable names) present in the code, while discussing the concepts denoted by them. In order to read the higher-level abstractions of any concept, kindly refer to the preliminary technical report of IMOP.

Contents

This code-review document provides a class-by-class walk-through of the code for some of the most important and elementary portions of IMOP. It has been written with an aim to help the users of IMOP understand the intent of each method and class of importance, so that they can use, modify, and extend them with ease.

Note 0.1

This code-review document is, by no means, exhaustive in nature. The code of IMOP is ever-evolving. Periodically, while attempts are made to keep this document up-to-date, many portions of it might not reflect the current state of the code, and many new portions of the code might not have any corresponding review section in this document.

1 INITIALIZATION

Program.parseNormalizeInput(args), the usual starting point for the framework, takes the string of command-line arguments as input, and performs the parsing and *normalization* steps on the input program. Certain points to note about this method (along with the steps that it performs) :

- First of all, this method sets the default values for various global flags via a call to the method **Program.defaultCommandLineArguments()**. Following are various flags and globals that are set by this method :
  - Program.isPrePassPhase, which is used to indicate whether this call of the framework is done in the *pre-pass* mode or not. The default value of this flag is true.
  - Program.proceedBeyond, which is used to indicate whether this call of the framework should exit after the pre-pass phase, or proceed beyond it. The default value of this flag is true.
  - Program.removeUnused, which is used to indicate whether the unused symbols, types, etc., are removed by the pre-pass or not. The default value of this flag is true.
  - Program.sveSensitive, and Program.fieldSensitive, which specify various analysis dimensions, as their names suggest. By default, both these dimensions are kept enabled.
  - Program.enableUnmodifiability is, by default, kept disabled. When enabled, various read-only sets are returned to the user code as unmodifiable-sets.
  - Program.maxThreads is, by default, set to 13 – this number is assumed to be the maximum number of threads with which the input program will be executed. This value is used to bound the state-space exploration in the Z3 solver.
  - Program.z3TimeoutInMilliSecs is kept to 5000 milliseconds – this is the maximum amount of time that is given to the solver for each query.

  - Program.oneEntryPerFDInCallStack, when set, ensures that corresponding to each function-definition, there can be only one entry in a context-sensitive call-stack. This flag is, by default, kept as enabled.
  - ***UPDATE:*** **"Program.prceiseDFDEdges, when set, ensures that an inter-task edge exists between two DummyFlushDirectives only if a communication may happen across them using at least one shared variable; when this flag is not set, the edges would exist between all pairs of DummyFlushDirectives that may share a phase."**
  - Apart from setting these flags, this method is also used to obtain the relative path of the .i input file, which is returned via the value of filePath variable.
- Then, this method overrides the default values as per the provided command line switches. Following is a list of command-line switches that are currently available in IMOP : (*Note : For most of the switches their corresponding negations are available as well.*)
  * --prepass, or -p : when used, it enables the pre-pass mode, and the call exits after the pre-pass phase. (Negation : --noPrepass.)
  * --file, or -f : used to specify that the immediately succeeding string is the relative (or absolute) file path for the input program.
  * --removeUnused, or -ru : sets Program.removeUnused, to specify that unused symbols and types get removed. (Negation : --noRemoveUnused.)
  * --sveSensitive, or -sve : enables SVE sensitivity, by setting the flag Program.sveSensitive. (Negation : --noSveSensitive.)
  * --fieldSensitive, or -fs : enables field-sensitivity, by setting the flag Program.fieldSensitive. (Negation : --noFieldSensitive.)
- Then, this method saves the name of the input file (without file extension) in Program.fileName.
- Next, this method makes a call to FrontEnd.parseAndNormalize() (discussed below), after ensuring that the standard input stream is set to the specified file. This call may or may not return, depending upon the –prepass and –proceedBeyond switches.
- Finally, this method prints the output program into the file with -postpass.i suffix, after pre-pass is complete, and returns. For cases where we just need to use/inspect the pre-pass code, we can simply make a call to this method, and exit from the framework.

FrontEnd.parseAndNormalize(String|InputStream) is used to perform the actual pre-pass of the input. Various normalization steps have already been discussed in the preliminary technical report of IMOP. We discuss some other observations from the code in next few sections.

2 PARSING

For parsing a code versus a snippet, two different (overloaded) methods are used, with name **parseAndNormalize()** in FrontEnd. While parsing a full program, rooted at a node of type TranslationUnit, following are the key observations :

- The variable Program.root is automatically set to the root of the newly parsed AST.
- Depending upon the type of a node being parsed, following extra steps are carried out, apart from construction of its subtree :
  - In the constructor of FunctionDefinition, at the end, a call is made to the visitor OldFunctionStyleRemover, which is used to remove the old style of function declaration.
  - In order to obtain an ExpressionStatement, we need to call its factory methods. With the help of a static method CallStatement.getCallIfAny(Expression):Statement, this factory method will return a CallStatement if this expression directly represents a CallStatement lexically; otherwise, this method returns an ExpressionStatement,

Note 2.0.1

Note that the method CallStatement.getCallIfAny(Expression) would return null if the argument does not lexically represent a simplified call-statement; otherwise, the corresponding CallStatement object is created and returned back. Also note that this method does **not** simplify any expression which contains function call(s) somewhere within it.

  - The constructors of EnumSpecifier and StructOrUnionSpecifier call the visitor StructUnionEnumTagger, which ensures that unnamed structs/unions/enums are provided with a unique name.

Note 2.0.2

Note that certain internal nodes of struct/union/enum will get parsed again if it is unnamed.

  - When a Statement is getting constructed, if it wraps a LabeledStatement, the wrapping is done for the constituent of the LabeledStatement instead (while pushing label as an annotation on that constituent.) Note that this also happens whenever the constituent node is changed to a LabeledStatement. This processing is done via a call to LabelRemover.populateLabelAnnotations(Statement).
  - In the constructor of OmpConstruct, we plan to (haven't yet) implement a call to splitParForAnSections which can split the combined OpenMP constructs. This code would be extracted from ExpressionSimplifier. (We should modularize and reuse that portion of code.)
  - In parsing of a Node (i.e., all AST nodes), each node is given a unique integral id. If the parsed node is a leaf node, then it is stored in a static array, allLeafCFGNodes of Node.

Also note that a back-pointer, named parent is automatically maintained in the constructor of each node, or while modifying any of its children.

- If the pre-pass flag is enabled, this method calls **FrontEnd.prePass()**; certain points to note about the same:
  - This method calls expression simplification pass (ExpressionSimplifier) on the parsed program, to regenerate a new AST with simplifications performed on the input. The output is dumped in a file with -simplified.i extension.
  - This is followed by creation of CFG edges in the newly constructed, simplified AST, using CFGGenerator.createCFGEdgesIn(Node). The CFG is dumped with an extension of -nestedDotGraph.gv, in DOT format.
  - Then, with the help of CompoundStatementEnforcer pass, it is ensured that for all constructs of the program, non-CS single-body statements are wrapped within a compound statement. The output file contains the extension -enforcer.i.
  - Next, a call is made to remove implicit barriers using nowait clause, with the help of ImplicitBarrierRemover.removeImplicitBarrier(Node). The output file has an extension of -explicitBarriers.i.
  - Extra scoping, if any, is removed via a call to NodeInfo::removeExtraScopes(). Output file contains an extension of -scoped.i.
  - If the flag –removeUnused is set, declarations of unnecessary elements are removed by a call to NodeInfo::removeUnusedElements(). The file corresponding to the code with unnecessary declarations removed, has an extension of -useful.i.

Note 2.0.3

Note that if the Program.proceedBeyond flag is disabled, the framework will exit at the end of the method FrontEnd.prePass().

- CFG edges are created for the newly parsed AST. Note that since pre-pass steps might not trigger automated update of CFG edges, it is important to recreate the edges here, wherever required. This step overwrites the previously generated DOT file, which represents the CFG of the input program.
- Then, various parallelism related steps, executed by a call to **FrontEnd.processParallelism()**, whenever the translation unit contains a main() function, are as follows :
  - DummyFlushDirectives are inserted at required places (as per the rules discussed in the preliminary technical report of IMOP), in all the compound statements of the program, using CompoundStatementCFGInfo::initializeDummyFlushes().
  - Then, initial MHP analysis is run, using Misc.performMHPAnalysis(Node).

  - Relying on the MHP information, this method then calls Misc.createDataFlowGraph() to create inter-task data-flow edges among various DummyFlushDirectives within every phase.
  - Note that the lock-set analysis (LockSetAnalysis) has been disabled for now.
- If field-sensitivity is enabled, this method invokes FrontEnd.testIncompatibleTypeCasts() which uses Type.hasIncompatibleTypeCastOfPointers() to test whether there are any incompatible type casts of pointers anywhere in the program. If there are, then field-sensitivity gets disabled.
- As of now, we have enabled the call to only points-to data flow analysis (PointsToAnalysis()) in this method. Note that the IDFAs are triggered only if main() function is present in the translation unit.

While parsing an AST that is rooted at any type of node other than a TranslationUnit, following steps are performed by the method parseAndNormalize(InputStream, Class<T>) :

- The initial AST gets created, as in the case of parsing of a TranslationUnit. Note that all extra steps that are carried out in the constructors of various specific types of nodes, as mentioned in the case of parsing of a TranslationUnit above, are performed even here.
- As of now, we do not call simplification pass on this newly constructed tree; we need to handle this issue. Until then, users must ensure that they do not create any AST that contains anything that a simplified AST cannot.
- As before, this step is followed by,: (i) creation of CFG edges, (ii) enforcing of compound statements across non-CS single-statement bodies, (iii) removal of implicit barriers, (iv) removal of extra scoping, (***UPDATE:*** **"Disabled now, until next iteration of the module."**) (v) addition of DummyFlushDirectives at required places, (vi) an initial MHP analysis on all ParallelConstruct nodes within the newly created node, using MHPAnalyzer.initMHP(), (vii) creation of inter-task edges, and (viii) testing whether there is any incompatible type-casting of pointers. Note that after initial MHP run is complete, we remember all the current phases of each CFG node in the field NodePhaseInfo::inputPhaseSet:HashSet<Phase>, using the method NodePhaseInfo::rememberCurrentPhases().

```
/* To be tested: Are we able to correctly parse all the benchmarks under review?
*/
```

3 OLD-STYLE FUNCTION-DECLARATION REMOVAL

This visitor visits each FunctionDefinition of the program, and ensures the following :

- If there is no return type specified with the function, this method adds int as the return type.
- If the function's declarator has old-style declaration, a new declarator is created (via AST construction on manipulated string) and used in place of the old declarator. In other words, the declarator corresponding to the function-name gets parsed again. Various methods utilized

during creation of the manipulated string are : Misc.getDeclarator(Declaration, String), and Misc.getIdNameList(Declaration).

4 EXPRESSION SIMPLIFICATION

We have already listed various expression simplifications that this pass performs in the preliminary technical report of IMOP. In this section, we look into other implementation-level details.

- First of all, we should note that the current implementation is terribly slow since it is string-based. We need to port the existing logic into a new pass which performs least number of changes in the given AST to perform expression simplification. (We are planning to do this task later.)
- Meanwhile, apart from the simplifications discussed in the preliminary technical report of IMOP, we do not have much to discuss here.

5 LABEL ANNOTATIONS

In the setter of field stmtF0 of any Statement node, whenever the node to be added is a LabeledStatement, we call **LabelRemover.populateLabelAnnotations()** on that Statement. Some key points/steps to observe concerning this method :

- Note that a statement may have more than one labels attached to it. Hence, we process even nested occurrences of LabeledStatement while attempting to store labels as annotations.
- For each labeled-statement of one of the three types (simple, case, or default), first of all, this method creates a corresponding label annotation of type Label. Then, on the CFG node on which this label will be annotated, this method calls **StatementInfo::initAddLabelAnnotation(Label)**, which adds the label to appropriate field in the node, and vice versa.
- After creating all the internal label annotations and attaching them with the appropriate CFG nodes, this method calls **LabelDeleter**. This visitor replaces each LabeledStatement in the AST with the internal CFG node on which the corresponding Label has been annotated. Note that this change is performed at the AST level, and is not done via any elementary transformation.

6 GETTING AST STRINGS

The class **StringGetter**, and its supporting static inner classes, provide methods to obtain different variations of strings corresponding to any given AST node.

In this section, we first look at how comments can be added and used for various Nodes. Then, we discuss other key methods in StringGetter.

Note 6.0.1

If a simple String representing a Node is required, along with its default comments, then the method NodeInfo::toString() should suffice.
For other specific use cases, read this section (Section 6) in detail.

6.1 Commentors

A **Commentor** is a functional interface, which provides a method getString() that takes a Node, and provides a String which is relevant to the node. This interface is implemented at a large number of places for debugging purposes, where it is used to specify debug strings for various kinds of node.

Each NodeInfo contains a default Commentor, named defaultCommentor which is used in generating the default comments corresponding to the node. These default comments are read from the field comments:List<String>, to which debug messages can be added by any client code (by adding new Strings to the return of NodeInfo::getComments()). The respective default comments

Note 6.1.1

Note that any string can be added to NodeInfo::getComments(). It need not follow the syntax of C comments.

are added as suffix (with proper C syntax of comments) to strings of all the nodes which are obtained using any variation of StringGetter.getString().

6.2 String getters.

Following are some key methods from StringGetter :

getString(). Various overloaded versions of getString() exist within the class StringGetter. When it takes only a Node, it internally calls the version that takes a Node and a boolean indicating whether annotated pragma's [1], if any, need to be printed, by passing the same node and `false` value to it. The invoked method uses the visitor InternalStringGetter (explained later in this section) with two arguments : (i) a list containing the defaultCommentor, and (ii) pragma boolean as the arguments. The string to be returned gets populated in the field InternalStringGetter::str, from which extra white-spaces are removed before it is returned.

The variant of getString() which takes a Node, and a list of Commentor, passes its argument to another variant that also takes a boolean value for indicating whether annotated pragma's need to be printed. Similar to one of the variants from the previous paragraph, the invoked variant utilizes the visitor InternalStringGetter to obtain the string to be returned. The list

[1]Note that in the current state of IMOP, user-defined pragma annotations are not fully implemented. The only ones that exist are for specifying SVE annotations.

of Commentors obtained as arguments is sent by this method to the visitor, after adding the defaultCommentor to it, if not already present.

Note that the following key methods rely on StringGetter.getString(), and hence, will have respective default comments added to each printed node in the returned strings :

- **Node::toString()**.

Note 6.2.1

As well known, the method toString() is implicitly called on a Java object when the object is passed as an argument to the print() or related methods of System.out and System.err. Hence, System.out.print(node);, for example, would print the node, preceded by its default comments, to `stdout`. For each constituent CFG node of the given node, the default comments, if any, of the constituent node would also be printed before the constituent node.

- **NodeInfo::printNode()**, which prints the node, preceded by its default comments to `stdout`. Respective default comments are also printed for any constituent CFG nodes.
- **NodeInfo::getString()** is used to obtain a string that represents the node, with string for each constituent CFG node (including the receiver node) preceded by its default comment. This method optionally takes a list of Commentors which are used to append other comments to the strings of the default comments.
- **NodeInfo::getDebugString()** is a special variation of string getters, which can be used as a *Detail Formatter* for Nodes in the Debug mode in Eclipse, or at any other places where a node needs to be identified in the context in which it appears (along with its phase information). Given a node, this method returns a String of its enclosing function, (or of the node itself, if no enclosing function exists, or if the node is not a CFG node) with a string of #'s preceding the node to highlight it. Each constituent leaf CFG node is also preceded with a list of phase id's in which that node may get executed. [2]
- Another relevant method is **Misc.printToFile()**, which takes a node, name of the file to be generated (along with proper extension), and optionally a list of Commentors. It creates the file in directory `output-dump` in the root of the project, and pretty prints the given node, with its default comments and any other comments provided by the argument list of Commentors, if any.

The visitor **InternalStringGetter** works as follows :

- This visitor takes a list of Commentors as an argument, along with a boolean that indicates whether annotated pragma's of any node need to be printed.
- The string created by this visitor is stored in its field str:StringBuilder, which is initialized to an empty StringBuilder.

[2]Note that NodeInfo::getDebugString() does *not* trigger automatic stabilization of the phase information.

- This visitor creates a string which contains proper tabs and newlines to pretty print the visited node.
- Visit of every CFG node also invokes the method InternalStringGetter::printCommentorsAndPragmas(), which appends str with the string obtained from various Commentors (and annotated pragmas, if any) provided to the visitor, in the form of multi-line comments.
- All labels, which are present as metadata of form Label in various statement CFG nodes, are prepended to the string of the relevant statements, with the help of calls to InternalStringGetter::printLabels() from appropriate places.

getStringNoLabel(). This method is invoked when a node's string is required without any labels in it. Such situation occurs, for example, when attempting to create a duplicate of the given node, which needs to be added within the same function (or switch statement), as having two statements with same label in a function (or at the same level of a switch statement) would be incorrect.

The required effect of this method is achieved by using a subclass InternalStringNoLabel of InternalStringGetter which simply overrides the method printLabels() with an empty body.

Assuming that the string obtained from this method is required for the purpose of some internal processing (such as parsing a duplicate), we do not print the default commentors or pragma's of any of the printed nodes.

getRenamedString(). Given a node, and a map over the set of strings, which maps the string of an identifier to some new string, this method returns the string of the node modified in such a way that each occurrence of the identifier is replaced by the new string for that identifier as per the map, if any.

Note that this is not same as simple substring replacement as some other non-identifier parts of the given node too may have same string as that of an identifier – in such cases, we do not replace that occurrence of the string with the new string from the map, if any, unlike what a substring replacement method would do.

This method relies upon the visitor RenamedStringGetter, which extends from InternalStringGetter, and works as follows : It takes a map renamingMap:Set<String, String> as an argument. Now, while creating the string to be returned, it replaces all nodes of type NodeToken that represents an IDENTIFIER lexeme, with the associated mapping for that identifier's string in the given map, if any.

Note that in any of the internal scopes within the given node, the identifier that exists with a given name, would shadow the identifier from the outer scope. Hence, in such cases, the mapping corresponding to the identifier from the outer scope should not be used. To ensure

this, before entering the body of any FunctionDefinition or CompoundStatement, this visitor removes all those keys from the given map that may have same string as the string of any of the symbols defined in the scope of the visited FunctionDefinition or CompoundStatement. After coming out of the visit of such scopes, all the removed entries are added back.

getNodeReplacedString(). This method does not rely on any visitors within the class StringGetter. Given a base node, and any of its constituent nodes that needs to be replaced with a given string this method utilizes BasicTransform.crudeReplaceOldWithNew() to obtain the desired strong for the base node, where the string of the constituent node is replaced with the given string. Note that the given string must get successfully parsed as the type to which the constituent node belongs.

7 CFG CREATION

CFG edges are created within a node via a call to **CFGGenerator.createCFGEdgesIn(Node)**. ~~After creation of complete edges, where both, source and destination nodes of the edge are available, this method generates incomplete edges. For such edges, either source or destination is not available. For example, if the provided node contains within it a case or default labelled statement such that there is no enclosing switch statement to which these labels can bind to, then we maintain an incomplete edge with unknown source for each of these statements.~~ Both these steps are explained in detail next.

7.1 Creating complete edges

Corresponding to each non-leaf node, an internal CFG node is created as per the semantics of the associated construct in C language. Each non-leaf node is handled in its own visit() method in CFGGenerationVisitor. Furthermore, extra edges are created as per the four jump statements of C.

In this section, we describe the methods that create these edges. Note that these methods do not create any inter-function or inter-task edges. Following is a quick (obvious) description of the graph structure, corresponding to each non-leaf node, and jump statement :

FunctionDefinition. The BeginNode of the function-definition is connected to the first parameter, if any. Otherwise, it is connected to the function body. If the function contains any parameter, then each parameter connects to its next parameter, if any; the last parameter is connected to the function body. This visitor is then called on the function's body, to create the nested CFG. If end of the function body may be *reachable* for any control-flow (checked using the method CFGInfo::isEndReachable()), then the function body is connected to the EndNode of the function. Note that an AssertionException is thrown if this function-definition has old-style of function signature.

ParallelConstruct. First of all, all the executable OpenMP clauses are collected for the given parallel-construct. These include : IfClause, and NumThreadsClause. The order (and number of times) in which these clauses need to be evaluated is implementation specific. If there are

Note 7.1.1

In IMOP, we assume that all the executable clauses are executed once, in the order of their appearance.

no clauses present, then the BeginNode of the parallel-construct is connected to the body of the parallel construct. Otherwise, the BeginNode is connected to the first clause; each clause is connected to the next clause, and the last clause is connected to the body of the parallel construct. This visitor is then called on the body of this parallel construct, to create the nested CFG. If the end of the body is reachable, then the body is connected to the EndNode of the parallel construct.

ForConstruct. The BeginNode of the for-construct is connected to the initialization term, OmpForInitExpression, which, in turn, is connected to the termination check expression, OmpForCondition. This expression condition is connected to the EndNode of this for-construct, as well as to its body. This visitor is then called on the body of this for construct, to create the nested CFG. If the end of the body of this for-construct is reachable, then it is connected to the step change expression, OmpForReinitExpression. Either way, the step-change expression is connected to the termination-check expression.

SectionsConstruct. For each OpenMP section in this construct, the BeginNode of the construct is connected to the body (CompoundStatement) of the section. After calling the visitor on the body of a section, it is checked if the end of the body is reachable. If the end is reachable, the body is connected to the EndNode of the construct. Note that if the construct contains no OpenMP section, then the BeginNode is connected to the EndNode.

TaskConstruct. First of all, we collect all the executable clauses that may be present in the construct. These clauses can be IfClause or FinalClause. If there are no executable clauses, the BeginNode of the construct is connected to its body. Otherwise, the BeginNode is connected to the first executable clause; each clause connects to its next one; and, the last clause connects to the body of this construct. This visitor is then called on the body of this construct to create the nested CFG. If the end of this body is reachable, the body is connected to the EndNode of this construct.

Other OpenMP constructs. For other OpenMP constructs, namely, SingleConstruct, MasterConstruct, CriticalConstruct, AtomicConstruct, and OrderedConstruct, the CFG structure is created in a similar fashion. First of all, the visitor is called on the body of the construct. After

Note 7.1.2

Note that this simplistic view of TaskConstruct doesn't help us model the flow semantics correctly. However, in the current state of IMOP, we do not handle the flow analyses for programs that contain TaskConstruct. We do have a scheme that can be implemented to change the edges of the CFG such that a TaskConstruct can be naturally modelled in the flow analyses. However, that scheme would break the invariants concerning the number of successors and predecessors certain CFG nodes may have – these invariants are used in all five higher-level CFG transformation modules. Hence, we will have to make changes in all five of these, upon implementing the scheme. This task has been added as a TODO.

that, the BeginNode of the construct is connected to its body. Then, if the end of the body is reachable, the body is connected to the EndNode of the construct.

CompoundStatement. If there are no statements in this construct, the BeginNode of the construct is connected to its EndNode. Otherwise, the BeginNode is connected to the first statement or declaration within the construct. For each element of the construct, we call the visitor on the element to create the nested CFG. If the end of an element is reachable, then the element is connected to its immediately succeeding element, except for the last element, which is connected to the EndNode.

IfStatement. The BeginNode of the construct is connected to its predicate (an Expression). The predicate is connected to the then-body. The visitor is called on the then-body to create the nested CFG. If the end of the then-body is reachable, it is connected to the EndNode of the construct. If the *else* part is present in the construct, the predicate is also connected to the else-body; the visitor is called on the else-body; and if the else-body is reachable, then it is connected to the EndNode. Otherwise, if the *else* part is not present, the predicate is connected to the EndNode of the construct.

SwitchStatement. The BeginNode of the construct is connected to its predicate. This visitor is called on the body of the construct, to create the nested CFG. If the end is reachable, the body is connected to the EndNode of the construct. Then, we collect all those nodes that are annotated to those case and default labels that are associated with this construct. The predicate is connected to all these nodes. If there is no associated default label, the predicate gets connected to the EndNode.

WhileStatement. The BeginNode is connected to the predicate of the construct. The predicate is connected to the body of the construct, and to the EndNode of the construct. The visitor is called on the body of the construct, to create CFG for nodes nested within the body. If the end of the body is reachable, the body is connected to the predicate of the construct.

DoStatement. The BeginNode connects to the body of the construct. The visitor is called on the body. If the end of the body is reachable, it is connected to the predicate. The predicate is connected back to the body, as well as to the EndNode of the construct.

ForStatement The visitor is called on the body of the construct, to create the nested CFG. The BeginNode of the construct is connected to either the initialization expression, termination expression, or the body of the construct, whichever is present (checked in that order). The initialization expression, if any, is connected to the termination expression, or the body of the construct, whichever is available (checked in that order). The termination expression, if any, connects to the body and the EndNode of the construct. If the end of the body is reachable, it connects to either the step expression, termination expression, or itself, whichever is present (checked in that order). Finally, the step expression, if any, is connected to the termination expression, body of the construct, or itself, whichever is present (checked in that order).

CallStatement. The BeginNode connects to the PreCallNode, which connects to the PostCallNode, which connects to the EndNode.

JumpStatement. Corresponding to all four types of jump statements, ~~an incomplete, or~~ a complete edge is created, if possible. Here, we discuss the approach of creating a complete edge, if the desired source/destination is available. ~~In the next section, we look into the creation of incomplete edges for these nodes.~~

- GotoStatement : First of all, we obtain the outermost non-leaf encloser for the given statement, using NodeInfo::getOuterMostNonLeafEncloser(). If there is no such encloser, then the edge cannot be complete. Otherwise, we search for the statement with required label in the outermost encloser. If no such statement is found, then ~~we need to add an incomplete edge~~. Otherwise, the goto statement is connected to the statement with required label.
- ContinueStatement : We check if there is any enclosing serial or parallel loop for this statement. If none exists, then ~~an incomplete edge needs to be created~~. Otherwise, if this encloser is a while loop or a do-while loop, we connect the node to the termination expression of the loop; In case of a for loop, we connect the node to either the step expression, termination expression, or body of the loop, depending upon whichever is available (checked in that order). If the encloser is an omp-for loop, we connect the node to the OmpForReinitExpression of the omp-for loop.
- BreakStatement : In case of a break-statement, we find the enclosing serial loop or switch statement. If no such encloser exists, then ~~an incomplete edge needs to be created at this node~~. Otherwise, we connect this node to the EndNode of the encloser.
- ReturnStatement : If there exists any enclosing function, we connect this return-statement to the EndNode of the function; otherwise, ~~an incomplete edge is added.~~

The method **CFGInfo::isEndReachable()** is defined on CFG nodes as follows : If the node is a jump-statement (*goto*, *break*, *continue* or *return*), then the end of this node is not considered to be reachable. For all other *leaf* CFG nodes, the end is considered to be reachable. For each non-leaf node, if and only if its EndNode has any predecessor, then the end of the non-leaf is considered to be reachable.

Note 7.1.3

The method CFGInfo::isEndReachable() assumes that CFG edges have already been populated within the visited node. One should never call this method on those nodes for which CFG edges are not yet created/maintained.

For each edge creation that is requested as per the scheme described above, the following steps are taken in the method connect(Node pred, Node succ) :

- If the edge to be created is *precise* (described below), then the edge is modelled by saving the successor in CFGInfo::getSuccBlocks() of the predecessor, and predecessor in CFGInfo::getPredBlocks() of the successor. If the edge is not considered to be precise, we don't create the edge. We ensure that there always exists only one edge between any two pair of nodes.
- When a predicate's value is a compile-time constant, we know the precise edge from the predicate to one of its destinations – all the other edges are considered to be imprecise (i.e., these edges are not created in the CFG). An edge is checked for precision by passing the source and destination nodes to the method **CFGGenerator.verifyEdgePrecision()**, which proceeds as follows :
  - If the source is not a predicate expression that evaluates to a constant at compile-time, then the edge is considered to be precise. Given an expression, Misc.evaluatesToConstant() decides whether it is a compile-time constant as follows : the type of the expression is obtained using Type.getType(Expression) (refer to Section 8); if the expression is not an arithmetic type, then the expression is not considered to be a constant; if an arithmetic floating-type expression evaluates to a known float value (of Java), or if an arithmetic integer-type expression evaluates to a known integer or character constant value (of Java), then the edge is considered to be precise; otherwise, the edge is considered to be imprecise.

Note 7.1.4

Note that the precision of the method Misc.evaluatesToConstant() can be improved by performing a pass of constant propagation and replacement first. We haven't yet implemented the same.

- Next, we check whether the constant expression evaluates to false (= 0), or true (= any non-zero value), and obtain the CFG *link* of the source node. A CFG link for a given node specifies what component of its enclosing CFG node is a given node. (Refer to Section 7.3 for more details).
- If the link corresponding to the source node isn't a WhilePredicateLink, IfPredicateLink, DoPredicateLink, ForTermLink, or a SwitchPredicateLink, then the edge is considered to be precise.
- When the source node's link is a WhilePredicateLink, and the predicate is false, then the edge from source to the EndNode of the while loop is precise, and the edge from source to the body of the loop is considered as imprecise. If the predicate is true then edge between source and body is considered to be precise, whereas the one between source and the EndNode is considered to be imprecise. Similar logic is applied for the case of IfPredicateLink, DoPredicateLink, and ForTermLink, as per the semantics of C language.
- Now, let's look at the case when source node link is of type SwitchPredicateLink. Depending upon the type of the switch's predicate, we obtain the target statement that corresponds to the given integer or character constant. If no such target statement could be found, then only that edge from the source is precise which connects it to the EndNode of the switch construct; all other edges originating at the source are considered as imprecise. If the target statement is found, but is not same as the destination node, then the edge is considered to be imprecise. If the target statement is found to be same as the destination node, then the edge is considered to be precise.

7.2 Creating incomplete edges

When the source or destination of a CFG edge is not available in a given snippet of a code, ~~we create *incomplete edges.*~~***UPDATE:* "we do nothing."** An incomplete edge can be of any of the following types (which are enum members of TypeOfIncompleteness) :

Note 7.2.1

UPDATE: Fri Aug 30 18:18:21 IST 2019
Now, we do not save the incomplete edges explicitly with any node; instead, they are created by looking into a node locally, whenever requested.

UNKNOWN_CASE_SOURCE : an incomplete edge terminating at a case statement that doesn't have any enclosing switch statement (and hence, the source predicate is unavailable).

UNKNOWN_DEFAULT_SOURCE : an incomplete edge terminating at a default statement that doesn't have any enclosing switch statement (and hence, the source predicate is unavailable).

UNKNOWN_GOTO_DESTINATION : an incomplete edge originating at a goto, such that there is no destination statement in the enclosing function/snippet, annotated with the target label.
UNKNOWN_BREAK_DESTINATION : an incomplete edge originating at a break, such that there is no enclosing loop or switch (to whose EndNode this break would have connected to).
UNKNOWN_CONTINUE_DESTINATION : an incomplete edge originating at a continue, such that there is no enclosing loop (to whose predicate this continue would have connected to).
UNKNOWN_RETURN_DESTINATION : an incomplete edge originating at a return statement, such that there is no enclosing function (to whose EndNode this return would have connected to).

There are various ways in which these incomplete edges are created for a given node, during/after creation of the complete edges. ~~During the visit of following types of nodes by CFGGenerator, the incomplete edges are created as described :~~

- ~~GotoStatement : If in the outermost non-leaf encloser of this node, there doesn't exist any statement that is annotated with the target label, then we add an incomplete edge of type UNKNOWN_GOTO_DESTINATION via a call to the method IncompleteSemantics::addToEdges(IncompleteEdge).~~
- ~~ContinueStatement : When the continue statement is not enclosed within a serial or parallel loop, we add an incomplete edge of the type UNKNOWN_CONTINUE_DESTINATION.~~
- ~~BreakStatement : In the absence of an enclosing serial loop or switch statement, we add an incomplete edge of type UNKNOWN_BREAK_DESTINATION.~~
- ~~ReturnStatement : When no enclosing statement exists for a return-statement, we add an incomplete edge of type UNKNOWN_RETURN_DESTINATION.~~

~~Once the visitor for creation of CFG edges returns, we invoke **CFGGenerator.handleIncompleteSwitchLabels()** on all the relevant CFG statements. In this method, we check if any other incomplete edges need to be added for internal case or default labels, in case if there is no surrounding switch statement for the given node. When no enclosing switch statement exists, we obtain the set of all those statements within node which contain case or default labels that are not bound to any enclosed switch statement. This collection is obtained using the visitor SwitchRelevantStatementsGetter. In the visitor, we ensure that no statements within any enclosed switch-statement are visited. For all other visited CFG statements, if the statement contains any case or default label, then we collect it into our set of interest.~~

For each collected statement, we add an incomplete edge of the type UNKNOWN_CASE_SOURCE or UNKNOWN_DEFAULT_SOURCE for each case or default label that the statement is annotated with.

Note 7.2.2

Update : Thu Aug 29 10:57:48 IST 2019.

Now, we do not explicitly save any incomplete edges with a node, but create them locally on demand instead.

After looking at all their current usages, we realized that we do not gain much in terms of computation by maintaining the incomplete edges explicitly. However, maintaining these incomplete edges during each elementary update involves complicated and costly logic.

Corresponding to each incomplete edge, following are the alternative methods that can be used to infer the notion of incompleteness in a CFG :

UNKNOWN_GOTO_DESTINATION. At any given GotoStatement, if the list of CFG successors is empty, then it implies that the destination label of this GotoStatement does not exist in the relevant context.

UNKNOWN_BREAK_DESTINATION. At any given BreakStatement, if the list of CFG successors is empty, then it implies that the relevant context does not contain the EndNode of the enclosing LoopStatement or SwitchStatement (i.e., there is no enclosing LoopStatement or SwitchStatement for the given BreakStatement).

UNKNOWN_CONTINUE_DESTINATION. At any given ContinueStatement, if the list of CFG successors is empty, then it implies that the relevant context does not contain the increment expression (if applicable), or predicate, of the enclosing LoopStatement (i.e., there is no enclosing LoopStatement for the given ContinueStatement).

UNKNOWN_RETURN_DESTINATION. If there is no successor of a ReturnStatement, then it implies that there is no enclosing FunctionDefinition for this statement, as otherwise this statement would have connected to the EndNode of that FunctionDefinition.

UNKNOWN_CASE_SOURCE and UNKNOWN_DEFAULT_SOURCE. For any statement that contains a CaseLabel or a DefaultLabel if there does not exist any predicate of a SwitchStatement in its list of predecessors, then it implies that there is no enclosing SwitchStatement for the statement.

7.3 Identifying CFG Links

A CFGLink denotes the nesting relation of a CFG node with its enclosing CFG node. For example, given an expression that is the predicate of a while-statement, when we invoke CFGLinkFinder.getCFGLinkFor() on the expression, we will get a WhilePredicateLink object as the result, which will contain references to this expression, as well as to the while-statement.

In the method **CFGLinkFinder.getCFGLinkFor()**, we first find the CFG node corresponding to the given argument. Then, we obtain the enclosing non-leaf CFG node for the argument. On this parent, we call the visitor CFGLinkGetter to obtain the link corresponding to the nesting relation between the parent and the given argument. In this visitor, we have overridden the visits of all non-leaf CFG nodes, such that the provided argument is checked for equality with the immediately

nested CFG components of the non-leaf node. Accordingly, a link is created and returned back by the visitor.

8 GETTING TYPE OF AN EXPRESSION

The method **Type.getType(Expression)** utilizes the visitor ExpressionTypeGetter in order to obtain the type information for the given expression. In this visitor, each visited expression returns back the type of that expression. From visits of nodes that do not correspond to an expression, null is returned.

When a variable is accessed in an expression, we attempt to obtain its symbol using Misc.getSymbolEntry(String name, Node node). The working of this method is explained in detail in the Section 9. If no symbol is found corresponding to a variable, then we assume the type of such free-variable to be SignedIntType. (Note that later we are planning to use a FreeType here.)

Note 8.0.1

A note on pointer generation.
As per the semantics of C language, the following holds. A symbol of type array of T will remain to be of that type; however, when that symbol is *used*, its type becomes pointer to T. Similarly, a function symbol of type function returning T will remain to be of that type; when that function is *used*, its type becomes pointer to function returning T. These automatic pointer generation will *not* happen when the symbol is being used an operand of unary &, ++, –, sizeof operators, or is present in the LHS of an assignment or a dot operator.

When visiting a comma expression, the type of the expression is denoted by the type of the last expression in the comma-operands. In case of a NonConditonalExpression, we undo any automated pointer generation and return the original type of the LHS.

For a conditional expression, the type is provided as the usual arithmetic conversion of the types of the second and third expressions, if they both are arithmetic types. This conversion is provided by Conversion.getUsualArithmeticConvertedType(Expression, Expression). Otherwise, if they both are struct type, or both are union type, or both are void type, then the returned type is that of the second (which is equal to the third) expression.

Upon visiting a logical OR, or logical AND expressions, the returned type is SignedIntType. For bitwise inclusive OR, bitwise exclusive OR, and bitwise AND expressions, the return type is the usual arithmetic conversion of the types of both the operands. In case of an equality-check expression (== or !=), or a relational expression, the return type is SignedIntType. For a bitwise left or bitwise right shift operation, the resulting type is the integer-promoted type of the left operand.

The resulting type of an additive operation is the usual arithmetic conversion of both the operands, if the operands are of arithmetic type; if one of the operands is of pointer type, and another

is an integer type, then the resulting type is same as the pointer type; when both the operands are pointers, then the resulting type is a signed long int (which should have been ptrdiff_t as defined in stddef.h instead). For multiplication operation, the resulting type is the usual arithmetic conversion of the types of the operands. In case of a cast expression, the resulting type is same as the cast type. This type is obtained from the given type name using Type.getTypeTree(TypeName, Scopeable), where the second argument is the enclosing scope (a CompoundStatement, FunctionDefinition, or TranslationUnit) in which the expression occurs. For pre-increment and pre-decrement operators, the returned type is same as the original type of the operand (i.e., pointer generation, if any, is undone).

The resulting type of applying & operator is the pointer to the original type of the operand. On the other hand, if the operand is of type *pointer to T*, then the resulting type upon applying * operator would be *T*. In case of a unary +, unary -, or unary ~ operator, the resulting type is the integer promoted type of the operand, obtained using Type::getIntegerPromotedType. The resulting type upon application of the logical negation operator (!) is SignedIntType. For sizeof operator, the resulting type is UnsignedLongIntType.

A PostfixExpression comprises of a PrimaryExpression, optionally followed by a sequence of postfix operators. If the type of the primary expression is unknown, and if the first operator is a list of arguments, then this postfix expression corresponds to a call of a function with unknown signature. In such a case, we assume the type of this expression (i.e., the return type of the unknown function) to be SignedIntType, instead of relying on any type inference algorithm. Otherwise, if the first postfix operand of an unknown primary-expression is something else, then we assume that the type of this postfix-expression is null.

Given a list of postfix operations on a primary expression with known type, we proceed as follows : We take each operator one-by-one, from left to right, and keep on obtaining the type of the partial postfix expression visited so far, upon application of each operand, starting with the type of the primary expression. When the current type is an ArrayType or a FunctionType, we perform pointer generation on the type, if applicable. (However, this case might not occur ever as the type for each symbol has already undergone pointer generation.) When the next operator is a BracketExpression, we need to find whether the index expression is an integer, or the partial expression visited so far is. Then, if the other expression is of type *pointer to T*, the type for the partial expression so far, after application of this operator, would be *T*. If the next operator is an argument list, then the partial expression so far should be of type *pointer to function returning T*; upon application of the argument list, the new type would then be *T*. For DotId, first of all we undo the pointer generation on the current type. Then, the current type would either be a StructType or a UnionType. In that type, we search for the member corresponding to the identifier on RHS of

the dot operator. The type of that member becomes the type of the partial expression on which the dot operator has been applied. When the next operator is a ArrowId, then the current type must be a pointer-type. We obtain the type of the pointee, which would either be a StructType or a UnionType. As before, we obtain the type of the member being dereferenced, which becomes the type of the partial expression on which dereference has been performed. When the next operator is a postfix increment, or decrement, then the type of the partial expression after application of the operator remains same as the current type.

When a PrimaryExpression is an identifier, as described above, we obtain the symbol corresponding to that identifier. If none exists, we return null, else we return the type of that symbol after performing pointer generation on it, if applicable.

Given a Constant, firstly we try to collect a list ArithmeticTypeKey's corresponding to the constant. In case of an integer constant, we check if the constant is a long and/or unsigned with the help of appropriate prefixes (e.g., *l*, *L*, *u*, *U*, etc.), apart from being an int, and create the arithmetic keys accordingly. For a floating-point constant, we check for the appropriate suffixes to determine whether the constant is a single-precision float, or a double, and create the keys accordingly. In case of a character constant, or a string literal, we assume that char is the only key. After collecting all the arithmetic type keys, we use the method Type.getTypeFromArithmeticKeys() to obtain the inferred type of the integer, floating, or character constants. In case of a string literal, we wrap the obtained type in a pointer-type, and return it.

A SimplePrimaryExpression can either be a constant or an identifier. If it is a constant, then we return the type, as obtained during the visit of that constant, as it is. In case of an identifier, we try to obtain the corresponding symbol (variable or function). If no such symbol exists in the snippet, then we return null. Otherwise, we return the type of the symbol, after applying pointer generation, if applicable.

***UPDATE:* "16-May-2019"** In order to ensure that pointer generation and degeneration are done correctly, we have performed the following changes :

- We have created the following two methods : ExpressionTypeGetter.performPointerGeneration() and ExpressionTypeGetter.performPointerDegeneration(). The former takes a type, and performs pointer generation on it, if applicable, to obtain the type to be returned. Whereas, the latter returns back the type on which pointer generation might have been applied to obtain the passed type.
- In each visit, we ensure that all inferred types, or types read from symbol tables, undergo pointer generation. While passing the type up the expresison-tree, we do not perform any pointer generation or degeneration. To obtain the type of the unary operands of &, ++, −−, `sizeof`,

and LHS operand on assignment operators and dot operator, we perform pointer degeneration on the type obtained by the visitor.

9 SYMBOLS

Conceptual details about the notion of a symbol are present in the preliminary technical report of IMOP. Given an identifier, and a program point (i.e., a Node), we use **Misc.getSymbolEntry()** to obtain the symbol corresponding to the identifier at that program point. In this method, starting with the given node, we traverse upwards on the AST, and see if any scope contains declaration of a symbol with this name; if we find one, we return it.

Note 9.0.1

Note that if the node itself is a scope, its internal symbol table too is checked in Misc.getSymbolEntry(), while searching for the symbol.
Question : Is this logical to do? If we don't consider the internal symbol table, will it require changes in any part of the code?

Otherwise, if the node is not connected to the program (i.e., the traversal ended without reaching a TranslationUnit), then we search for the symbol in the global scope of the program. Otherwise, we resort to checking into the list of built-in library symbols. Finally, if the symbol is not found anywhere, we return null. Note that while attempting to read the symbol table of the built-in library methods, we should not save the observed cells/symbols in the set of all the cells/symbols of the program. Also, while checking in the built-in libraries, we ignore the prefix `__builtin_`.

The symbol table of a scope is obtained by the methods : RootInfo::getSymbolTable(), FunctionDefinitionInfo::getSymbolTable(), or CompoundStatementInfo::getSymbolTable(). These methods either return the symbol-table of the scope if it has been populated already, or call the corresponding populateSymbolTable() first (which also reinitializes the type and typedef tables).

9.1 Initialization of Symbol-, Typedef-, and Type-tables

The method CompoundStatementInfo::populateSymbolTable() is used to populate the symbol-table, typedef-table, and type-table of a given compound statement. Firstly, we obtain the list of declarations that are made at the level of the compound-statement. Each of these declarations could be a declaration of a symbol, type, or a typedef. For each declaration, we obtain the list of identifiers declared in the declarator. If the list is empty, then the declaration is a type declaration that contains no declarators of that type. For such cases, we utilize the method Type.getTypeTree(Declaration, Scopeable) to visit and collect the declared type. If the list of declarators is not empty, then for each declarator, we obtain the type of the declarator using the same method, Type.getTypeTree(). If the declaration is a typedef, then we see if a mapping already exists for this declarator in the

typedef table – if so, then we ignore this declarator; otherwise, we add a mapping from this declarator's name to a new Typedef (which comprises of the name, type, declaring node, and the declaring scope, of the typedef.) Finally, if the declarator declares a Symbol, we proceed as follows. If the mapping already exists for this name, we ignore the declarator. Otherwise, we create a new mapping for this name with the associated symbol. Generally, we create a new symbol object, with the appropriate name, type, declaring node, and declaring scope. However, there are times when a declaration is removed from the program and added somewhere else; in such cases, we try and reuse the symbol, as explained next. We maintain a set of cells that have been deleted from the program. If there exists any cell which is a symbol with same name as the one provided to this method, then we reuse it to create the mapping in the symbol table, under the following additional constraints : the old symbol should have a declaring node of type Declaration, it should have its declaring scope of type CompoundStaement, and this scope should either enclose, or be enclosed by the scope of the provided declaration. (Assuming that there are no naming conflicts during movement of a declaration across nesting of scopes, the last condition here ensures that such movements do not generate recreation of a new Symbol.)

In case of the method FunctionDefinitionInfo::populateSymbolTable(), note that a function's signature would not contain any type or typedef declaration; it is comprised up of a list of parameter declarations, instead. Firstly, we obtain the symbol corresponding to this function in the symbol table of the program's TranslationUnit. Given the function symbol, we iterate over all the parameters, and create a mapping in the symbol table of the function, from parameter's name (or a new name, if none exists) to the symbol corresponding to that parameter. A symbol corresponding to a parameter comprises of the parameter's name, type, corresponding ParameterDeclaration, and the function itself (which is the scope, in this scenario). Note that while the temporary name of the parameter is used as a placeholder at various places, this name is not reflected in the AST of the program (to prevent any surprises).

To initialize the global symbol table of the program, i.e., of the associated TranslationUnit, we use the method RootInfo::populateSymbolTable(). Like compound statement, the declarations in the global scope too may correspond to a symbol, type, or a typedef. Hence, all three corresponding tables (symbol table, type table, and typedef table) are initialized by this method. Also note that each symbol can either correspond to a variable, or a function. For each function symbol, there may exist more than one declarations, only one of which may contain the body of the function. The method proceeds as follows. We consider each element in the translation unit, one-by-one.

If the element is not a Declaration or a FunctionDefinition, we ignore the element. When the element is a FunctionDefinition, we first obtain its name. Then, type of the function is obtained using Type.getTypeTree(FunctionDefinition, TranslationUnit). Using the name of the function, we

add a new mapping (overridding any existing ones) to the function symbol. A function symbol comprises of the function's name, type, function-definition node, and a reference to the TranslationUnit corresponding to the program (which is the *scope* of this symbol). When the element is a Declaration, we check the list of declarators declared in it. If the list is empty, then this is a type declaration, for which the type-tree is created and added to the type table. Otherwise, for each element in the list, we first obtain its type. If the declaration is a typedef declaration, then we add a mapping from the declarator's name to the associated Typedef in the typedef table, if no mapping already exists for that name. Otherwise, the element corresponds to a symbol declarator, for which we add a mapping in the symbol table, from the declarator's name, to the newly created symbol, if there is no mapping corresponding to the symbol already. (Note that this last clause is important, so that a function declaration should not override information about a function definition.)

10 CELL ACCESSES IN A NODE

The class CellAccessGetter contains various visitors and methods that are used to obtain the locations (cells) represented by any given Expression, sets of cells that may have been read or written within a given Node, etc. Following are the key methods of this class :

getReads(). This method is used to get a list of cells that may be read in a given node. If the given node is an Expression, we invoke the visitor AccessGetter on it, and return the concatenation of the returned list (which would represent the locations represented by the CFG node corresponding to the Expression) with the list AccessGetter::cellReadList.

Otherwise, we take each leaf CFG node that is inter-procedurally present within the CFG node corresponding to the given node, and process it in a similar fashion, using AccessGetter. As before, the result of processing of each node is the return of the visitor's invocation, concatenated with the list AccessGetter::cellReadList. Finally, the result of each processed leaf is concatenated and returned back.

Note that the first argument of the constructor of an AccessGetter indicates whether the lists are collected only for shared cells (selected by sending `true`), or for all cells (selected by sending `false`). This method sends `false` in all its invocations of the constructor of AccessGetter.

getSymbolReads(). This method is used to get a list of symbols (and not other kinds of cells) that may be read in a given node. This method is exactly similar to the method getReads(), except that it utilizes the visitor SymbolAccessGetter instead of AccessGetter.

getWrites(). This method is used to get a list of cells that may be written in a given node. If the given node is an Expression, we invoke the visitor AccessGetter on it, and return the list AccessGetter::cellWriteList.

Otherwise, we take each leaf CFG node that is inter-procedurally present within the CFG node corresponding to the given node, and process it in a similar fashion, using AccessGetter. The result of each processed leaf is the list AccessGetter::cellWriteList, which is then concatenated for all leaves, and returned back.

This method sends `false` as the first argument to all invocations of the constructor of AccessGetter.

getSymbolWrites(). This method is used to get a list of symbols (and not other kinds of cells) that may be written in a given node. This method is exactly similar to the method getWrites(), except that it utilizes the visitor SymbolAccessGetter instead of AccessGetter.

mayWrite(). When we need to check whether a given node may write to any cell, it would be too inefficient to first obtain the list of cells that may be written in the node, and then check if the list is empty. In such scenarios, we use this method which informs whether there are *any* writes in the given node.

If the given node is an Expression, this method invokes the visitor MayWriteCheker on it, and returns the boolean MayWriteCheker::mayWrite.

Otherwise, this method traverses through each inter-procedurally contained leaf node for this node, one by one, and performs the check. If any leaf node may write to any location, then this method immediately returns `true`. Otherwise, after checking all the leaf nodes, this method returns false.

Note that a similar method could be written for checking for reads instead of writes. However, we have not yet encountered any possible utility of such a method.

getSharedReads(). This method is used to obtain a set of shared cells that may have been read within a node. It is exactly similar to the method getReads(), except that : (i) it returns a set instead of a list, and (ii) it passes `true` to the visitor AccessGetter.

getSharedWrites(). This method is used to obtain a set of shared cells that may have been written within a node. It is exactly similar to the method getWrites(), except that : (i) it returns a set instead of a list, and (ii) it passes `true` to the visitor AccessGetter.

getLocationsOf(). Given an expression, in many scenarios we wish to obtain the list of cells that the expression might *denote*. For example, `*ptr` denotes all those cells that are in the points-to set of `ptr`, at the leaf CFG node in which the expression `*ptr` appears. For such situations, we use the method getLocationsOf().

If the given expression is lexically equal to a *(void *) 0* or *0*, then we conservatively consider the expression to represent the `null` cell (i.e., Cell.nullCell).

Otherwise, we invoke the visitor AccessGetter on the expression, passing `false` as the first argument, and then simply returning the list returned by the visitor. If the returned list is `null`, we return an empty list instead.

mayRelyOnPointsTo(). Given a node, this method checks whether any of the access lists of this node may depend on the points-to information. Such knowledge is quite useful in deciding whether we need to stabilize the global points-to information before recalculating the access lists for a given node.

If the given node is an Expression, then the visitor PointsToRelianceGetter is invoked, and its field PointsToRelianceGetter::reliesOnPointsTo is returned.

Otherwise, similar processing is done on all inter-procedurally contained leaf nodes of the given node, and `true` is returned as soon as a leaf node is found for which PointsToRelianceGetter::reliesOnPointsTo is `true`; if no such leaf node is found, then this method returns `false`.

mayRelyOnPointsToForSymbols(). This method is similar to the method mayRelyOnPointsTo(), except that it checks whether the accesses of any symbols in the node (and not necessarily those of any cells of other kinds) require stabilization of points-to information. It uses the visitor PointsToForSymbolsRelianceGetter instead of PointsToRelianceGetter; rest of the code remains same.

mayUpdatePointsTo(). This method is used to check whether execution of the given node may update the points-to information.

If the node is an Expression, this method invokes the visitor MayUpdatePointsToGetter, and returns its field MayUpdatePointsToGetter::mayUpdatePointsTo.

Otherwise, the similar processing is done for each *lexically enclosed* leaf CFG node of the given node, and `true` is returned as soon as a leaf node is found for which the field MayUpdatePointsToGetter::mayUpdatePointsTo is set. Otherwise, the similar steps are performed on all leaf CFG nodes that are lexically present in any of the other FunctionDefinitions that are reachable from the given node. If the field MayUpdatePointsToGetter::mayUpdatePointsTo is not set for any of the leaf nodes, then this method returns `false`.

Note 10.0.1

Note that corresponding to all static methods of CellAccessGetter, there exists a member method with same name in NodeInfo (or ExpressionInfo, in case of getLocationsOf()), which should be preferred over the static methods of CellAccessGetter, as the member methods memoize various results, which can reduce re-computation efforts.
The invalidation/update of the memoized data is done automatically under any transformations of the program that are performed, directly or indirectly, using elementary transformations.

Now, we discuss key points about various visitors that have been utilized by different methods listed above :

AccessGetter. This visitor maintains two lists, cellReadList, and cellWriteList, which are populated by the visitor with list of cells that may have been read or written by the visited node. If its field isForShared is set, then both these lists contain only shared cells.

Each visit in this visitor performs the following steps : (i) adds all/shared cells that are clearly read from at the node into cellReadList, (ii) adds all/shared cells that are clearly written to at the node into cellWriteList, and (iii) returns the list of cells, if any, that the visited node denotes.

The following two methods of AccessGetter are used to perform the first two tasks above :

(i) addReads() Given a node and a list of cells that may have been read at the node, this method adds the universal cell to the list cellReadList, if the given list contains a universal cell. Otherwise, if the flag isForShared is set, then it adds all those cells from the given list to cellReadList which are shared at the given node (i.e., invocation of NodeInfo::getSharingAttribute() for that cell returns DataSharingAttribute.SHARED). If the flag is not set, then all elements of the given list are added to the cellReadList.

(ii) addWrites() This method is similar to addReads(), except that it adds the given elements to cellWriteList instead of cellReadList.

Note 10.0.2

Note that a more precise (but slower) version of code exists for the scenario where the given list contains the universal cell. To enable the more precise version, set the flag morePrecise within addWrites().

Next, we discuss some key observations concerning various kinds of visits in AccessGetter :

- A NodeToken that represents an <IDENTIFIER> lexeme, denotes the Symbol or FreeVariable returned by invocation of Misc.getSymbolOrFreeEntry().
- A Declaration is always considered as a write to the symbol being declared, unless it is a typedef declaration.
  If the symbol being declared in an InitDeclarator is of type ArrayType, then we assume that its FieldCell gets written as well. If there exists any Initializer, then the cells that it denotes are considered as read, and the visitor is called recursively on it. Similarly, a ParamaterDeclaration is considered to be a write of the parameter that it declares.
- An IfClause, a FinalClause and a NumThreadsClause read the cells denoted by their expressions.
- An ExpressionStatement and a ReturnStatemnet are considered as reads of the cells denoted by their expressions.
- Expressions within the sizeof() operators are not considered as read or written.

- A PreCallNode is considered as read of the cells denoted by all its arguments. Furthermore, if the corresponding CallStatement does not have any known destinations, then conservatively we assume that the PreCallNode can be read/write of all the cells that may be accessible via the arguments. All such accessible cells are obtained by the method AccessGetter::getOptimizedPointsToClosure – this method returns the closure of the points-to sets of the given arguments.
- The operands of various unary and postfix operators are added to the read and write lists as per the semantics of the operators. [3]
- Any other operator with no side-effects reads the cells denoted by its operands. If there are any side-effects, then the cells denoting the corresponding operands are considered as written as well.

SymbolAccessGetter. This visitor works similar to the way AccessGetter works, except that it utilizes Program.getCellsThatMayPointToSymbols() to avoid triggering unnecessary calls to stabilization of the points-to information. This method is used to obtain the set of all those cells that may contain a Symbol in its points-to set. This processing is done in flow-insensitive manner, without having to rely on the full-fledged points-to analysis. It is based on the notion of address-taken symbols, where the method attempts to find all those cells that may have been assigned the address of any of the symbols, directly or indirectly.

MayWriteChecker This visitor simply visits over all the nodes and sets the flag MayWriteChecker::mayWrite at all those places where the visitor AccessGetter would have added any cells to the set cellWriteList.

PointsToRelianceGetter. This visitor sets its flag PointsToRelianceGetter::reliesOnPointsTo at all those places where a cell may have been dereferenced using a `*` `->`, or `[]` operators. It also sets its field if there exists any CallStatement with missing target, such that it contains an argument which is not of ArithmeticType.

PoinstToForSymbolsRelianceGetter This visitor relies on the method Program.getCellsThatMayPointToSymbols(); the visitor would set its field PoinstToForSymbolsRelianceGetter::reliesOnPointsTo if there exists any dereference (using `*`, `[]`, or `-> operators`) on a cell that may be present in the return of Program.getCellsThaMayPointToSymbols().

MayUpdatePointsToGetter. This visitor sets its field MayUpdatePointsToGetter::mayUpdatePointsTo at all those places where the visitor AccessGetter would have added any cells of type PointerType to the set cellWriteList.

[3]This code review document currently does not contain review for visits of the unary and postfix operators as they have already been tested earlier.

11 SIDE EFFECTS

Due to various syntactic and semantic constraints, any CFG transformation may result in generation of various side-effects in the program [4].

Note 11.0.1

Each elementary and higher-level transformation returns a list of side-effects back to the caller; the onus of handling these side-effects in the client code is on the caller.

In IMOP, we represent a side-effect using an enumerator UpdateSideEffects. While we will discuss their exact usage in proper contexts later, following is a list of various side-effects :

- **ADDED_DFD_SUCCESSOR**, **ADDED_DFD_PREDECESSOR**, **REMOVED_DFD_SUCCESSOR**, and **REMOVED_DFD_PREDECESSOR** : When a request is made to insert an OpenMP construct/directive in the program, such that the construct/directive contains an implicit flush at the entry to or exit from the construct, then we need to ensure that we insert the corresponding DummyFlushDirectives as the predecessor and/or successor of the inserted construct/directive. This side-effect is conveyed back to the caller by adding **ADDED_DFD_PREDECESSOR** and/or **ADDED_DFD_SUCCESSOR** to the return list.

  Similarly, upon receiving a request to remove a node that contains implicit flushes at its entry and/or exit, IMOP automatically removes the corresponding DummyFlushDirectives, if any, and adds REMOVED_DFD_PREDECESSOR and/or REMOVED_DFD_SUCCESSOR to the return list.
- **UNAUTHORIZED_DFD_UPDATE** : Since IMOP automatically ensures the insertion/deletion of DummyFlushDirectives at all places where an implicit/explicit flush exists, it doesn't allow a user to explicitly specify requests to insert or delete DummyFlushDirectives. Whenever such requests are made, they are ignored, and a side-effect UNAUTHORIZED_DFD_UPDATE is added to the list to be returned to the caller.
- **NO_UPDATE_DUE_TO_NAME_COLLISION** : When insertion of a node may cause inconsistencies in the bindings of the free variables of the node, with the variables declared in the scope where insertion has to be made, then the transformation does not complete, and adds a side-effect named NO_UPDATE_DUE_TO_NAME_COLLISION to the return list.
- **ADDED_EXPLICIT_BARRIER**, and **ADDED_NOWAIT_CLAUSE** : When a request is made to insert a ForConstruct, SectionsConstruct, or SingleConstruct, then in the absence of a `nowait` clause, an explicit barrier is automatically inserted as the immediate successor of the construct, and the implicit barrier is removed by adding a `nowait` clause. In such cases, the

[4] If a transformation cannot be performed, we consider *that* also as a side-effect.

side-effects ADDED_EXPLICIT_BARRIER and ADDED_NOWAIT_CLAUSE are added to the return list.

- **ADDED_ENCLOSING_BLOCK** : There are various circumstances where a node to be inserted is first wrapped into a compound statement (i.e., a block). For example, IMOP ensures that bodies of all C and OpenMP constructs (except for AtomicConstruct) should be compound-statements. Hence, if a request is made to add a single non-compound-statement (e.g., a function call) as the body of any of such constructs, then the statement is automatically enclosed within the basic block, and a side-effect ADDED_ENCLOSING_BLOCK is added to the return list.
- **MISSING_CFG_PARENT** : When an attempt is made to add a snippet of code outside another base snippet of code (e.g., as the successor or predecessor of the base snippet), then the transformation fails and adds MISSING_CFG_PARENT to the return list. Similarly, this side-effect is also used in the scenario where an attempt is made to remove a snippet of code that does not contain any enclosing snippet or program.
- **SYNTACTIC_CONSTRAINT** : There are various types of translations that cannot be performed owing to the syntactic constraints of C, OpenMP, or IMOP. For example, a BeginNode or EndNode cannot be added or removed manually. For all such cases, the side-effect SYNTACTIC_CONSTRAINT is also added to the return list, along with some more specific side-effect, if any.
- **INDEX_INCREMENTED** and **INDEX_DECREMENTED** : In case of insertion/removal of an element (let us say, a statement) in a list of elements (let us say, a compound-statement) at a given index, if the index at which the statement is actually inserted gets incremented or decremented due to automated addition or removal of special nodes, such as DummyFlushDirectives or an explicit barrier, then a side-effect INDEX_INCREMENTED or INDEX_DECREMENTED is added to the return list, to indicate the same.
- **NAMESPACE_COLLISION_ON_REMOVAL** and **NAMESPACE_COLLISION_ON_ADDITION** : As a result of addition/removal of a declaration of a symbol, say *temp*, to/from a scope, if the bindings of identifiers with name *temp* used within that scope changes (instead of simply getting converted from/to a FreeVariable), then such side-effects are indicated using NAMESPACE_COLLISION_ON_REMOVAL and NAMESPACE_COLLISION_ON_ADDITION.
- **ADDED_COPY** : Due to various syntactic constraints, there might be no place where a single copy of a node can be inserted, when a request is made to insert the node as an immediate successor of predecessor of the base node. It may happen that two or more copies of the target node might have to be created and inserted to attain the expected semantics. For example,

while attempting to insert successors/predecessors of various predicates of different loops, such conditions arise easily where two copies of the target node gets inserted.

- **INIT_SIMPLIFIED** : When attempting to insert a declaration, if the declaration or its initializer gets simplified automatically, then the side-effect INIT_SIMPLIFIED is returned via the return list.
- **JUMPEDGE_CONSTRAINT** : When the source or destination of the labeled or jump statements can get incorrectly matched as a result of a transformation, the transformation fails, while returning JUMPEDGE_CONSTRAINT as one of the side-effects.
- **REMOVED_DEAD_CODE** : As the name suggests, this side-effect is generated when dead code is removed as a result of some transformation. (Details for the same would be present in some later section.)

12 ENFORCING ALL BODIES TO BE COMPOUND STATEMENTS

The visitor **CompoundStatementEnforcer** is used to ensure that the bodies of all the visited nodes are converted to CompoundStatement, if they are not already so. This invariant is utilized by various downstream transformations in IMOP.

Note that as per the grammar, a FunctionDefinition would always have a CompoundStatement as its body. Nodes of type CallStatement or AtomicConstruct do not contain a body. A CompoundStatement can have a list of statements of any type as its body. For all the other remaining non-leaf CFG nodes except SectionsConstruct and IfStatement, this visitor performs the transformation in a top-down manner, as follows :

- First of all, the body of the node is checked for whether it is already a CompoundStatement. If so, then no further processing is done for this node.
- If the body is not a CompoundStatement, then a new empty compound statement is created, and the old body is replaced by this new compound statement (with the help of setBody() methods in various CFGInfo objects for different nodes). Then, the old body is added as the sole element of this newly added compound statement (using the method CompoundStatementCFGInfo::addElement()).
- After processing of the node finishes, this visitor is called recursively on the body of the node.

In case of a SectionsConstruct, there are multiple sections, all of which should be translated to have their bodies as CompoundStatement. Firstly, we obtain the list of CFG nodes that represent various sections present in this construct. If any of the CFG nodes is not a CompoundStatement, then, as above, an empty CompoundStatement is created; the old CFG node is removed (using SectionsConstructCFGInfo::removeSection()), and the newly added compound statement is added at the same index as that of the old CFG node (using SectionsConstructCFGInfo::addSection());

finally, the CFG node is inserted as an element in this newly added compound statement. The visitor is called recursively on all the sections in the construct.

In case of an IfStatement, similar processing as above is applied on both, its true branch, as well as its false branch, if any.

Note that none of the elementary transformations called from this visitor could have any update side-effects.

13 IMPLICIT BARRIER REMOVAL

The method **ImplicitBarrierRemoval.removeImplicitBarrier()** is used to make all implicit barriers within the provided node, explicit, with the help of nowait clauses.

Note 13.0.1

Note that the implicit barrier at the end of a parallel construct is not removable, syntactically.

In this method, we collect a post-order list of ForConstruct, SectionsConstruct, and SingleConstruct nodes that are present within the given node. For each element in the list, we use ImplicitBarrierRemover.obtainNormalizedNode() which returns a statement that does not contain any implicit barriers. This method works as follows :

- First of all, the CFG node corresponding to the given node is obtained using Misc.getCFGNodeFor().
- If the node is a ForConstruct, we check whether the `nowait` clause is present in the clause-list, using the method OmpConstructInfo.hasNowaitClause(). If so, then the node is returned as it is. Otherwise, using string-based AST construction, via the method `FrontEnd.parseAndNormalize(String, Class<? extends Node>)`, we create a compound statement that contains the provided for-construct, with a `nowait` clause attached to it, followed by an explicit barrier. Note that the labels of the for-construct are shifted to the created compound-statement. This compound-statement is then returned by this method.
- Similar steps are applied in the case of SectionsConstruct and SingleConstruct as well.

If the returned statement is not same as the provided element, then we replace the element with the returned statement using NodeReplacer.replaceNodes(). Note that the call to replaceNodes() will not return any side-effects at this call-site.

Optimization: Why don't we change the existing structure and insert an explicit barrier as its successor? Note that due to CS-Enforcer, every OpenMP construct would be present only within a CompoundStatement, when connected to the program. Of course, if the construct isn't connected to the program, it can be present as an standalone entity without any enclosing block (i.e., no successor or predecessor can be added).

14 EXTRA SCOPING REMOVAL

When statements are nested within unnecessary levels of scoping, then various transformations may get hampered. In order to remove such unnecessary scoping of statements (i.e., within CompoundStatements), the method **NodeInfo::removeExtraScopes()** is used (which, in turn, calls CompoundStatementNormalizer.removeExtraScopes()), as explained in this section.

Using the method NodeInfo::getAllSymbolNamesAtNodeExclusively(), we first obtain the set of names for all those symbols that are accessible at the (AST) parent node of the given node (i.e., if this node is a Scopeable object, then we do not consider the set of names for symbols present in the symbol table of this node). This set is passed to the visitor CompoundStatementNormalizationVisitor, which removes extra scopes from all nodes within the given node as follows :

- This visitor visits all the blocks that may contain statements within them. In each visit, it takes an argument, which corresponds to the set of symbols that are accessible at the (AST-) parent node of the node being visited. For efficiency purposes, we truncate traversal at those nodes (by overriding existing definitions with empty bodies) which may not contain any statement within them.
- While visiting any elements within the TranslationUnit, the set of symbol names present in the symbol table of the translation unit are sent as an argument to the visit.
- When visiting the body of a FunctionDefinition, the set of symbol names that are sent as an argument comprises of the set received by the function's visit, as well as the parameter names.
- The main code for removal of extra scopes lies in the visit of the nodes of type CompoundStatement. For any given CompoundStatement, we first call the visitor recursively on its elements, to ensure that the scope removal is done inside out. In order to obtain the set of names to be passed to the visits of the elements, we take a union of the set obtained as the argument and the set of names declared in this CompoundStatement.

  After the visit of all the elements of a CompoundStatement (say, an *enclosing compound-statement*) is complete, we perform the nesting removal as follows. If any given element itself is a CompoundStatement (say a *nested compound-statement*), we check whether there can be any name collisions if all the elements of the nested compound-statement are brought up to the level of the enclosing compound-statement.

  If there are no name collisions, we take each element of the nested compound- statement, and bring it up to the level of the nested compound-statement one-by-one. Note that if the nested compound-statement had any labels to begin with, then those labels must be shifted to the first internal statement that has been brought up to the level of the enclosing compound-statement. This is achieved using the method StatementInfo::addLabelAnnotation(), an elementary transformation. Next, each element of the nested compound-statement is removed using the method

CompoundStatementCFGInfo.removeElement(). Note that the removal may not succeed if the element to be removed is a DummyFlushDirective. However, the maintenance of the same is done automatically by IMOP, hence nothing specific needs to be done in this case. No other consequential side-effects will result during removal of the element.

Next, the removed element is inserted at a proper index (calculated by keeping track of number of elements inserted into the enclosing compound-statement so far, and the index of the nested compound-statement in the enclosing compound-statement), using CompoundStatementCFGInfo.addElement(). If the element to be inserted is a DummyFlushDirective, then the insertion will not succeed – in such a case, we neither increment the counter for the number of elements inserted, nor do we increment the insertion index. We do not need to do anything else, in this scenario. Otherwise, we increment both these counters, and handle the rest of the side-effects of insertion, if any, as follows :

- ADDED_DFD_PREDECESSOR or ADDED_DFD_SUCCESSOR : the counter for number of elements inserted, and the insertion index, both are incremented further by one.
- No other side-effect could have been returned by the addition request.

After bringing up certain declarations from the nested compound-statement to the enclosing compound-statement, we need to ensure that the set of names for symbols declared in the enclosing compound-statement gets updated with the newly added names, before processing any other nested compound-statement for removal.

Finally, we remove the nested compound-statement from its position within the enclosing compound-statement, and decrement the counter that represents number of elements inserted by one.

UPDATE: **"Currently, we do not perform this transformation on the disconnected snippets of the code; in future, we plan to update this algorithm by sending only those names from the enclosing compound-statement to the visit of the nested compound-statement that are *used* in any of the elements other than the nested compound-statement. "**

15 UNUSED DECLARATIONS REMOVAL

During preprocessing, a large number of unused function definitions, and unused declarations of types, typedefs, and symbols get added to the source code, as a result of inclusion of the header files. In order to reduce the code size, and more importantly, the size of the universal sets of types, typedefs, and symbols, we need to remove various unused declarations from the source code. Similarly, to reduce the number of function symbols, we remove the definitions for all those functions which cannot be reached from the main() function, if any.

Towards this goal, we use the function **NodeInfo::removeUnusedElements()** for all types of nodes, except TranslationUnit, for which we use the overridden variant RootInfo::removeUnusedElements(). The latter simply calls RootInfo::removeUnusedFunctions() (to remove those functions which have not been called from any code reachable from main()), followed by a call to the former, which performs the following steps : (i) remove unused variables using NodeInfo::removeUnusedVariables(), (ii) remove unused typedefs using NodeInfo::removeUnusedTypedefs() until no more typedefs can be removed, (iii) remove unused types using NodeInfo::removeUnusedTypes() until no more types can be removed, and finally (iv) repeat the last two steps, until fixed-point is reached.

Now, we look into the details of all these methods.

15.1 Removing unused functions

Since ISO C does not support the notion of nested functions, we allow this method only on a TranslationUnit (i.e., the whole program).

Given a program, list of all its function definitions can be obtained using the visitor AllFunctionDefintionGetter, which works quite simply by collecting all visited FunctionDefinition nodes. This list gets memoized in RootInfo.allFunctionDefinitions) upon first use. Using this list, we obtain the main() function. If none exists, we do not perform removal of unused functions.

In order to gather the names of all those functions corresponding to which a CallStatement exists in some reachable code from main(), we use the method NodeInfo::getReachableCallStatementsInclusive() on main(). This method relies on a lambda-based graph collector (refer to Section 17), with following arguments :

- We obtain the start nodes for the graph traversal (on call-graph) by using the method NodeInfo::getLexicallyEnclosedCallStatements(). For a TranslationUnit, this method recursively calls itself on all the function definitions that are present in the TranslationUnit. In case of any other node, this method collects all CallStatement nodes that are present within the lexical CFG contents of the given node. Additionally, when called on CFG nodes, this method memoizes the return value in the field NodeInfo::callStatements.
- The termination condition always returns false (i.e., we traverse all reachable nodes in the graph).
- For any given node (i.e., a CallStatement) in the graph, we obtain the set of neighbors as a union of returns of NodeInfo::getLexicallyEnclosedCallStatements() when called on all possible function-definitions that may be a target of that CallStatement.

Once the collector returns, we obtain our reachable call-statements as a union of all nodes that have been traversed by the collector (including the start and end nodes).

For each reachable CallStatement, we collect the name of all the symbols that can be the target of that CallStatement, using the method CallStatementInfo::getCalledSymbols(). In this method, firstly, we obtain the function designator cell corresponding to this CallStatement using CallStatementInfo::getFunctionDesignator() (explained towards the end of this section). If the cell is a not a function pointer, we directly add it to the list of called functions to be returned. Otherwise, we add all the function symbols from the points-to set of the cell to the return list. The return value of this method is memoized as CallStatementInfo::calledFunctions. From this return value, we collect the list of all the names corresponding to the collected function symbols.

Given the set of names of function symbols that can be called via some call-statement reachable from main(), we now proceed with deletion of all those ElementsOfTranslation from the program that correspond to either the declaration or definition of the functions whose name does not appear in the set, as follows. For every ExternalDeclaration in the program that encloses a FunctionDefinition, we check whether the function name is present in the collected set or not; if not, we obtain the enclosing ElementsOfTranslation, and remove it from the TranslationUnit. Similarly, if an ExternalDeclaration encloses a Declaration which defines a function symbol with name that is not present in the collected set, we remove the enclosing ElementsOfTranslation.
Collecting function designator cell : To obtain the function designator cell using CallStatementInfo::getFunctionDesignator(), we proceed as follows : Given the function designator name, we obtain the corresponding free-variable or symbol using Misc.getSymbolOrFreeEntry(). If the obtained cell is a symbol, it is returned as it is. However, if the obtained cell is a free-variable, we return null. Note that the returned cell is memoized as CallStatementInfo::functionDesignatorCell.

15.2 Removing unused variables

In order to remove unnecessary variables from within a given node, we use the method NodeInfo::removeUnusedVariables(). First step in this process is to obtain a set of variables that have been *used* lexically (i.e., read from or written to, directly via their names), anywhere within the node. Note that while we consider all declarations to be writes, we do not assume that to be the case here. To obtain this set, we use NodeInfo::getUsedCells(), which internally calls UsedCellsGetter::getUsedCells(). This method, in turn, calls the visitor UsedCellsGetter.AccessGetter on all lexically enclosed CFG leaf nodes, individually. The visitor works as follows : Each visit in this visitor collects the set of those Symbols/FreeVariables which are *used* within the visited node, and adds the collected set to the field cellAccessSet, while returning the set of those Symbols/FreeVariables, if any, that the visited node may represent; UsedCellsGetter::getUsedCells() reads from the field cellAccessSet, and takes a union of it with the set returned by the visitor, to obtain the return value.

Now, given the set of used cells, we process all the scopes that are lexically present within the given node (including the node if it is of type Scopeable), as follows :

- If the scope is a FunctionDefinition, we do nothing. Currently, we do not remove unused parameters.
- If the scope is a CompoundStatement, we obtain a set of all the symbols that are present in the symbol table. For each symbol that is not present in the set of used cells, we obtain the corresponding declaration (using Symbol::getDeclaringNode()), and remove that declaration using CompoundStatementCFGInfo::removeDeclaration(). Before that, if there exists any initializer in the declaration (checked using DeclarationInfo::getInitializer()) then we create a new ExpressionStatement with that initializer as the Expression. We insert this newly created statement at the index at which the declaration was present (using CompoundStatementCFGInfo::addStatement()).
- When the scope is a TranslationUnit, we collect the set of symbols from its symbol table. For each symbol that is not present in the set of used cells, we obtain the corresponding declaration, and remove the enclosing ElementsOfTranslation. Since the method that removes declaration from the TranslationUnit does not trigger automated update of program abstractions, we perform changes in the symbol table using RootInfo::removeDeclarationEffects().

Note 15.2.1

Since the removal of declarations from TranslationUnit does NOT update the semantics of any program abstractions (except AST, and symbol/type/typedef tables) automatically, the prepass phase should be used in a different invocation of IMOP than the actual phase.

15.3 Removing unused types

In order to remove declarations for user-defined types, like structs, unions, or enums, that have not been used anywhere, we use the method NodeInfo::removeUnusedTypes(). Note that the removal of a type may render some other types unused; hence, the removal is done iteratively, until a fixed-point is reached.

As in the case of removal of unused variables, we first collect the set of those types which have been *used* anywhere within the given node, by invoking the method NodeInfo::getUsedTypes(). In this method, we process each scope present within the given node as follows :

- If the scope is a TranslationUnit or a CompoundStatement, we read all the symbols from the symbol table, and typedefs from the typedef table, of the scope. On the type of each symbol, we invoke the method Type::getAllTypes() to obtain the set of types that have been used in the

declaration of that type (including the type itself). The types obtained this way are added to the set of used types.
- If the scope is a FunctionDefinition, we perform the similar processing for all elements of its symbol table (i.e., on all the parameters). Furthermore, we also add the return of getAllTypes() when called on the return type of the function, to the set of used types.

After processing all the scopes, we traverse again on the node (assuming that it might be a statement or an expression), to collect the set of named types that may have been used in the `sizeof` operator, or in cast expression. All the constituent types of these types are also added to the set of used types.

Next, given the set of used types, we process all the scopes that are nested lexically within the given node (including the node itself, if it is of type Scopeable), as follows :

- Nothing is done for the case when the scope is a FunctionDefinition.
- If the scope is a TranslationUnit, we traverse over all the user-defined types stored in the type table of the scope, and obtain their corresponding declarations. If the type is complete *(Question : Is the removal of incomplete types incorrect?)* and not present in the set of used types, we obtain the enclosing ElementsOfTranslation and remove it from the program. Note that this step is followed by a call to RootInfo::removeTypeDeclarationEffects(), which removes the type from the type table.

Note 15.3.1

The method RootInfo::removeTypeDeclarationEffects() does not guarantee automated update of all program abstractions.

- When the scope is CompoundStatement, we find declarations of unused types from the type table and remove them in a similar fashion, using CompoundStatementCFGInfo::removeDeclaration().

15.4 Removing unused typedefs

We use the method NodeInfo::removeUnusedTypedefs() to remove all unused typedefs from the program. Note that the removal of one typedef may render some other typedef unused. Hence, we call this method iteratively until fixed-point is reached for removal of typedefs. (As mentioned before in this section, after reaching the fixed-point of removal of typedefs and of types individually, we need to reach the fixed-point for removal of both of them.)

To obtain the set of typedefs that have been used, we use a visitor UsedTypedefGetter via NodeInfo::getUnusedTypedefs(), which simply collects all the typedefs corresponding to each visited TypedefName (obtained using Misc.getTypedefEntry()) under the given node.

After collecting all the used typedefs, we process each scope lexically nested within the given node (including the node itself, if applicable) as follows :

- For a FunctionDefinition, nothing needs to be done.
- In case of a TranslationUnit, or a CompoundStatement, we traverse over all the typedefs from the typedef table, and find and remove their declarations if the typedefs are not present in the set of used types. As before, we use the methods CompoundStatementCFGInfo::removeDeclaration(), and RootInfo::removeDeclarationEffects() for this purpose.

16 INCOMPATIBLE TYPE-CAST ON POINTERS

In IMOP, the field-sensitivity dimension for any analysis assumes that there are no incompatible type-casting of pointers. For example, there should not be any type-cast from a `pointer to pointer to int` to a `pointer to int`. To ensure that field-sensitivity is not enabled when any such type-cast exists in the program, we invoke **FrontEnd.testIncompatibleTypeCasts()** on the parsed snippet or program. This method invokes Type.hasIncompatibleTypeCastOfPointers() on all CastExpressionTyped expressions present within the given node. If any of the invocations return `true`, this method disables field-sensitivity and returns immediately. Of course, this method does not do anything if field-sensitivity is disabled, to begin with.

For a given CastExpressionTyped expression, the method Type.hasIncompatibleTypeCastOfPointers() proceeds as follows :

- Using Misc.getEnclosingBlock(), first of all, the enclosing scope of the expression is obtained. Then, using Type.getTypeTree() (which requires the scope), and Type.getType(), the source and destination types of the cast expression are obtained.
- If the destination type (or the source type) are not ArrayType or PointerType, then this method returns `false`.
- Similarly, if the source type is a `pointer to void`, this method returns `false`. In other words, we ignore any type casts that casts a `(void *)` to any other type. Note that we do not ignore type casts that cast any other type to a `(void *)`, as otherwise with the help of type casting to and from `(void *)`, incompatible casts can be performed.
- Otherwise, this method returns `true` if and only if the source and destination types are not exactly *same*.

17 LAMBDA-BASED GRAPH COLLECTORS

On generic graphs, IMOP provides a number of traversals that can perform specified operations on the visited nodes, and collect some specific nodes, while ensuring termination along cyclic paths.

These collectors, from class **CollectorVisitor**, are used at various places, to specify different kinds of graph traversals, collecting nodes with specific features.

In order to let callers of these methods specify the notion of *neighbours* of a given node (of generic type T) from a generic graph, this class provides a functional interface NeighbourSetGetter<T> (and NeighbourListGetter<T>) which provides a method getImmediateNeighbours(T):Set<T> (and getImmediateNeighbours(T):List<T>) that can be used to obtain a set (or list) of nodes to which an outgoing edge is assumed to exist from the given node, for the purpose of traversals in this invocation of a collector.

Following are some key methods that are provided by the class CollectorVisitor to work on any generic graph :

collectNodeListInGenericGraph() This method takes the following four parameters : (i) startPoints:List<T>, (ii) endPointst:List<T>, (iii) terminatonCheck:Predicate<T>, and (iv) nextLayerFinder:NeighbourListGetter<T>. This method starts the traversal from immediate successors of the given nodes in startPoints, found by invoking nextLayerFinder.getImmediateNeighbours() on a node, and collects all nodes from the traversed paths that terminate on nodes where the Predicate terminationCheck succeeds. Such nodes where the paths end are stored in the argument endPoints, whereas the collected nodes (that do not contain the elements of startPoints unless those elements are encountered during traversals starting from their successors) are returned back from the method.

This method works as follows :

- It maintains two lists – (i) collectedNodeList, which is used to collect the nodes that have been traversed so far, starting with the neighbours of the nodes from startPoints, and (ii) workList, which is used to collect the nodes that need to be traversed. The list collectedNodeList is initialized to an empty list, whereas the list workList is initialized to contain all the elements from startPoints.
- Until the workList gets empty, its first element is removed and processed as follows. For each neighbour of the element, obtained by invoking nextLayerFinder.getImmediateNeighbours() on it, this method invokes terminationCheck() and performs the following actions : if terminationCheck() returns `true`, then the neighbour is added to the endPoints, otherwise, the neighbour is added to the lists collectedNodeList as well as workList, unless it is already present in collectedNodeList.

Note 17.0.1

Note that the nodes from startPoints are not added to collectedNodeList unless they are encountered as neighbours of any traversed nodes, and, of course, the terminationCheck() fails on them.

Once the workList gets empty, the nodes collected in collectedNodeList are returned back.

Note 17.0.2

(i) Since the lambda terminationCheck will essentially be invoked on all nodes in the returned list, as well as on those in endPoints, its definition may also contain some extra processing that needs to be carried out on any of these nodes.
Of course, after the processing, it must return a boolean specifying whether the visited node is an end node (by returning `true`), or whether the traversal should proceed to the neighbours of the node (by returning `false`).

(ii) Since the lambda nextLayerFinder will essentially be invoked on all nodes in the returned list, as well as on those in the startPoints, its definition may also contain some extra processing that needs to be carried out on any of these nodes.
Of course, after the processing, it must return a set of neighbours for the given node.

collectNodeSetInGenericGraph() This method is exactly similar to the method collectNodeListInGenericGraph(), except that its arguments and returns are Sets instead of Lists. Hence, note that the order in which nodes are added to the workList (termed as workSet in this method) need not be the order in which they get processed.

The following methods are specific to the inter-procedural super control-flow graphs of the program/snippets, which are combinations of the control-flow graphs and call graphs.

collectNodesIntraTaskForward() This method is used to collect the set of nodes that are reachable in the super control-flow graph from the given set of nodes, until a specified termination condition is met on each path, while ensuring termination on cyclic paths. It traverses the graph on only *valid paths* (as obtained when the method CFGInfo::getInterProceduralLeafSuccessors(CallStack):NodeWithStack is invoked).

This method takes only three parameters – all that a regular collectNodeSetInGenericGraph() takes, except for the nextLayerFinder parameter. It works as follows :

- First of all, this method creates sets of NodeWithStacks from the given sets of Nodes for startPoints, endPoints, while using empty call-stacks for each NodeWithStack object. It also creates a Predicate<NodeWithStack> from Predicate<Node>.
  Then, it invokes the method collectNodeSetInGenericGraph<NodeWithStack>, while passing the newly created sets/predicate, and a lambda for the nextLayerFinder, which invokes CFGInfo::getInterProceduralLeafSuccessors(CallStack):NodeWithStack on the node of any given NodeWithStack.
- Once the invocation succeeds, this method populates its endPoints parameter, of type Set<Node> using the nodes of the corresponding argument set (of NodeWithStack type) that was passed to the method getInterProceduralLeafSuccessors(CallStack). Similarly, it

also creates the set to be returned using the nodes from the set that was returned from that method.

collectNodesIntraTaskBackward() This method is used to collect the sets of nodes that are reachable upon backward traversal on the super control-flow graph from the given set of nodes, until a specified termination check succeeds on any node, while ensuring termination on cyclic paths.

It is exactly similar to the method collectNodesIntraTaskForward(), except that it uses the method CFGInfo::getInterProceduralLeafPredecessors(CallStack) instead of CFGInfo::getInterProceduralLeafSuccessors(CallStack).

collectNodesIntraTaskForwardContextSensitive() This method serves the same purpose as collectNodesIntraTaskForward(), except that its arguments use the type NodeWithStack instead of Node, (as the element type for Set and Predicate).

Since the arguments are already based on type NodeWithStack, this method directly invokes collectNodeSetInGenericGraph(), passing its three arguments, along with a lambda that invokes CFGInfo::getInterProceduralLeafSuccessors(CallStack):NodeWithStack on any given NodeWithStack.

collectNodesIntraTaskForwardOfSameParLevel() This method is used to confine the traversal within the current ParallelConstruct, without jumping to any nested ParallelConstruct, or to the construct that encloses the current ParallelConstruct.

It is similar to collectNodesIntraTaskForwardContextSensitive(), except that (i) it takes a single NodeWithStack as its first argument, (ii) it uses CFGInfo::getParallelConstructFreeInterProceduralLeafSuccessors(CallStack):NodeWithStack instead of CFGInfo::getInterProceduralLeafSuccessors(CallStack):NodeWithStack. The former lambda differs from the latter as follows :

(i) The successors of the BeginNode of a ParallelConstruct are assumed to be same as the successors of the ParallelConstruct itself. This is done in order to ensure that only the BeginNode of the nested ParallelConstructs are added to the collected set, and the other contents of the nested construct are ignored.

(ii) The EndNode of a ParallelConstruct is assumed to have no successors. Therefore, the traversals do not consider any nodes that are outside the current ParallelConstruct [5].

Note that, as a result, this method restricts the traversal within the same level of the parallel construct as the one in which the first argument exists (except when it is the BeginNode of that parallel construct.

[5]Note that as OpenMP does not allow any jumps outside a ParallelConstruct, we will not have any other exit points for the construct.

Note 17.0.3

When the first argument to collectNodesIntraTaskForwardOfSameParLevel() is a BeginNode of a ParallelConstruct, then the traversal jumps outside the ParallelConstruct, skipping its contents altogether. Hence, in order to traverse within a ParallelConstruct, one should never give its BeginNode as the first argument to this method.

collectNodesIntraTaskForwardBarrierFreePath() . This method takes only two arguments : (i) a startPoint, from where the traversal has to start, and (ii) a set of endPoints, where the traversal would terminate (this set would be filled by the method). The traversal terminates on a path when a BarrierDirective, or EndNode of a ParallelConstruct are encountered. This method is used to obtain a set of nodes that are reachable on barrier-free path traversals from the given nodes. It works as follows :

- This method invokes collectNodeSetInGenericGraph() with following arguments :
  (i) a singleton set, comprising of the first argument, startPoint,
  (ii) the second argument of this method, endPoints,
  (iii) a lambda that returns, for any node, `true` only when the node is a BarrierDirective, or EndNode of a ParallelConstruct,
  (iv) a lambda that returns, for any node, the return of CFGInfo::getParallelConstructFreeInterProceduralLeafSuccessors(CallStack):NodeWithStack when invoked on the given node.
- Once the internally invoked method terminates, the set of NodeWithStacks that have to be returned is created as follows : This set comprises of each element which is returned by the internally invoked method. Furthermore, corresponding to each BeginNode of a ParallelConstruct that is encountered, we also add all the leaf CFG contents of that ParallelConstruct (collected using CFGInfo::getIntraTaskCFGLeafContents()) to the set to be returned. Note that all such ParallelConstructs are nested parallel constructs. Finally, this set is returned back to the caller.

Note that other symmetric versions of some of these methods exist, such as **collectNodesIntraTaskBackwardContextSensitive()**, and **collectNodesIntraTaskBackwardBarrierFreePath()**, which haven't yet been used anywhere; hence, we do not lay out their workings here.

18 INITIALIZATION OF DUMMY FLUSHES

In OpenMP, implicit flushes exist at the entry to, and/or exit from, various constructs or directives. Furthermore, users can specify flushes explicitly. In order to ease the task of handling flushes in

various analyses/transformations, we make the implicit flushes explicit, by representing all types of flushes uniformly with DummyFlushDirective.

Note 18.0.1

Note that in the source code, all DummyFlushDirectives appear as comments.

Refer to the preliminary technical report of IMOP for a detailed list of places where a DummyFlushDirective is inserted.

While normalizing a newly parsed program or snippet, we invoke **CompoundStatementCFGInfo::initializeDummyFlushes()** on all nested CompoundStatement. This method takes each element of the given compound-statement, and passes it as an argument to CompoundStatementCFGInfo::insertNewDFDsWithoutNode(), which is used to insert the missing dummy-flushes around the given element, as follows : For the given node, we check whether a DummyFlushDirective of appropriate DummyFlushType is present as the predecessor and/or successor of the node, as per the placement rules of DummyFlushDirective, given in the preliminary technical report on IMOP. These checks are performed using InsertDummyFlushDirective.hasPredDFD() and InsertDummyFlushDirective.hasSuccDFD(). If the required DummyFlushDirective is missing, we create one, of appropriate DummyFlushType, and insert it above/below the given node using CompoundStatementCFGInfo::commonNodeAdditionModule(), at appropriate index.

19 MHP ANALYSIS, AND INTER-TASK DATA-FLOW GRAPH

19.1 Data structures

First of all, let's look into the list of data structures that together represent the MHP information :

- **Phase::parConstruct** refers to the parallel-construct in which the receiver phase may get executed. For a given parallel-construct, the corresponding field **ParallelConstructInfo::allPhaseList** contains the list of all the phases that may get executed within that parallel-construct.
- Other relevant fields of a phase are as follows : (i) **nodeSet:HashSet<Node>**, the set of nodes which may get executed in this phase, (ii) **beginPoints:HashSet<BeginPhasePoint>**, the set of starting points for this phase, (iii) **endPoints:HashSet<EndPhasePoint>**, the set of ending points for this phase, (iv) **succPhase:Phase**, the phase which will be executed after this phase, (v) **predPhases:ArrayList<Phase>**, the list of phases that may get executed immediately before the execution of this phase, and (vi) **phaseId:int**, a unique identifier for this phase.
- Corresponding to a CFG leaf node, following two elements in its associated NodePhaseInfo object denote the MHP information : (i) **phaseSet:HashSet<Phase>**, the set of phases in

which this node may get executed as per the current state of the program, and (ii) **inputPhaseSet:HashSet<Phase>**, the set of phases in which this node may have executed, in the input program (before any transformations).
- A PhasePoint is a 2-tuple, of a Node and a CallStack (which comprises of a Stack of CallStatements).
- Each phase starts at a starting point, referred as BeginPhasePoint (a subtype of PhasePoint). A BeginPhasePoint comprises of the following relevant fields : (i) **reachableNodeSet:HashSet<Node>**, the set of CFG leaf nodes that are reachable on barrier-free paths from this point, (ii) **nextBarrierSet:HashSet<EndPhasePoint>**, the set of ending points corresponding to traversals starting at this point, and (iii) **phaseSet:HashSet<Phase>**, the set of phases which may start at this point.
- We also maintain the set of all the starting points in a set **BeginPhasePoint.allBeginPhasePoints:HashSet<BeginPhasePoint>**.
- At times, a starting point can be marked as *invalid* (by setting the field **setsInvalid**), if its relevant data structures (described above) might not contain correct values.
- In order to maintain the set of all those starting points that might not contain valid information, we use a static set, termed as **BeginPhasePoint.staleBeginPhasePoints:HashSet<BeginPhasePoint>**. Note that as a result of program transformation, this set might contain certain elements which are not connected to the main AST.

Closely related to notion of MHP information, is that of *inter-task communication edges*, which are formed between the dummy-flush directives that may share a phase. An inter-task communication edge is represented by the class InterTaskEdge, which comprises of the following two fields: (i) sourceNode:DummyFlushDirective, which represents the source node, and (ii) destinationNode:DummyFlushDirective, which represents the destination node. Given a dummy-flush, following fields in its info object represent inter-task edges :

- **DummyFlushDirective::incomingInterTaskEdges:HashSet<InterTaskEdge>**, which represents the edges via which communication can happen from some other dummy-flush to this dummy-flush.
- **DummyFlushDirective::outgoingInterTaskEdges:HashSet<InterTaskEdge>**, which represents the edges via which communication can happen to some other dummy-flush from this dummy-flush.

Note that the method DummyFlushDirectiveInfo::getInterTaskDummyPredecessors(): HashSet<DummyFlushDirective> reads from the field incomingInterTaskEdges:HashSet<InterTaskEdge>, and returns a set of all those dummy-flushes from which at least one shared variable may get communicated to this dummy-flush, via an inter-task edge. Similarly, the method DummyFlushDirectiveInfo::getInterTaskDummySuccessors():HashSet<DummyFlushDirective> reads from the field outgoingInterTaskEdges:HashSet<InterTaskEdge>, and returns a set of all those dummy-flushes to which at least one shared variable may get communicated from this dummy-flush, via an inter-task edge. ***UPDATE:*** **"Also, note that if the flag Program.preciseDFDEdges is not set, then the set of successors and predecessors of a DummyFlushDirective would contain all other DummyFlushDirective nodes of any common phases, regardless of whether a shared variable may get communicated from source DummyFlushDirective to destination DummyFlushDirective."**

19.2 Initialization of MHP information

Entry point for the initialization of MHP analysis is a call to **Misc.performMHPAnalysis(Node)**, made while processing parallelism-related analyses in the normalization step of the front-end (using FrontEnd.processParallelism()) when parsing the complete program. While parsing a snippet, we directly invoke **MHPAnalyzer::initMHP()** on all the internal ParallelConstructs. This is followed by calls to NodePhaseInfo::rememberCurrentPhases() on all CFG leaf nodes lexically contained within the parsed snippet. This method is used to *remember* the current phases in which a given node may get executed; this information is saved in the field NodePhaseInfo::inputPhaseSet.

In Misc.performMHPAnalysis(), we call MHPAnalyzer::initMHP() for each ParallelConstruct in the passed node (root node in this case); the list of ParallelConstruct, nested or otherwise, is obtained using a visitor InfiParallelConstructGetter. As the second and last step, this method traverses through all the CFG leaf nodes lexically contained within each FunctionDefinition, and invokes NodePhaseInfo::rememberCurrentPhases().

Each MHPAnalyzer::initMHP() call corresponds to a certain ParallelConstruct node. Some key points to observe concerning this method :

- Before populating the parallel construct with new phases, this method removes the existing phases as follows : For each CFG leaf node reachable from within the parallel construct, and each existing phase of the parallel construct, this method calls NodePhaseInfo::removePhase() to remove the phase from the node. After this, the field ParallelConstructInfo::allPhaseList is set to an empty list.

The method NodePhaseInfo::removePhase() removes the provided phase from NodePhaseInfo::phaseSet; if the set Phase::nodeSet of the given phase contains the receiver node, then the node is removed from that set, via a call to Phase::removeNode().

During removal of the given phase from the node, we also perform the following additional step, if the node is a DummyFlushDirective : We remove all those incoming and outgoing inter-task edges for the given node which connect the node to some other node (DummyFlushDirective), such that the other node shared only one phase with the given node – the phase that has been removed.

- Using the method MHPAnalyzer::shouldProceedWithMHP(), the method MHPAnalyzer::initMHP() decides whether it should create a new Phase and invoke MHPAnalyzer::processNextPhase() on it or not. In the constructor of Phase, the newly created phase gets automatically added to ParallelConstructInfo::allPhaseList. Once MHPAnalyzer::processNextPhase() returns, this whole process is repeated until MHPAnalyzer::shouldProceedWithMHP() returns `false`.

The method **MHPAnalyzer::shouldProceedWithMHP()** works as follows :

- If this method is called when there are no phases in allPhaseList, it returns true, i.e., we proceed with the marking of nodes with phases, if the parallel construct does not contain any phase.
- Otherwise, this method first obtains a sub-list of EndPoint from the endPoints of the last phase in allPhaseList, removing all those EndPoints that correspond to the EndNode of the parallel-construct being processed. If the entries in the obtained list are a subset of the beginPoints of some pre-existing phase, then we connect the last phase to that phase, and return false. Note that the creation of further phases will not lead to any discovery of new MHP relations.

Here are some key points to note when **MHPAnalyzer::processNextPhase()** is called with a given phase :

- Note that before this method is called on a phase, the phase is added at the end of allPhaseList. Hence, this method assumes that the provided phase is the phase to be executed after the second-last phase in allPhaseList. If allPhaseList does not contain more than one element (i.e., this phase is the only element in that list), then the phase is the first phase to be executed in the associated parallel construct. Otherwise, using Phase.connectPhases() this phase is set as the successor of the last phase, and last phase is set as one of the predecessors of this phase.
- In this method, a list of BeginPhasePoint, named beginPoints is used to store those phase points which correspond to the end-points of the previous phase, if any. Whereas, another list, startPoints, of elements of type NodeWithStack, is used to store those elements starting which phase marking has to be done for the given phase.

- If this phase is the only phase in allPhaseList, then a BeginPhasePoint is obtained using the factory method BeginPhasePoint.getBeginPhasePoint() (explained later in this section), while passing BeginNode of the parallel-construct, an empty call-stack, and this phase, as the arguments. The list startPoints is same as beginPoints, in this case.
- When there exists a last phase for the given phase, we take each EndPhasePoint of the last phase to obtain the corresponding BeginPhasePoint. If the EndPhasePoint corresponds to the EndNode of the parallel construct, then we ignore it. Otherwise, we use BeginPhasePoint.getBeginPhasePoint() to obtain the desired BeginPhasePoint, by passing the following three arguments : the node of the end phase-point, call-stack of the end phase-point, and this phase.

  Note that startPoints in this case differs from beginPoints (i.e., when this phase is not the first phase within the parallel construct). It contains context-sensitive inter-procedural leaf successors of each end phase-point.
- Now, the obtained beginPoints is used to populate the field beginPoints of this phase.
- Next, we invoke the visitor **ParallelPhaseMarker** on all elements in startPoints (context-sensitively), which works as follows:
  - In this visitor, each visit method takes a CallStack as an argument, denoting the call-stack with which the node is being visited.
  - This visitor maintains a map visitedMap:Map<Node, Set<CallStack» which is used to map a node to all those call-stacks with which the node has been visited so far; this data structure is used to ensure termination of the marking process.
  - For all types of CFG nodes, except for ParallelConstruct, BarrierDirective, PreCallNode, and EndNode, the following process is carried out (via a call to initProcess) in the visits : If the visited node is a non-leaf CFG node, the visitor is called on the BeginNode of that non-leaf node; in case of a leaf CFG node, if marking of phase via addPhase() is successful, this visitor is called on all context-sensitive inter-procedural leaf successors of the leaf node, one-by-one. The method addPhase() takes two arguments – node to be marked with the given phase, and the call-stack with which the node has been visited. If the given node has already been visited with the given call-stack earlier, this method returns `false`. Otherwise, it adds the given call-stack to the set of call-stacks corresponding to the given node, in the map visitedMap, adds this node to the current phase (and phase to the current node) using the method Phase::addNode(), and returns `true`. (Note that inter-task edges are created as well, via the call to Phase::addNode(); for details, refer to Section 19.3.)
  - If the visited node is a BarrierDirective, the current phase is added to the node via addPhase(). Furthermore, the node, and current call-stack, is added as an EndPhasePoint to

the current phase, using Phase::addEndPointNoUpdate(). The traversal does not proceed to the successors of this node.
- If a ParallelConstruct is encountered during the traversal, then it would refer to a nested parallel construct. In that case, the current phase is added to all the intra-task leaf nodes that are reachable within the visited parallel construct. Then, the visitor is called on all the context-sensitive inter-procedural leaf successors of the parallel construct.
- When the visited node is a PreCallNode, we first check whether the corresponding FunctionDefinition(s) exists. If not, then we process this node as any other node, as mentioned above. Otherwise, we mark this node with the current phase, using addPhase(), and then call this visitor on the target FunctionDefinition, with the modified call-stack, obtained by pushing the call-statement corresponding to this PreCallNode on top of the current call-stack.
- Upon visiting an EndNode, this visitor marks the node with the current phase, using addPhase(). If the marking method returns `false`, then the we return from this visit. Otherwise, except when this node is an EndNode of a ParallelConstruct which is same as the parallel construct of the current phase, or of a FunctionDefinition, we simply call the visitor on all the context-sensitive inter-procedural leaf successors of the node. When the visited node is an EndNode of a ParallelConstruct of which the current phase is a part, then we add this node and the current call-stack as an EndPoint to the set of endPoints of the current phase. When this EndNode is that of a FunctionDefinition, we need to readjust the call-stack, by popping the top element, before continuing with the traversal. If the top of the call-stack is a context-insensitivity marker (represented by CallStatmenet.getPhantomCall()), then we collect all possible call-sites of the FunctionDefinition (using FunctionDefinitionInfo::getCallersOfThis()), and call this visitor on the PostCallNode of the call-site, with the unchanged call-stack. On the other hand, if the top of the call-stack is not a context-insensitivity marker, we remove the top element of the call-stack, and using the unwinded call-stack call the visitor on the PostCallNode of the popped CallStatement.

Obtaining a BeginPhasePoint: Only factory methods, with name **BeginPhasePoint.getBeingPhasePoint()** exist for obtaining a BeginPhasePoint; all constructors are `private`. Internally, a list of all BeginPhasePoint objects is maintained in the field BeginPhasePoint.allBeginPhasePoints, which is used to ensure that corresponding to each unique pair of node and call-stack, there will be only one BeginPhasePoint. Hence, given a node and call-stack, this method first checks if the corresponding BeginPhasePoint exists. If so, then it is selected to be returned. Otherwise, it obtains the BeginPhasePoint to be returned by calling its constructor which also adds the newly created object to allBeginPhasePoints. Finally, before returning the selected BeginPhasePoint, this method adds the given phase to its field phaseSet.

19.3 Initialization of inter-task data-flow graph

Inter-task data-flow graph is composed up of inter-task communication edges, represented as InterTaskEdge. In Section 19.2, when ParallelPhaseMarker::addPhase() is called on a node, it calls Phase::addNode(). This method, in turn, invokes NodePhaseInfo::addPhase() on the node being marked.

Given a phase, the method NodePhaseInfo::addPhase() adds that phase to the phaseSet of the receiver node, if not already present. (Note that the actual receiver here is the phase-info object corresponding to the node of interest.) If the node is a DummyFlushDirective, we iterate over all the DummyFlushDirective in the given phase, and invoke method **DataFlowGraph.createEdgeBetween()** for the given node and iterated node, which invokes DataFlowGraph.createInterTaskEdgeBetween(), in turn. This method creates two inter-task edges – (i) an edge from first node to the second, and (ii) an edge from second node to the first, and adds these edges to the fields incomingInterTaskEdges and outgoingInterTaskEdges of both the nodes, appropriately.

From FrontEnd.processParallelism() while parsing the complete program, and from FrontEnd.parseAndNormalize() while parsing a snippet, we invoke (unnecessarily, it seems) Misc.createDataFlowGraph(). This method invokes DataFlowGraph.populateInterTaskEdges() on all phases of all parallel constructs, which, in turn, invokes DataFlowGraph.createEdgeBetween() on all pairs of DummyFlushDirective nodes that may get executed in the given phase.

20 GENERIC ITERATIVE FLOW ANALYSIS

IMOP provides various kinds of generic iterative flow analyses, along with some of their instantiations that implement certain standard flow analyses, such as *points-to* analysis, *dominance* analysis, etc.

At the root of all the flow analyses, lies **FlowAnalysis**. A FlowAnalysis can be either a DataFlowAnalysis, or a ControlFlowAnalysis [6]. When the values of flow facts corresponding to an iterative flow analysis (IFA) are dependent upon the contents of the node, then we categorize such iterative *data* flow analyses (IDFAs) as DataFlowAnalysis. Whereas, when the flow-facts of analyses are dependent only upon the structure of the flow graph, then we categorize such analyses as ControlFlowAnalysis.

In IMOP, a ControlFlowAnalysis is always intra-thread in nature. The inter-thread edges, which connect DummyFlushDirectives within any phase, represent the flow of shared *data* from one task to another. Hence, such edges are considered non-existent in all *control*-flow analyses. There

[6]Note that we misuse the phrase *control flow*. In IMOP, it does not refer specifically to call-graph construction.

are two subclasses of ControlFlowAnalysis – (i) the intra-procedural variant, termed as IntraProceduralControlFlowAnalysis, and (ii) the inter-procedural variant, termed as InterProceduralControlFlowAnalysis. Both these subclasses model *flow-sensitive* analyses. One key instantiation of IntraProceduralControlFlowAnalysis is PredicateAnalysis, and that of InterProceduralControlFlowAnalysis is DominanceAnalysis. Both these instantiations are explained in Section 22.
A DataFlowAnalysis can be of two types :

(i) CellularDataFlowAnalysis is a superclass for those analyses in which the structure of a flow-fact is a map from a set of Cells to a set of some Immutable values. There are certain scope-specific optimizations that are applicable only to the analyses that fall under this category.
(ii) NonCellularDataFlowAnalysis is a superclass for all other data-flow analyses.

Both these subclasses of DataFlowAnalysis are further categorized into two subclasses each – one for forward analyses, and another for backward. All DataFlowAnalysis are inter-thread in nature, in order to respect the semantics of OpenMP. For precision, currently IMOP provides only inter-procedural flow-sensitive versions of these analyses. Various example instantiations of different kinds of DataFlowAnalysis are illustrated later in this section.

We first look at the structure of a generic flow fact, (FlowFact), which corresponds to any FlowAnalysis, as well as its important subclass CellularFlowMap, which applies to all instances of CellularDataFlowAnalysis. This is followed by a discussion on various kinds of generic passes mentioned above, along with their supporting data structures.

In Section 22, we discuss the steps for instantiating any generic flow pass, along with certain important instantiations, such as points-to analysis, copy-propagation analysis, reaching-definitions analysis, etc.

The type of a flow fact is taken as a type argument, while instantiating any generic flow analysis. This type should (directly or indirectly) extend the class FlowFact.

20.1 Generic flow facts

All concrete subclasses of **FlowFact** must define the following methods :

FlowFact::isEqualTo(other:FlowFact):boolean takes a flow-fact, and should return *true* if and only if that flow-fact is semantically equivalent to the receiver flow-fact. This method is employed to check whether a fixed-point has been reached. Hence, termination can be ensured only when this method has been correctly defined.

FlowFact::getString():String should return the String equivalent of the receiver flow-fact. This method is used to dump information about the flow-facts, in the form of comments in the output program, for each leaf node.

FlowFact::merge(other:FlowFact, cellSet:CellSet) takes the given flow-fact other, and performs meet of that flow-fact with the receiver flow-fact. Note that this method has side-effects since it changes the receiver flow-fact, such that it starts representing the result of the meet operation. The argument cellSet, when null or empty, is ignored. However, when this set contains any elements, they are usually the shared cells which can get communicated from a predecessor DummyFlushDirective to the processed node (which itself is a DummyFlushDirective), via inter-task data-flow edges. This set can therefore be used to ensure that those components of data-flow facts which correspond to private variables at the processed node, do not get affected by the data-flow facts corresponding to private variables at any of the node's inter-task predecessors.

This method returns *true*, if the receiver flow-fact was changed as a result of the merge.

While the structure of a FlowFact can be of any type, there exists a specialized subclass of FlowFact, named CellularFlowMap<V extends Immutable>, where the data-flow information is represented as a map from set of Cells to set of some generic *immutable* type (V). Most of the important IDFA analyses, like points-to (PointsToAnalysis), reaching definitions (ReachingDefinitionAnalysis), and copy propagation (CopyPropagationAnalysis), have flow-facts that are inherited from CellularFlowMap.
Following are some key points to note about **CellularFlowMap** :

- Each CellularFlowMap contains an internal map from set of Cells to set of some generic immutable type (say V). The objects of type V should correspond to various elements of the data-flow lattice. This map is of type ExtensibleCellMap<V>. (Refer Section 20.4 for details on internal workings of an ExtensibleCellMap<V>.)
- No subclasses of ExtensibleCellMap can override the methods isEqualTo(), getString(), or merge() (which were the abstract methods defined by FlowFact). The implementations of these methods is provided by ExtensibleCellMap, as follows :

  CellularFlowMap::isEqualTo(other:FlowFact):boolean returns the equality of the flow maps present in the receiver and the other CellularFlowMaps (using ExtensibleCellMap::equals()).

  CellularFlowMap::merge(other:FlowFact, cellSet:CellSet) utilizes ExtensibleCellMap::mergeWith() to update the flow map of the receiver such that it reflects the result of meet of the receiver's flow-fact with that of the other. The merge operator required in the mergeWith() method can be provided by subclasses of CellularFlowMap by overriding the abstract method CellularFlowMap::meet(V, V):V. This meet() method should model the meet operation of the data-flow lattice.

CellularFlowMap:getString():String method returns the string representing the values stored in flow map of the receiver. This method uses CellularFlowMap::getAnalysisNameKey(), an abstract method, to tag the generated string of the data-flow fact with a short string that denotes the corresponding data-flow analysis. For example, for data-flow facts of points-to analysis, the short string is *ptsTo*.

- Following are the two methods that must be implemented by any concrete subclasses of CellularFlowMap :

  CellularFlowMap::getAnalysisNameKey():String should be overridden by the concrete subclasses of CellularFlowMap, returning a short string that can be used to identify the data-flow analysis which this data-flow fact corresponds to. This string is used in CellularFlowMap::getString() method, for getting the string used for debugging purposes.

  CellularFlowMap::meet(v1:V, v2:V):V is used to model the meet operation, given two elements of the data-flow lattice corresponding to this flow-fact. Note that both the arguments to this method are immutable. Hence, returning (thereby reusing) any of the provided arguments should not create any correctness issues.

20.2 Base generic flow analysis pass

In this section, we look into how various generic IFA passes work. The complete code is present in the class hierarchy rooted under FlowAnalysis. Each flow analysis must inherit directly or indirectly from FlowAnalysis. Note that this analysis can be run in one of the two modes – (i) *no-update* mode, which denotes the first run of the analysis on the program, starting at entry point of the `main()` function, or (ii) *update* mode, which is used during automated incremental update of the IDFA flow-facts, under elementary transformations of the program. (We discuss the *update* mode later in Section 28.)

During the *no-update* mode, the analysis maintains the following internal data structures :

- analysisName :AnalysisName refers to an enumerator constant that is specific to each kind of analysis. For example, points-to analysis is denoted by the constant AnalysisName.POINTSTO.
- analysisDimension:AnalysisDimension specifies the analysis dimensions, such as whether the analysis is sensitive or insensitive along FlowDimension, FieldDimension, ContextDimension, SVEDimension, etc.
- A list, workList, of type ReversePostOrderWorkList, which is used to maintain a list of nodes that need to be (re)processed for reaching the fixed-point during computation of the analysis.
- For debugging and profiling purposes, each IDFA analysis maintains the following information :
  (i) nodesProcessed:long, which keeps track of the total number of times any nodes were processed to reach the fixed-point in any of the modes (*update* or *no-update*),

(ii) a map, tempMap, which, for each node, individually keeps track of the number of times the node was processed by this analysis (in either of the modes); if this number crosses a threshold (denoted by Program.thresholdIDFAProcessingCount), then the framework throws an error and exits, (Default value for this threshold has been randomly set to 7e5.)

Finally, a static field analysisSet:Map<AnalysisName, FlowAnalysis<?>> of the generic IDFA pass is used to maintain a set of all the instances of FlowAnalysis<?> that have been created so far in the framework, mapped using the enumerator constants of type AnalysisName.

The no-update mode. In this mode, each concrete subclass of FlowAnalysis should implement (or inherit), the following three abstract methods :

- **run(FunctionDefinition):void**, which takes a FunctionDefinition and runs flow analysis on it, and on its other reachable methods, either intra-procedurally or inter-procedurally, depending upon the nature of the analysis. The usual argument to this method is `main()`.
- **getTop():F** should return a new object upon each invocation, representing the top element of the lattice.
- **getEntryFact():F**, should return a new object upon each invocation, representing the initial flow-fact, which can be given as initial :
  - IN flow-fact for first element of `main()`, for forward inter-procedural analyses,
  - IN flow-fact for first elements of all functions, for forward intra-procedural analyses,
  - OUT flow-fact for last element of `main()`, for backward inter-procedural analyses, and
  - OUT flow-fact for last elements of all functions, for backward intra-procedural analyses.

The default visit methods : We utilize various visit() methods to model the transfer functions of nodes, which specify how the states of a flow-fact undergoes transition from the entry state for node (IN or OUT) to the exit state (OUT or IN). About their default implementation in FlowAnalysis, following are the points to note :

- We maintain flow facts only at the level of leaf nodes, and not non-leaf nodes. Hence, the *final* implementation of the visits for non-leaf nodes throws an AssertionError.
- In the visit of every leaf CFG node, the argument flow-fact is passed to the common method FlowAnalysis::initProcess(Node, F):F; the return of this invocation is returned back from the visit method. The default implementation of the method initProcess() simply returns its arguments.

Note 20.2.1

Note that if neither the visit method of a specific node, nor initProcess() have been overridden by an analysis, then the transfer function for that type of node is considered to be an identity function for that analysis.

The edge-transfer function. While processing the successors of a predicate in forward analyses, and predicates in backward analyses, one might need to model the effects of taking a branch, on a flow-fact. In order to facilitate such operations, FlowAnalysis provides the method edgeTransferFunction(F, Node, Node):F, which can be overridden as per the analyses, such that, given the meet of IN of predecessors (in forward analyses), and the predecessor-successor pair representing an edge, the method should return another flow-fact which models the effect of the edge on the input flow-fact. (For backward analyses, the edge effects are modelled for meet of OUT of successors). The default implementation of this method assumes that edges do not affect the flow-facts.

20.3 Specialized generic flow passes

In this section, we look into implementation of some important methods in generic subclasses of FlowAnalysis.

The driver run() method. Given a function-definition as argument, the method FlowAnalysis<?>::run() performs flow analysis starting with that function, until fixed-point is reached. In case of intra-procedural analyses, all reachable function-definitions from the given definition are processed explicitly in run(). Whereas, in case of inter-procedural analyses, only the given function-definition is processed explicitly. (All the reachable methods get processed implicitly.)

For forward analyses, while processing a function definition, the workList of nodes to be processed is initialized with the BeginNode of that function definition. In case of backward analyses, the workList is initialized with the EndNode of the given function-definition.

For each element of the workList, this method invokes processWhenNotUpdated() (i.e., in *no-update* mode), to apply data-flow equations in the flow-facts maintained at the node. The elements of the workList are processed as per the reverse postorder in which these elements appear in the phase-flow graph and control-flow graph of the program. For backward analyses, we process the nodes as per their postorder. (Refer Section 20.5 for more details.) Note that new elements may get added to the workList during processing of any element. The method run() terminates only once there are no more elements to be processed in the workList. In case of InterThreadForwardCellularAnalysis, just before termination, this method sets PointsToAnalysis.stateOfPoinstTo as CORRECT, if the receiver object is an instance of the standard points-to analysis used by IMOP (referred by the analysis name AnalysisName.POINTSTO).

Note 20.3.1

In the discussion that follows, we assume that the direction of flow analysis is *forward* in nature, unless otherwise stated.

For backward analyses, the discussion would remain same, except that *IN* and *OUT* will get interchanged, as would *successors* and *predecessors*.

Processing each node, in *no-update* mode. The method processWhenNotUpdated() processes one node at a time, by applying the data-flow equations. It works as follows :

- The current IN data-flow fact at the given node is obtained via a call to NodeInfo::getInfo(). If the return value is `null`, then it implies that this node has not been processed by this analysis yet. In such a case, we initialize the IN flow-fact by (i) getEntryFact(), if this node does not have any immediate predecessor node (e.g., the `BeginNode` of the `main()`'s FunctionDefinition), OR (ii) getTop(), otherwise. The predecessors are obtained via CFGInfo::getInterTaskLeafPredecessorEdges() for inter-procedural analyses; for intra-procedural analyses, we use CFGInfo::getLeafPredecessors(). Furthermore, when the current

Note 20.3.2

In case of backward analyses, as mentioned before, we replace *IN* with *OUT*, *OUT* with *IN*, *successor* with *predecessor*, and *predecessor* with *successor*, while reading this section.

Note 20.3.3

Note that getEntryFact() and getTop() should not return `null`. Also, they must always return a new object; otherwise, the IN flow-facts of multiple nodes might start referring to the same object.

IN data-flow fact is `null`, we ensure that all the successors of this node will get added into the workList, so that all the nodes reachable from entry-point of the program are processed at least once. We do so by setting a boolean flag propagateFurther.

Note that if the current IN is not `null`, then we do reuse the same object to obtain the new value (if any) for IN.

- Now, the current IN flow fact of the node being processed is merged, one-by-one with the non-`null` OUT of each of its predecessors. If a predecessor has not been processed yet, its OUT would be `null`. Assuming that all OUT flow-facts have been initialized to TOP (obtained using getTop()), we can ignore such flow-facts, as a merge with TOP would anyway not affect any data-flow fact.

  We maintain a flag `inChanged` which is set only when the internal state of the IN flow-fact gets updated as a result of these merge operations.

Note 20.3.4

The merge() method should not update the internal state of the data-flow fact that it received as its argument (which may denote OUT of a predecessor, for example).

In order to model the effects of taking a branch, we pass the newly generated flow-fact to edgeTransferFunction(), which should return back the modified flow-fact.

- In case of inter-thread forward cellular data-flow analysis (InterThreadForwardCellularAnalysis), if the node is a PostCallNode, then we perform other scope-specific changes to the flow-fact, by invoking processPostCallNodes(). (This method is explained later.) If the internal state of IN flow-fact has changed as a result of this invocation, then we set the `inChanged` flag.

  Similarly, in case of inter-thread backward cellular data-flow analysis (InterThreadBackwardCellularAnalysis), if the node is a PreCallNode, then we perform other scope-specific changes to the flow-fact, by invoking processPreCallNodes(). (This method is explained later.)
- For any data-flow analysis, if the node is a BarrierDirective, and if the inChanged is set to true, then we invoke addAllSiblingBarriersToWorkList() (to all all sibling barriers to workList). We explain the details and necessity of this step later, while discussing the visit() method for BarrierDirective nodes.
- Finally, we set the newly obtained IN as the current IN of the node, using NodeInfo::setIN(). (This step would make a difference only when the newly obtained IN is a different object than the one obtained at the start of the method processWhenNotUpdated().)
- Next, we fetch the current OUT flow-fact object of the node being processed. In order to obtain the new state of OUT flow-fact, we need to apply the transfer function (flow function) of the node on its new IN flow-fact. The transfer function is modeled by the visit() methods (invoked using accept() methods, as is the norm in visitor design pattern). We discuss various visit() methods in detail later in this section.

Note 20.3.5

Note that a visit() method may return the same object which it obtained as its argument. That is, the IN and OUT flow-facts of a node may be represented by the same Java object. However, a visit() must never update the internal state of its argument (which represents the current IN flow-fact).

In case of a intra-procedural control-flow analysis (IntraProceduralControlFlowAnalysis), if the node is a PreCallNode, we use the method modelCallEffect() on it, instead of any visit method, to model the effect of call to some unknown method. The default implementation of this method simply returns its argument.

After obtaining the new OUT flow-fact, we set it as the current OUT flow-fact of the node.

- For any cellular flow analysis, before setting the new OUT flow-fact as current OUT flow-fact, we invoke processEndNodes() or processBeginNodes() methods if the current node is an EndNode or a BeginNode, respectively. These methods are used to ensure that, for the case of

CellularFlowMaps, we maintain flow-fact information for only those cells that are relevant in the scope in which this node occurs. These methods are explained later in this section.

- Next, we need to decide whether we should add the successors of this node in the workList. We track this decision using a flag propagateFurther. As mentioned above, if the analysis is processing this node for the first time, then we set this flag. Furthermore, we also set this flag if the current IN flow-fact is different from the old IN flow-fact [7] (i.e., if inChanged is `true`, we set propagateFurther as well.)

  If the flag is set, then in case of an intra-procedural control-flow analysis, we simply add the leaf successors of the node to the workList; for inter-procedural control-flow analysis, we add the inter-procedural leaf successors.

  In case of data-flow analyses, when the node is a BarrierDirective, the OUT flow-fact depends not just on the IN flow-fact of the node itself, but also on the IN flow-facts of the sibling barrier nodes. Hence, even when the IN flow-fact of the node does not change, the OUT flow-fact might still change. Hence, if the old OUT flow-fact is not same as the new OUT flow-fact for a barrier node, we set the propagateFurther flag. Note the following observations when comparing the OUT flow-facts (old and new) of a node :
  - If the IN and OUT flow-facts of a node are represented by the same object, then we should consider the new OUT to be different from old OUT whenever the new IN was different from the old IN. This check was already performed during update of IN flow-facts, using the flag inChanged. Hence, in this situation, we set propagateFurther if inChanged has already been set.
  - Otherwise, if IN and OUT flow-facts are represented using different objects, we perform equality checks between the old and new OUT flow-fact objects using FlowFact::isEqualTo(FlowFact):boolean, and set the flag accordingly.

Note 20.3.6

Unless the IN flow-fact and OUT flow-fact are represented by the same object, we must ensure that a visit() method of a BarrierDirective does not update the internal state of the old OUT object, obtained via a call to NodeInfo::getOUT(), nor should it return back the old OUT object.

Finally, if the flag propagateFurther is set, then we add all the successors of this node to the workList, and return from the method processWhenNotUpdated().

Transfer functions. Following are key points to note concerning the generic definitions of, and suggested guidelines for, various visit() methods.

[7] Note that comparison of OUT might help in early termination of the IDFA. However, that would require extra equality checks which can be more time-consuming.

- The transfer functions are not supposed to be specified at the level of a non-leaf node. Hence, for all those visits that correspond to non-leaf nodes, an assertion is thrown in a *final* implementation of the visits in the base class FlowAnalysis.
- Except for visit() methods for a ParameterDeclaration, and a BarrierDirective, the visit() methods can be overridden by specific analyses as per their needs; the default definition of these methods passes the given IN flow-fact to initProcess() to obtain the OUT flow-fact, which is then returned. The default implementation of initProcess() simply returns its argument.

Note 20.3.7

In the generic pass, except for ParameterDeclaration and BarrierDirective, all transfer functions are identity functions.
In order to specify transfer functions for all types of nodes in a single method, one should override the method initProcess(FlowFact):FlowFact. On the other hand, one can also specify different transfer functions for different types of nodes by overriding their respective visit(Node, FlowFact):FlowFact methods.

- The visit() method for ParameterDeclaration works as follows : First of all, this method obtains the FunctionDefinition to which the given ParameterDeclaration belongs. If FunctionDefinition belongs to main(), then as there are no callers of main() (or so we assume) from within the program, we simply return the obtained (IN) flow-fact as the OUT flow-fact.

Note 20.3.8

The method getEntryFact() should also model the effects of writing of user-defined command-line arguments to the two parameters of main(), if present.

Otherwise, for an intra-procedural control-flow analysis, we obtain the flow-fact to be returned by taking a merge of a TOP flow-fact with the return of an invocation of assignBottomToParameter(), to which we pass this parameter, as well as the argument flow-fact. The method assignBottomToParameter() needs to be implemented by any of the concrete classes of IntraProceduralControlFlowAnalysis; it should model the effect of writing the BOTTOM value to the given parameter.

For all other types of analyses, if there does not exist any caller for the function-definition of this parameter [8] then we simply return back the obtained (IN) flow-fact.

Otherwise, starting with TOP state for OUT flow-fact (that needs to be returned), this method traverses through each argument corresponding to this ParameterDeclaration, from all call-sites for the FunctionDefinition. For each argument, we invoke the method writeToParameter(), passing the provided (IN) flow-fact. This method should be overridden by specific analyses to

[8]This case may be true only when IDFA is being run on a snippet of code, unreachable from main().

model the effect of the write of an argument to formal parameter on the provided (IN) flow-fact. The return of each invocation of writeToParameter() for the argument corresponding to the ParameterDeclaration from a call-site is cumulatively merged into the OUT flow-fact (using merge() method). Finally, the obtained OUT flow-fact is returned.

- In case of control-flow analyses, the visit of a BarrierDirective simply models the identity function. However, in case of data-flow analyses, the visit method is made final in the generic passes; following are some key points concerning the visit() method for a BarrierDirective (in *no-update* mode) [9] :
  - For any given phase, the data-flow values associated with any shared variables (Cells) must be same across all the barriers that end that phase. That is, the OUT flow-fact of barriers ending a phase must be same for those components that correspond to shared variables.
  - Hence, the OUT flow-fact of a barrier is obtained by merging its IN flow-fact with the shared components of the IN flow-fact of other barriers with which this barrier may synchronize in any of the phases.
  - Whenever the IN flow-fact of a barrier changes, we should ensure that we add all its sibling barriers (of every phase), to the workList. This step is not performed within the visit() method, but in processWhenNotUpdated(), relying upon the flag inChanged, by invoking addAllSiblingBarriersToWorkList(). This method traverses through all phases in which the given BarrierDirective may exist (i.e., ones which the given barrier my end), and adds all sibling barriers of the given barrier to the workList [10].
  - Each invocation of the visit() method on a BarrierDirective creates a new OUT flow-fact, initialized to the old OUT flow-fact, if any (or TOP, otherwise). (Note that we should not update the internal state of the old OUT flow-fact, as explained in one of the notes on visit() methods.) Then, for every phase in which the given barrier may exist, we take each sibling barrier and merge the shared components of IN flow-fact of the sibling barrier cumulatively to the new OUT flow-fact for the given barrier. Also, we merge the IN flow-fact of the given barrier to its new OUT (for all components). Finally, we return the newly generated OUT flow-fact.

  As before, the notion of IN and OUT reverses when dealing with backward analyses.

Methods to ensure scope-relevancy in flow-facts. In order to ensure that a flow-fact does not carry forward information corresponding to out-of-scope symbols, we utilize the following four methods :

[9]Note that a different variant of the visit() method (named visitChanged()) is used for a BarrierDirective when the analysis is being run in *update* mode.

[10]Note that this method utilizes SVE-sensitivity for precision, described later in Section 24.

(i) **processBeginNodes(node, newOUT)**. For inter-thread forward cellular flow analysis, if the provided BeginNode corresponds to a FunctionDefinition, then this method collects a set of all those Symbols that are accessible at the entry to FunctionDefinition. This set would be the set of all the global symbols, and the set of formal parameters. After obtaining this set, all those entries from the CellularFlowMap<?> are removed where the key corresponds to a Symbol, its FieldCell, or its AddressCell, such that the symbol is not present in the collected set of symbols.

This way, we ensure that the given (OUT) flow-fact does not contain information about any of the local variables of the caller. (The removal of irrelevant keys is not considered as a change in the state of OUT.)

In case of inter-thread backward flow analysis, this functionality is provided by the method processEndNodes(node, newIN) instead, for the EndNodes.

(ii) **processEndCallNodes(node, newOUT)**. For inter-thread forward cellular flow analysis, given an EndNode, and its new OUT flow-fact, this method performs the following changes in the flow-fact :

a. If the given EndNode corresponds to a CompoundStatement, then for all non-static symbols that are declared in that compound statement, we remove the entries corresponding to that symbol, as well as its AddressCell, and FieldCell, if any, from the flow-fact.
b. Similarly, if the given EndNode corresponds to a FunctionDefinition, we remove entries of a symbol, along with those of its AddressCell and FielCell, if any, from the flow-fact, if the symbol is a formal parameter.

This ensures that the (OUT) flow-fact does not contain information corresponding to the local variables (i.e., formal parameters) of the function. (The removal of irrelevant keys is not considered as a change in the state of OUT.)

Note 20.3.9

Since certain scope-relevancy methods update the OUT flow-fact of a node, we must ensure that the IN flow-fact of such node should not be represented by the same object as the one representing the OUT flow-fact.

In case of inter-thread backward flow analysis, this functionality is provided by the method processBeginNodes(node, newIN) instead, for the BeginNodes.

(iii) **processPostCallNodes(node, newIN)**. This method is applicable only to inter-thread forward cellular flow analysis. This method takes a PostCallNode and its new IN flow-fact as arguments. If the argument is of type CellularFlowMap<?>, it proceeds as follows. To the given flow-fact, this method adds all those entries from the OUT of the PreCallNode (corresponding to the given PostCallNode), whose keys are not already present in the given flow-fact. If any new

entries are added to the given (IN) flow-fact, then the state of IN is conservatively assumed to be changed. (The analysis would still terminate.)

(iv) **processPreCallNodes(node, newIN)**. This method is applicable only to inter-thread backward cellular flow analysis. Its working is exactly similar to that of processPostCallNodes(), except that it works on a PreCallNode, instead of on a PostCallNode.

20.4 Extensible CellMaps

An **ExtensibleCellMap<V extends Immutable>** is a kind of map from set of Cells to set of some immutable values V. The interfaces and behaviour of this map are same as that of a CellMap<V>. Internally, this map differs from a CellMap<V>'s implementation in following key ways :

- The elements in the internal map, named internalRepresentation, may not represent the complete map. They may have extra entries that are logically not assumed to be in the map. They may also have lesser entries than what are logically assumed to be present (which are read from some other map of which this map is an *extension*).
- Every map of this type may contain link, named fallBackMap, to another map of same type. If a key is not present in this map, then the key is searched for in the fallBackMap; the corresponding value, if any, is considered to be the value to which that key is mapped in the fallBackMap.
- This map contains an explicit set of cells, named keysNotPresent, which is used to keep track of keys that are *assumed to be* not present in the set, even if the internal map may have the keys. In other words, if a key is present in the set keysNotPresent, then regardless of whether that key is present in the internal map, a contains() check would always return `false`. Other methods too will behave as though the key is not present in the map.

Other important internal data structures for an ExtensibleCellMap<?> are :

- A flag containsUniversal, inherited from CellMap<?>, indicates whether the *internal map* (and not necessarily the logically represented map) contains an entry for the universal cell.
- A counter freeVariableCount, inherited from CellMap<?>, is used to keep track of the number of keys that are of type FreeVariable in the *internal map*.
- Each map also maintains a set (extensionMaps) of all those maps for which this map serves as a fallBackMap.

Next, we discuss some key details to note concerning various non-trivial methods of this map :

Constructors. An empty constructor initializes the internalRepresentation with a new empty map, and does not set any fallBackMap.

Otherwise, if a map is provided as an argument to the constructor, it sets that map as the fallBackMap if the link length does not exceed a threshold. A *link length* is defined to

be the length of chain of fallBackMaps, starting with the provided map; default threshold is set to 3 (empirically). If the threshold is exceeded, then the constructor copies all logical entries from the provided map into the newly constructed map, by copying the following internal structures : internalRepresentation, keysNotPresent, fallBackMap, freeVariableCount, and containsUniversal.

Note 20.4.1

Note that if we set the threshold for link length as m, the memory consumption may get reduced by a fraction of up to m times.
However, if the threshold m is set too high, then the cost of contains() check can increase prohibitively, as its complexity is $O(m)$ (as compared to $O(1)$ for a HashMap<K, V>).

FreeVariable converter. During any method invocation, if a key under consideration is a FreeVariable, we see if it can be converted back to a Symbol, by invoking the method ExtensibleCellMap::testAndConvert(Cell), or more generally, ExtensibleCellMap::testAndConvert(), which works as follows : The method is first of all called on the fallBackMap, if any. Now, in the current map, if there are no free variables, then the method returns. Otherwise, all FreeVariable cells are collected from the set of non-generic key set (i.e., all explicit keys in the logical map, ignoring the universal key, if any). If a Symbol can be obtained for any of the collected FreeVariables, then we map the value for those free variable to corresponding symbols, and remove the entries corresponding to such free variables; also, the count of free variables is reduced accordingly.

Note that various methods take an extra argument of type ConvertMode, which can either be ON or OFF. When the argument is ON, the method testAndConvert() is invoked at appropriate places; otherwise, no such invocation is performed.

Iterators. The keys of an ExtensibleCellMap<?> may contain the universal cell (which, when present in a set, makes the set behave as a Universal set). For efficiency purposes, there are two variations of iterators –

(i) **keySetExpanded()**, iterates over all the keys explicitly present in the set, followed by all the other keys from the Universal set of keys, if the universal cell is present in the set.
This method starts with an invocation of testAndConvert() to ensure that all FreeVariables that could be replaced by Symbols have been replaced. Then, it simply returns a new object of type ExtensibleCellMap.KeySetExpanded which is a specialized set view of the keys of the logical map. A KeySetExpanded is a RestricedSet, in which, only a handful of set operations are allowed, namely, size(), isEmpty(), contains(), and containsAll(). These operations rely on the corresponding operations on the logical map. A KeySetExpanded also provides an

implementation of the Iterable interface. Its iterator() method returns an iterator of type KeySetExpandedIterator(), which works as follows :

- There are three states in which this iterator exists, specified by STATE, which can take the following values : UNIVERSAL, INTERNAL, and FALLBACK. Corresponding to these three states, there are three iterators on which this iterator relies upon – universalIterator, internalIterator, and fallBackMapIterator, respectively.
  During construction of the iterator, if the internal map representation (not the complete logical map) contains the universal cell, then the initial state of the iterator is set as STATE.UNIVERSAL, and the universalIterator is set to be an iterator on Cell.allCells. Otherwise, the initial state is set as STATE.INTERNAL, and the internalIterator set to be an iterator of the key set of the internal map representation.
- Internally, the next cell to be returned by the iterator is stored in a field nextCell. Each invocation of hasNext() attempts to assign the next available cell, if any, to this field, if the field is set to `null`. If there exists any next cell to be iterated, this method returns `true`, as expected. This method works as follows :
  - If the internal state of this iterator is STATE.UNIVERSAL, it reads the next available cell from universalIterator, if any, such that the cell is not present in keysNotPresent, and sets nextCell to that cell before returning `true`. If no such cell is found, this method returns `false`, indicating the end of iteration.
  - When the iterator's state is STATE.INTERNAL, it attempts to to set nextCell to the next cell available via internalIterator, such that the cell is not present in keysNotPresent. If a cell is found, this method returns `true`. Otherwise, if any fallBackMap exists, the iterator changes its state to STATE.FALLBACK, and sets the iterator fallBackMapIterator to an iterator of fallBackMap (of type KeySetExpandedIterator). Then, this iterator invokes hasNext() on itself in the new state. If no fallBackMap exists, this method returns `false`.
  - If the iterator's state is STATE.FALLBACK, it attempts to get the next available cell from fallBackMapIterator, such that : (i) the cell is not present in the keysNotPresent set, if any, and (ii) the cell is not present in the internalRepresentation's key set (to ensure that same cell is not iterated over more than once). If no such cell is found, this method returns `false`. Otherwise, it sets nextCell to the found cell, and returns `true`.
- Each invocation of next() internally invokes hasNext() to ensure that the field nextCell is properly set. It returns the value of nextCell, (throws a NoSuchElementException if the value is null,) and finally sets the field to null.

(ii) **nonGenericKeySet()**, iterates over only those keys which are explicitly present in the set, ignoring the universal cell, if any.
First of all, this method invokestestAndConvert() to ensure that all FreeVariables that can be replaced by Symbol have been replaced. Then, this method returns a new object of type ExtensibleCellMap.NonGenericKeySet(), which provides a specialized view of the keys present in the logical map. Not unlike a KeySetExpanded, this set is an extension of RestrictedSet, with only the following permitted operations : size(), isEmpty(), contains(), and containsAll(). These operations rely on the corresponding methods of the logical map. Furthermore, a NonGenericKeySet also implements the Iterable<Cell> interface, where the iterator() function returns a new iterator of type NonGenericKeySetIterator() upon each invocation. This iterator works as follows :

- This iterator may exist in one of the two states – (i) STATE.INTERNAL or (ii) STATE.FALLBACK. Corresponding to these two states, the iterators on which this iterator relies upon are internalIterator and fallBackMapIterator, respectively.
  During construction, the default state of the iterator is set to be STATE.INTERNAL, and the iterator internalIterator is initialized to an iterator of the key set of the internal map (internalRepresentation).
- The next item to be returned by the iterator is maintained in a field named nextCell. Each invocation of hasNext() attempts to set this field, if it is `null`, as follows :
  - When the state of the iterator is STATE.INTERNAL, this method searches for the next cell from internalIterator, such that : (i) the cell is not the universal cell, and (ii) the cell is not present in the set keysNotPresent, if any. If any such cell is found, then the method sets nextCell to that cell and returns `true`. Otherwise, if the internal map contains a universal cell, or if the fallBackMap is `null`, then the method returns `false`. (Note that if any key is already present in the internal map, then the fallBackMap would not be looked into.) Otherwise, the state of the iterator changes to STATE.FALLBACK, and the iterator fallBackMapIterator is initialized to a new iterator of the fallBackMap (of type NonGenericKeySetIterator), after which the iterator invokes this method again, in its new state.
  - When the iterator's state is STATE.FALLBACK, the nextCell is set to the next cell obtained from fallBackMapIterator such that : (i) the cell is not a generic cell, (ii) the cell is not present in the set keysNotPresent, if any, and (iii) the cell is not present as a key in the internal map. If found, this method returns true; else false.

- An invocation of next() invokes hasNext() to ensure that the field nextCell has been populated correctly. Then, it simply returns the value of nextCell, if the value is not `null`, and sets the field to `null`. Otherwise, it throws a NoSuchElementException.

Query methods. Now, we note certain observations concerning query methods of an ExtensibleCellMap.

- **isUniversal()** and **hasFreeVariables()**. These methods look only into the internalRepresentation (also referred as internal map), to check whether there exists a universal cell or any FreeVariable, in the keys, respectively.
- **hasDeletedSymbols()**. If there are no deleted cells in Cell.delectedCells, or if there does not exist any cell in Cell.deletedCells which is also obtained from nonGenericKeySet(), then this method returns `false`; else `true`.
- **readsKeyFromTheFallBackMap()**. Given a key, this method returns true only if the fallBackMap, if any, would be referred for searching for the corresponding value of the key (regardless of whether the key is present in the fallBackMap or not). If no fallBackMap exists, this method returns `false`.
- **size()**. If the logical map contains the universal cell as a key, then the size of the map is returned as the size of Cell.allCells, minus the size of the set keysNotPresent.
  If the fallBackMap and keysNotPresent sets are null/empty, then the size of the internal map is returned. If the fallBackMap is empty by keysNotPresent is not, then the size of the map is the number of elements that are present in the internal map but not in the set keysNotPresent.
  If the fallBackMap exists, we simply iterate over all the elements in the set returned by keySetExpanded(), increase a counter by one for each element, and return the counter.
- **isEmpty()**. If the internal map is empty, and there is no fallBackMap, then this method returns `true`. If the fallBackMap exists, and if internal map and keysNotPresent, if any, are empty, then the method recursively checks if the fallBackMap is empty, and returns the result. Otherwise, this method invokes size() on the map and returns `true` if the size is zero; else `false`.
- **containsKey()**. First of all, this method invokes testAndConvert() on the map to replace FreeVariables with Symbols, wherever possible. Then, if the queried key is a FreeVariable, this method attempts to convert it into a Symbol, using testAndConvert().
  If the key is present in the set keysNotPresent, if any, then this method returns `false`. Otherwise, if this key is a universal cell, then this method returns `true` iff the logical map contains the universal cell. Otherwise, if the internal map contains the given key, or if it contains the universal cell, then the method returns `true`. Otherwise, if the fallBackMap

does not exist, then the method returns false; if it exists, then the query is performed recursively on the fallBackMap, and the result is returned.

- **containsValue()**. This method iterates over the set returned by keySetExpanded(), and returns `true` only if there exists any key whose corresponding value matches the queried value.
- **get()**. This method invokes testAndConvert() on the map to replace FreeVariables with Symbols, wherever possible. Then, if the given key is a FreeVariable, this method invokes testAndConvert() for the key to check if a corresponding Symbol can be obtained.
  If the key is a universal cell, and the set keysNotPresent is not empty, then `null` is returned back. Otherwise, if the key is a universal cell, and the internal map contains the universal cell, then the corresponding value is returned only when there does not exist any other key in the internal map; if there are any other entries, then `null` is returned.
  If the key is not the universal cell, and the set keysNotPresent contains the key, then `null` is returned. Otherwise, if the key is not the universal cell, and the internal map contains either a value for the given key, or if not, then the value for the universal cell, then that value is returned.
  Otherwise, if the fallBackMap does not exist, then `null` is returned; else the query is made recursively on the fallBackMap.
- **clone()**. A clone of this map is obtained by simply calling the copy constructor, passing this map as an argument. Note that the internal map is not copied in the copy constructor (unless the *link length* exceeds certain threshold); the receiver map will be set as the fallBackMap of the returned map. The API ensures that no changes made on the returned map will be reflected in the receiver map.
- **hashCode()**. For simplicity, we assume the size of the logical map to be its hash code. A more precise, albeit complicated, solution of obtaining the hash code as some function of the hash code of all the keys in the logical map proved to be quite inefficient.

Note 20.4.2

The hashCode() must rely only on aspects of the *logical map* represented by an ExtensibleCellMap, and not on any of its individual components. Two maps with different sets of keysNotPresent and internalRepresentation may still be logically same (and hence, their hash codes must be same).

- **equals()**. If the given object is not of same class as the receiver map, then this method returns `false`. If the receiver and the given maps do not contain keysNotPresent, have equal internal maps, and point to the same fallBackMap, then, as a special case, this method returns `true`.

This method iterates over the elements of the set returned by keySetExpanded(), i.e., on the logical key set of the receiver map, and returns `false` if (i) no entry exists for any key of the receiver map in the given map, or (ii) the values mapped to any key are different in the receiver map and the given map.

Otherwise, if the number of entries in the logical view of the receiver map and the given map are not same, then this method returns `false`; else true.

Update methods. Following are some key points to note concerning internal workings of those methods that may update the logical map. Note that care must be taken in ensuring that changes made to an ExtensibleCellMap do not get reflected in any of the other maps of which this map is a fallBackMap.

- **put()**. If the given key is a FreeVariable, this method attempts to convert it into a Symbol, using testAndConvert().

  First of all, the old value is fetched from the map, and compared with the new value. If the values are equal, this method returns the old value.

  Depending upon whether the key for which an entry has to be added/updates is a universal cell, two cases arise :

  – *Case 1. Key is not the universal cell.*

    First of all, we remove the key from keysNotPresent, by invoking the method removeKeyFromKeysNotPresent(). Note that this step should not affect those maps which extend this map. The method removeKeyFromKeysNotPresent works as follows : If the given key is not present in the set keysNotPresent, this method simply returns. Otherwise, this method iterates over all maps from extensionMaps of the receiver map, and adds this key to the keysNotPresent of a map, if the extension map would refer its fallBackMap for the key (tested using readsKeyFromTheFallBackMap). Finally, the key is removed from the keysNotPresent of the receiver map.

    Next, we need to shift the old mapping of the key, if any, to the extension maps. The old entry is put into an extension map of the receiver map, if the extension map would refer to its fallBackMap for the key (tested using readsKeyFromTheFallBackMap). After shifting the old entries to the extension maps, the new key-value pair is added to the internal map of the receiver map. If the key is FreeVariable, then we increment the counter freeVariableCount by one.

  – *Case 2. Key is the universal cell.*

    In this case, first of all we need to remove all the keys from keysNotPresent. We do so by invoking removeKeyFromKeysNotPresent(), which will shift these keys to the keysNotPresent of the extension maps, if required.

Then, we shift all the mappings from internal map of this map to the extension maps, if applicable. A mapping from the internal map of the receiver map is added to the internal map of an extension map, if the extension map would refer to the fallBackMap for the key corresponding to the mapping (tested using readsKeyFromTheFallBackMap).
While adding a mapping to the extension map, if the key is the universal cell, then we set the flag containsUniversal in the extension map. Finally, for each extension map, the fallBackMap of the receiver, if any, is set as the fallBackMap of the extension map.
The internal map of the receiver map is then cleared, and the requested key-value pair is added to it. The containsUniversal flag is set, and freeVariableCount is reset to zero.

- **remove()**. If the key to be removed is a FreeVariable, this method attempts to convert it into a Symbol, if possible, using testAndConvert().
  If the map does not contain the given key, this method returns `null`. Otherwise, the value corresponding to the key is captured, and is used later as the return value of this method.
  If the key is not the universal cell, this method simply adds the key to the set keysNotPresent of the receiver map using addKeysToKeysNotPresent(), which works as follows :
  - If the key is not present in the logical map, this method returns. Otherwise, the value corresponding to the key is captured.
  - The mapping for the key and its associated value is added to the internal map of an extension map of the receiver map, if the extension map would refer its fallBackMap for the key (tested using readsKeyFromTheFallBackMap). Finally, the key is added to the keysNotPresent set of the receiver map.

  Otherwise, if the key is the universal cell, following steps are taken :
  - If the set keysNotPresent is not empty, we empty it after shifting all such keys to the extension maps, if required, using removeKeyFromKeysNotPresent(). These keys are also removed from the internal map of the receiver, if present.
  - If an extension map does not contain the universal cell, we take each non-universal cell from the receiver map, and add its corresponding entry in the internal map of the extension map if it refers to its fallBackMap for the cell (tested using readsKeyFromTheFallBackMap).
    Finally, the mapping for universal cell from the receiver map is moved to the internal map of each extension map, setting their containsUniversal flag.
  - Then, all the data structures corresponding to the receiver map are reset to their default values.

- **clear()**. In order to clear a set, we simply iterate over all the elements of the logical view of the receiver map (obtained using keySetExpanded()), and remove them from the receiver map.
- **mergeWith()**. This method takes the following three arguments :
  (i) thatMap, a map which has to be merged into the receiver map.
  (ii) mergeMethod, a method that is used to obtain the merged value from the values for any given key in the two maps.
  (iii) selectedCells, a set of cells for which the merge operation has to be performed; when `null`, we interpret this set to represent the universal set of cells.

  Given these arguments, this method takes each element that is present in the selectedCells and in the key set of thatMap, and merges it into the receiver map. If the receiver map is changed as a result of this merge operation, then this method returns true.

  If thatMap is `null`, this method simply returns `false`. Otherwise, for each key which is present in the selectedCell (assuming that a `null` set contains all the keys), this method performs the following steps :
  - For each non-universal key of thatMap (obtained using nonGenericKeySet()), this method applies the mergeMethod on the value corresponding to that key in the thatMap and value from the receiver map, if any. If the value is different than what was present in the receiver map, this method replaces the old value with new, and sets the changed flag (which was initialized to `false`).

    Then, if there is no universal cell in the thatMap, this method returns the value of the changed flag. Otherwise, this method makes the following changes :
    * For every non-universal key in the receiver map, which is not present in the non-generic key set of the thatMap, the value for that key in the receiver map is replaced with the value obtained after applying mergeMethod on that value and the value corresponding to the universal cell in thatMap. If the new and old values are different, the flag changed is set.
    * For every key in the selectedCells (or in the universal set of cells, if the selectedCells is `null`), which is neither present in the non-generic key set of the receiver map nor in that of the thatMap, an entry is added to the receiver map, with value obtained using mergeMethod when applied on the value corresponding to the universal cell in the thatMap, and the one in the receiver map, if any. If the receiver map does not contain the universal cell, then instead of invoking the mergeMethod, the value corresponding to the universal cell in the thatMap is used as it is for the new entries.

20.5 Postorder and reverse postorder collectors

For efficient termination of the forward IDFA passes, reverse postorder from the control-flow graph is the recommended order in which the nodes should be processed. IMOP provides a class **ReversePostOrderWorkList**, which is used to obtain a list in which the nodes are maintained in the order that respects the reverse postorder sequence of the nodes from the control-flow graph. There are certain other optimizations performed within this list to ensure efficient processing of the IDFA pass.

A ReversePostOrderWorkList also maintains the notion of phase ordering, where the nodes of a phase are never prioritized for processing unless the barriers that start that phase have been processed. Similarly, the ending set of barriers are never prioritized over the nodes of the phase to be processed. In order to implement such ordering, a ReversePostOrderWorkList comprises of two internal lists : (i) nonBarrierList, and (ii) barrierList. The list can exist in one of the two states, namely, Stage.NONBARRIER, and Stage.BARRIER. The initial state of any list is Stage.NONBARRIER. Following are some key observations concerning this class :

- **add()**. Given a node to be added in the list, first step is to check whether the node is a barrier (or EndNode of a parallel construct), or something else. Accordingly, one of the two lists is selected to add the node in. If the selected list is empty, the node is simply added, and the method returns. Otherwise, if the selected list already contains the node, the method returns. Otherwise, the sequence id (index) of the node in the reverse postorder is fetched by invoking NodeInfo::getReversePostOrder() on the node. (The method getReversePostOrder() is explained later in this section.) If the index could not be found, the node is added at the last of the list, and the method returns. If the index was found, this method uses insertUsingBinarySearch() to insert the node at its appropriate location within the selected list.
- **insertUsingBinarySearch().** This method performs insertion of the given node, by searching for its appropriate place using binary search on the reverse postorder id of the node. For the purpose of this discussion, let us assume that the reverse postorder sequence id of a node is referred to by the term *weight* of the node.

  Given a sub-list with its start and end index, in which the given node with given weight has to be inserted, this method searches for the proper index for the node and inserts it there, such that the complete list remains sorted with respect to the weights of the nodes. This method works as follows :
  - If the start index of the list is similar to, or greater than the end index, then we look into the weight of the node at the start index. If the weights are same and the nodes are same, then this method returns. However, if the nodes are different, despite having same weight, we add the given node at the start index, and then return. If the weight of the node at the start

Note 20.5.1

Currently, maybe due to a bug, or due to the way things should be, there are various nodes which might be sharing the same reverse postorder id, after program transformations are performed. For now, we ignore this observation. However, this points to some missing update of these id's during the elementary transformations. ***UPDATE:*** **"For now, we have implemented a workaround for this issue by setting the weights of all the nodes that are present in the reverse postorder sequence to -1, before calling Program.stabilizeReversePostOrderOfLeaves – a method that should have anyway reset the weights of every node that is reachable from `main()`. "**

index is smaller than that of the given node, then we insert the given node immediately after the node at the start index.

- Some nodes may not have a valid weight due to lack of required update during elementary transformation. (Their weight would be -1.) Such nodes are present at the end of the list. Hence, if the weight of the last node is -1, we invoke this function recursively on the sub-list that does not contain that node.
- If there are no nodes with invalid weight at the end of the list, we look into the weight of the node at the center of the list. If the weight of the center node is same as that of the given node, and if the nodes are same, then this method returns; however, if the weights are same but the nodes are not, then the node is added immediately before the center node, and the method returns.

  If the weight of the given node is smaller than the weight of the center node, then this method is invoked recursively on the sub-list that starts at the start index and ends immediately before the center node. Otherwise, the method is invoked recursively on sub-list that starts immediately after the center node, and ends at the end index.

- **removeAnyElement().** This method is used to remove and return an element from the appropriate list. If the state of the list is NONBARRIER, we select the nonBarrierList as the list from which the element should be taken. However, if the nonBarrierList is empty, we change the state to BARRIER, and select the list barrierList. Similarly, if the state of the list is BARRIER, we select the barrierList as the list from which the element should be returned. However, if the barrierList is empty, the state is changed to NONBARRIER, and select the list nonBarrierList.

  If the selected list is empty, we return `null` from the method. Otherwise, we remove the first element from the selected list (index zero), and return it.

Getting the reverse postorder.

The method **NodeInfo::getReversePostOrder()** is used to obtain the value of the field NodeInfo::reversePostOrderId for the receiver leaf CFG node, indicating its position in the reverse

postorder traversal of the program's super control-flow graph (i.e., a CFG obtained after connecting all the CFGs of various procedures with the help of call and return edges from the call graph) This ordering is used by IDFA analyses to decide the order in which nodes present in the work list are processed. A reverse postorder would respect the following properties :

- Assuming that each cycle in the super CFG is considered as a single node, none of the nodes in the resulting DAG is processed until all its parents have been processed.
- Similarly, within a cycle, assuming that the back-edge does not exist, none of the nodes is processed until all its parents have been processed.

These properties ensure that the fixed-point for an IDFA is reached early, without having to reprocess a node more number of times than required.

Before returning the value of the aforementioned field, this method first invokes Program.stabilizeReversePostOrderOfLeaves() to ensure that such fields have been populated for all reachable leaf CFG nodes.

The method **Program.stabilizeReversePostOrderOfLeaves()** works as follows

- The main aim of this method is to obtain a reverse postorder sequence of leaf nodes that are reachable from the entry point of `main()`, and save it in the global list of nodes, named Program.reversePostOrderOfLeaves. This method also saves the sequence id of each node in the field NodeInfo::reversePostOrderId for that node.
- There exists a global flag Program.postOrderValid, which is set to `true` at successful completion of this method. This flag is initially `false`, and it also gets reset again during automated update of the program, under any elementary transformation. If this flag is found to be set, then this method simply returns.
- If the flag Program.postOrderValid is not set, this method begins the processing by traversing over all the pre-existing nodes in Program.reversePostOrderOfLeaves, and resetting their field NodeInfo::reversePostOrderId to -1.
- If no `main()` exists then this method returns.
- In order to obtain the reverse postorder list of nodes, this method invokes TraversalOrderObtainer.obtainReversePostOrder() on the BeginNode of `main()`, and a lambda which returns a list of inter-task successor leaf nodes for any given node (using CFGInfo::getInterTaskLeafSuccessorList()). The returned list from obtainReversePostOrder() is saved in Program.reversePostOrderOfLeaves.
- Then, this method iterates over all nodes in the list Program.reversePostOrderOfLeaves and sets the field NodeInfo::reversePostOrderId of each node with its index in that list.
- Finally, this method sets the flag Program.postOrderValid to `true`, and returns.

The method **TraversalOrderObtainer.obtainReversePostOrder()** is used to obtain the reverse postorder of all the reachable nodes from the given node (of any generic type), and the given lambda which gives the neighbors (in any generic graph) of each node. It achieves so by first obtaining the postorder sequence, and then by returning the reverse of it. The postorder sequence is obtained by an invocation to TraversalOrderObtainer.obtainPostOrder(). This method, in turn, invokes a recursive method TraversalOrderObtainer.performPostOrder(), which works as follows :

- This recursive method takes a generic node, a lambda that provides a list of neighbors of the node in the generic graph, a set to keep track of the nodes that are under traversal (graySet), a set to keep track of nodes that have already been traversed (blackSet), and a list which this method would populate with the traversed nodes (printList), in postorder.
- If the given node is present in the graySet, then this method returns, lest it would enter a cycle. Otherwise, it adds the node into the graySet, indicating that the node is being traversed.
- For each neighbor of the node, obtained by applying the given lambda, this method recursively calls itself on the node, unless the node is already present in the blackSet.
- Upon returning back from the invocation on all neighbors of the node, this method shifts the node from the graySet to the blackSet
- Finally, it adds the node at the end of the printList. This way, this method ensures that a node is added to the list only after all its neighbors have been added, thereby attaining the postorder.

21 GENERAL GUIDELINES TO IMPLEMENT AN IDFA

In this section, we detail out various steps that are required while instantiating any of the generic flow passes to create a concrete flow analysis. We also reiterate various points from Section 20 to be kept in mind while instantiating the generic pass.

In order to create a new flow analysis, first of all, we need to decide which generic pass should we extend. If the flow fact of the analysis will not depend on the contents of the nodes, then we need a control-flow analysis. All such analyses will by default be intra-thread in nature. If we need an intra-procedural analysis, we should extend IntraProceduralControlAnalysis<F>; for inter-procedural analyses, we should extend InterProceduralControlAnalysis<F>.

If the flow fact of the analysis can depend on the contents of the nodes, then we require a data-flow analysis. These analyses are inter-thread in nature, by default. If the flow fact can be modelled as a map from a set of Cells to a set of Immutable values, then we need to use a cellular analysis. For forward cellular analysis, we should extend InterThreadForwardCellularAnalysis; for backward, InterThreadBackwardCellularAnalysis. On the other hand, if the structure of flow fact does not fall in this category, then we can extend InterThreadForwardNonCellularAnalysis for forward analyses, and InterThreadBackwardNonCellularAnalysis for backward analyses.

Note 21.0.1

Following are some important points to keep in mind while instantiating the any generic flow pass :

- The methods getTop() and getEntryFact() should never return `null`, and they should always return a new object.
- The method getEntryFact() should also model the effects of the parameters of main() if any, on the returned flow fact/map.
- None of the visit() methods, initProcess() method, merge() method, or writeToParameter() method, should change the internal state of their arguments. They are allowed to return any/either of their arguments, wherever applicable, except for the nodes listed in next point.
- If the flow maps are of type CellularFlowMap<?>, then the IN flow map must never be returned as OUT flow map by the transfer functions of the following nodes :
  - BeginNode of a FunctionDefinition.
  - EndNode of a FunctionDefinition, or of a CompoundStatement.
  - PostCallNode of any CallStatement.

Note that currently all the data-flow analyses are flow-sensitive, path-insensitive, and field-sensitive in nature. Versions with other values for these analysis dimensions can be added later on demand.

21.1 Cellular data-flow analyses.

Following are the steps to create an instance of the inter-thread flow-sensitive inter-procedural context-insensitive generic IDFA pass, where the IN and OUT flow maps at any given node are in the form of a map from set of Cells to set of some immutable values :

Step 1. First of all, we need to create a unique constant in the enumerator AnalysisName, corresponding to our new analysis. For example, the points-to analysis has a unique constant AnalysisName.POINTSTO.

Step 2. Next, we create the main class for our analysis by extending the generic pass InterThread*CellularAnalysis<F>, where F is another specific class extended from CellularFlowMap<V extends Immutable> denoting the type of the flow maps of this analysis, with V being the type of the co-domain of the flow maps of this analysis. We usually make F a static inner class of the analysis class.

Step 3. For each flow map that extends CellularFlowMap<V>, following are the three methods that need to be implemented :

(i) **Constructors**. Two constructors are needed to be provided. One that takes an ExtensibleCellMap<V>, and another that takes a CellularFlowMap<V>. Both these constructors can simply pass the arguments to their superclass.

(ii) **getAnalysisNameKey() : String**. This method is used to return a unique string for this analysis which may be helpful in identifying it in any debugging data. For example, the string for points-to analysis is *"ptsTo"*.

(iii) **meet(V, V) : V**. This method models the meet operation of the lattice corresponding to this analysis. It takes two arguments of type V, and returns back an object which represents the meet of both the arguments as per the lattice. Note that V is an immutable type. Hence, this method cannot (should not) change the internal states of its arguments. However, it is allowed to return one of its arguments, if needed.

Note that V must implement Immutable. This interface does not contain any members to be defined. It only serves as a reminder to the user that the internal state of V must be immutable (as same object may be reused by different flow maps to ensure efficiency). Some examples for V are : ImmutableCellSet, ImmutableDefinitionSet, etc. The related flow maps for these example V's are PointsToFlowMap, and ReachingDefinitionFlowMap, which extend CellularFlowMap<ImmutableCellSet>, and CellularFlowMap<ImmutableDefinitionSet>, respectively.

Step 4. Following are the four methods that must be overridden by each analysis that extends InterThread*CellularAnalysis<F> :

(i) **Constructor**. Each analysis needs to have a constructor that passes two arguments to the constructor of its superclass InterThread*Analysis<> – (i) the constant corresponding to that analysis in AnalysisName, and (ii) an object of type AnalysisDimension, which is used to specify various analysis dimensions for this analysis. For now, the only constructor of AnalysisDimension that we use is one which specifies whether the graph traversal used in the analysis is SVE-sensitive or not (by passing either SVEDimension.SVE_SENSITIVE or SVEDimension.SVE_INSENSITIVE). Refer to Section 24 on more details on SVE (single-valued expressions).

(ii) **getTop() : F**. This method should be overridden to return back a *new* object denoting the TOP element of the lattice corresponding to this analysis, upon each invocation.

(iii) **getEntryFact() : F**. This method should provide a new object upon each invocation, denoting the entry IN flow map for the BeginNode of main(). It needs to consider all the other globals present in the program, and also, both the parameters of main(), if present.

(iv) **writeToParameter(ParameterDeclaration, SimplePrimaryExpression, F) : F**. This method should provide the transfer function that models the effect of assignment of an argument (the SimplePrimaryExpression) to the parameter (ParameterDeclaration), on the given IN flow map (the third argument), and return the resulting OUT flow map. Note that

this method must not change the internal state of its third argument. However, it is allowed to return it as it is (when the transfer function is an identity function, for an instance).

Some examples of concrete IDFA analyses are PointsToAnalysis, ReachingDefinitionAnalysis, etc.

Step 5. The setup so far should give us a working IDFA pass, which assumes that all the transfer functions corresponding to each kind of CFG leaf nodes are identity functions. In order to specify the actual transfer functions, which provide the OUT flow map given an IN flow map at a given node, there are two alternatives :

(i) The **initProcess(Node, F) : F** method, can be used to provide a generic definition of the transfer function for each type of leaf CFG node. Given a node and the IN flow map, this method should return back the corresponding OUT flow map. It must not change the internal state of its argument. However, it can return the argument as it is, if needed, except for the following nodes : BeginNode of a FunctionDefinition; EndNode of a FunctionDefinition or CompoundStatemnet; and any of the PostCallNodes. If the type of the flow map is not CellularFlowMap<?>, then the argument can be returned, regardless of the type of the node. This option is well suited for those cases where the code needed to be written for implementing the transfer function does not differ much across the type of leaf CFG nodes. For an example, refer to the class ReachingDefinitionAnalysis.

(ii) Alternatively, one can use the **visit(Node, F) : F** methods, which take a specific type of leaf CFG node, and an IN flow map, and return the OUT flow map corresponding to that type of node. For each type of leaf CFG node, a different visit() method needs to be overridden. The transfer function for any type for which no visit() method has been overridden is assumed to be an identity function. As before, these methods must not change the internal state of their flow map argument. However, they can return the argument as it is, if required, except when they are visits of the following nodes, and the flow maps are of type CellularFlowMap<?> : BeginNode of a FunctionDefinition; EndNode of a FunctionDefinition or CompoundStatement; or any PostCallNode.

This approach is well-suited in those cases where different codes need to be provided for implementing the transfer functions for different kinds of leaf nodes. For example, refer to the class PoinstToAnalysis.

One must be careful if attempting to use both the options in any analysis. It is not recommended, and might not reflect a good design of the analysis.

Note that if one needs to implement any edge-transfer functions (useful for modelling effects of taking specific branches, on flow facts), then one should also override the edgeTransferFunction().

In order to ensure that the newly created analysis can automatically run upon first invocation of NodeInfo::getIN(AnalysisName) or NodeInfo::getOUT(AnalysisName), on any given node, we need to ensure the following :

(i) Add a unique boolean flag corresponding to the new analysis as a static member of NodeInfo, initialized to `false`.
(ii) In method NodeInfo::checkFirstRun(), which is automatically invoked at the beginning of getIN() and getOUT, add code corresponding to the new analysis (similar to the pre-existing codes for other analyses), which runs the analysis on main() only if the corresponding flag is not set, and then sets the flag.

21.2 Non-cellular data-flow analyses.

Note that if the type of flow fact for a data-flow analysis is not a map from set of cells to set of some immutable values, then one needs to directly extend from the FlowFact class to obtain the flow map type specific to the given analysis, which should extend from any of the InterThread*NonCellularAnalysis classes.

Any such subclass of FlowFact needs to override the following methods :

- The **isEqualTo(F):boolean** method, which takes a flow fact and compares it to the receiver flow fact. This method is used to ensure termination.
- The **merge(F, CellSet):boolean** method, which takes a flow fact, and updates the receiver flow fact such that it starts reflecting the merge of its initial state and the given argument. The merge operation between the flow facts might generally rely upon applying the meet operation on various elements of the flow facts (depending upon whatever be the internal structure of the flow fact). This method must not change the internal state of its argument.
- The **getString():String** method should return the string representation of the flow fact, which is used for debugging purposes.

Furthermore, a subclass of FlowFact would also have to explicitly declare all data members that can logically represent the flow fact; a constructor that populates these data members should also be provided. For an example of such a flow fact, refer to the class HeapValidityAnalysis, which uses ValidityFlowFact that does not extend from CellularFlowMap<?>.

21.3 Control-flow analyses.

Apart from getTop(), getEntryFact(), and one or more of the visit(), initProcess(), and edgeTransferFunction() methods, as explained above for data-flow analyses, in case of instantiations of intra-procedural control-flow analyses, we need to implement the following method :

- **assignBottomToParameter(ParameterDeclaration, F):F** which models the effect of writing the BOTTOM value (of lattice) to the parameter, on the given flow fact, and return the resulting flow-fact. Note that for the case of intra-procedural analyses, the method writeToParameter() is not applicable.

For inter-procedural control-flow analyses, the nature of methods that need to be overridden remains same as in the case of data-flow analyses.

22 INSTANTIATIONS OF GENERIC FLOW PASSES

In this section, we discuss the details of some concrete IFA passes that are already present in IMOP.

22.1 Points-to analysis

IMOP provides an inter-thread flow-sensitive inter-procedural context-insensitive sve-sensitive points-to analysis, via the class **PointsToAnalysis**. The corresponding enum constant for this analysis in AnalysisName is POINTSTO; this constant is used to access the IN and OUT flow maps corresponding to this analysis, for any given node.

The flow maps related to this analysis are of type PointsToFlowMap, which is a subclass of CellularFlowMap<ImmutableCellSet> (Section 20.1). An **ImmutableCellSet** is a special CellSet, which does not allow any update operations on itself after its construction. We use such immutable sets to ensure that (i) we can reuse the sets in multiple flow maps, (ii) we can use CellularFlowMap<?> as the type for flow maps of this analysis, as the values of CellularFlowMap<?> must be immutable.

At any given node, each entry in the internal map of a flow map at that node maps a cell to a set of cells that it may point to, at that node. Some key points to note concerning PointsToFlowMap :

- There are two constructors – one takes a cell map (ExtensibleCellMap<ImmutableCellSet>, from here on, synonymous to ExtensibleCellMap, within this section) as an argument, and uses the same object as the internal map within the newly constructed flow map, whereas, the another constructor takes a flow map (CellularFlowMap<ImmutableCellSet>, from here on, synonymous to CellularFlowMap within this section)as an argument and sets the internal map to a new ExtensibleCellMap< object, obtained from the internal map of the given argument.
- The debug string corresponding to this flow map is *"ptsTo"*.
- The **meet()** operation, which takes two ImmutableCellSet sets and returns another set which is the meet of the given sets, works as follows :

  If both the given sets are `null`, then `null` is returned. Otherwise, if one of the sets is `null`, then the another set is returned. If both the given sets are equal, then any of the sets is returned. Otherwise, the union of both the sets (a new object) is returned.

Now, we look into the details of the PoinstToAnalysis :

- **The TOP flow map.** The getTop() method simply returns a new PoinstToFlowMap upon each invocation, which contains an empty ExtensibleCellMap.

Note 22.1.1

For any given cell, we assume that an empty points-to set can also be denoted by a `null` value.

- **The entry flow map.** The entry flow map for an analysis is obtained by an invocation to the method getEntryFact(). In this method, we look into each global Declaration present in the TranslationUnit, such that : (i) the Declaration is not that of a `typedef`, (ii) it does not declare more than one entity, and (iii) the declared entity is of type PointerType.

  Corresponding to the symbol declared in each such Declaration, we add an entry from that symbol to an ImmutableCellSet, in the entry flow map (which is initialized to the TOP flow map). The set is obtained as follows :
  - If there is no initializer in the declaration, ImmutableCellSet contains only the NULL cell (obtained via Cell.getNullCell()).
  - Otherwise, we obtain the locations (cells) represented by the initializer, using the method CellAccessGetter.getLocationsOf() (Section 10). If the locations contain the universal cell, we populate the ImmutableCellSet only with the universal cell. Otherwise, corresponding to each location, we add (i) the element pointed to by the AddressCell, if the location is an AddressCell, (ii) the location itself, if it is a FieldCell, or (iii) the values obtained from the entry map itself, corresponding to each location.

  Furthermore, to the entry flow map, we also add an entry corresponding to the pointer parameter of main(), if any, with the value obtained via an invocation of HeapCell.getUnknownParamPointee().
- **The transfer functions.** The transfer functions for the following leaf CFG nodes is an identity function : UnknownCpp, UnknownPragma, OmpForCondition, OmpForReinitExpression, FlushDirective, DummyFlushDirective, TaskwaitDirective, TaskyieldDirective, GotoStatement, ContinueStatement, BreakStatement, BeginNode, EndNode, and PreCallNode.
  ***UPDATE:* "In case if a BeginNode belongs to a FunctionDefinition, or an EndNode belongs to a FunctionDefinition or to a CompoundStatement, then we return a copy of the IN flow map for such nodes, to ensure that the scope-relevance code does not break. "**

  Note that the transfer function corresponding to the BarrierDirective is provided by the generic pass, and cannot be overridden. Next, we explain the transfer functions for other kinds of CFG leaf nodes :
  - The transfer function of a Declaration works as follows :

* If the declaration is not that of a PointerType, then the transfer function is an identity function.
* If there is no initializer in the declaration, then we assume that the points-to set on the right-hand side of the assignment in the declaration contains only one element, the NULL cell. Otherwise, if the locations represented by the initializer contains the universal cell, then we assume that the points-to set of the RHS contains only the universal cell. Otherwise, the points-to set of the RHS will contain the points-to cells of all the locations represented by the initializer (obtained using Cell.getPointsTo(Node):ImmutableCellSet).
* If the points-to set of the RHS is not empty, then we invoke the method Optimized-PointsToUpdateGetter.generateUpdateMap() with appropriate LHS and RHS sets; the workings of this method are explained below :
  - This method is used to obtain the map of new points-to flow-facts given a logical assignment.
  - This method takes three arguments : (i) a node for which this method has been invoked, (ii) a set of cell, lhsSet, denoting the LHS of the assignment being modelled, and (iii) a set of cell, rhsPtsToSet, denoting the points-to set of the RHS of the assignment being modelled.
  - If any of the sets are empty or null, this method returns an empty update map.
  - In order to indicate whether the new may points-to information would kill the existing information, we maintain a flag strongUpdate, which is initially set to `true`.
    We reset the flag to `false`, if any of the following conditions are true :
    (i) if there are more than one elements in the lhsSet,
    (ii) otherwise, if the element in the lhsSet is the universal cell,
    (iii) if the element is a HeapCell,
    (iv) otherwise, if the element is an aggregate type (ArrayType or StructType), or a UnionType.
  - If the flag strongUpdate is set, we simply add a mapping from the sole element in lhsSet to the complete rhsPtsToSet, in the update map, and return the map.
  - Otherwise, when the flag is `false` and the lhsSet is a universal set, then we add a mapping from the universal cell to the universal set in the update map, and return it.
  - Otherwise, if the flag is false and the lhsSet is not a universal set, we add an entry from each element of the lhsSet to the union of the old points-to set of that element and the rhsPtsToSet (if the union is not equal to the old points-to set of the element).

The arguments passed to this method from the visit of a Declaration are : the declaration node; a set containing only the declared symbol; and a set containing union of points-to of all the locations represented by the initializer (as described in the previous point).

* If the result of the invocation of generateUpdateMap() is not empty, we change the OUT flow map (which is initialized with a copy of the given IN flow map) by invoking the mergeWith method on the flow map with following arguments : (i) the update map; (ii) a lambda that specifies a meet method that returns the value from the update map, if any, otherwise the value from the receiver map; and (iii) a null set, representing the universal set of cells.

  Finally, the updated OUT flow map is returned by the transfer function.

– For the case of an OmpForInitExpression, of the form `id = expression`, the transfer function is specified in a manner similar to how it is specified for a Declaration, replacing the declared symbol with `id`, and the initializer with the `expression`.

  To obtain the transfer function of the following type of leaf nodes, we invoke the method PointsToAnalysis.updateOptimizedPointsTo(), with the specified argument, to obtain the OUT flow map, given the IN flow map :

  * ExpressionStatement, with its expression as the argument.
  * ReturnStatement, with the returned expression as the argument.
  * Expression, with itself as the argument.
  * IfClause, NumThreadsClause, and FinalClause, with their respective expression as the argument.

  The method updateOptimizedPointsTo() works as follows :

  * This method takes an IN flow map, and an expression node, and returns the OUT flow map which would be obtained as a result of the symbolic execution of the given expression.
  * First of all, the visitor OptimizedPointsToUpdateGetter is invoked on the expression, to obtain the updated points-to flow map. If the update map is empty, this method simply returns the IN flow map as the OUT flow map. Otherwise, the OUT flow map is changed by invoking the method mergeWith, in a manner similar to how it is done in the transfer function of a Declaration.
  * The visitor OptimizedPointsToUpdateGetter works as follows :
    · This visitor contains an update map, named updateMap, which is initially empty; at the end of the call to this visitor, this update map is the map which is useful for the caller.
    · When visiting a NonConditionalExpression (which represents an assignment statement), this visitor first obtains the set of locations representing the LHS and RHS

of the assignment. If either of the sets are empty, the visit does not perform any actions. If there exists any location in LHS which is not a PointerType, then no action is performed. If the RHS set is a universal set of cells, then the points-to set of the RHS set contains only one value – the universal cell. Otherwise, it contains the union of points-to of all the elements of the RHS.
Now, given the set of locations on the LHS of the assignment, and the union of the points-to set of the locations on the RHS of the assignment, we update the field updateMap in a manner similar to how we do it for the OUT flow map in the transfer function of a Declaration.

- Note that the expressions within the `sizeof` operator are not evaluated. Hence, this visitor does not traverse within the UnarySizeofExpression and SizeofUnaryExpression nodes.

– The transfer function that models the write of an argument to the parameter of a called function is modelled using the method **writeToParameter()**, which is overridden as follows : If the argument is a Constant, this method returns the IN flow map as the OUT flow map. Otherwise, the processing is similar to how it is done for a Declaration, with the declared symbol replaces by the symbol denoting the parameter, and the initializer replaced by the argument.
– Finally, for a PostCallNode, the transfer function is defined as follows : If the associated function does not return any value, ***UPDATE:* "a copy of the"** IN flow map would be returned as the OUT flow map. Otherwise, the transfer function behaves similar to how it does for a Declaration, where the declared symbol is replaced with the capturing identifier, and where instead of taking the points-to set of the locations represented by the initializer of the Declaration, we take union of points-to set of the locations represented by each possible return value, of each possibly called function.

22.2 Reaching-definitions analysis

IMOP provides an inter-thread flow-sensitive inter-procedural context-insensitive sve-sensitive reaching definitions analysis, via a class **ReachingDefinitionAnalysis**. The unique constant in AnalysisName corresponding to this analysis is REACHING_DEFINITION.

The flow maps of a reaching definition analysis are of type ReachingDefinitionFlowMap, which extends CellularFlowMap<ImmutableDefinitionSet>. An **ImmutableDefinitionSet** is an AbstractSet where all the update methods are prohibited. As in the case of PointsToAnalysis, we use immutable sets here so that we can use CellularFlowMap<> as type of flow maps for this analysis, and also so that we can reuse the sets.

At any program point (node), each entry in the internal map of the flow map at that point maps a cell to a set of Definitions that may have defined the cell, and that may reach that point. Some key points to note concerning ReachingDefinitionFlowMap :

- There are two constructors : (i) one of the constructors takes an ExtensibleCellMap<ImmutableDefinitionSet> (from here on, referred by ExtensibleCellMap within this section), and sets that map as the internal map within the newly constructed flow map, and (ii) the another constructor takes another flow map, and sets the internal map of the newly constructed map as a copy of that of the given argument.
- The debug string corresponding to this analysis is *"rd"*.
- The **meet()** operation takes two ImmutableDefinitionSets, and returns an ImmutableDefinitionSet as follows : If both the given sets are `null`, then a new empty ImmutableDefinitionSet is returned. Otherwise, if one of the sets is `null`, then the another set is returned. If both the given sets are equal, then any of the sets is returned. Otherwise, the union of both the sets (a new object) is returned.

Below, we discuss certain key points to note concerning ReachingDefinitionAnalysis :

- **The TOP flow map.** The getTop() method simply returns a new ReachingDefinitionFlowMap on each invocation. This flow map contains an empty map from cells to sets of definitions.
- **The entry flow map.** The getEntryFact() is used to obtain the flow map which will be the IN flow map for the first node in the control-flow graph of the `main()`.

  For each variable symbol declared in the global scope, an entry from symbol to a set (ImmutableDefinitionSet) containing just its Definition (which is comprised up of the defining node and the symbol being defined), is added to a newly created ExtensibleCellMap. Similarly, to this map, we also add entries for both the parameters of `main()`. Finally, a new ReachingDefinitionFlowMap is created using this map, and returned as the entry flow map.
- **The transfer functions.** This analysis relies on the following two methods to specify the transfer functions for various kinds of leaf CFG nodes :
  - *initProcess()*. Invoked for a node and IN flow map at the node, this method is used to obtain the corresponding OUT flow map.

    ***UPDATE:* "If the node is a BeginNode of a FunctionDefinition, or an EndNode of a FunctionDefinition, or of a CompoundStatement, then this method simply returns a copy of the IN flow map, to ensure that the scope-relevance code does not break. "**

    Using AllDefinitionGetter, this method collects the set of definitions that exist within the given node. Note that each possible write within the node corresponds to a definition. If there are no definitions in the node, the IN flow map is returned as the OUT flow map;

***UPDATE:* "however, if the node is a PostCallNode, then a copy of the IN flow map is returned."**

Otherwise, the set of cells that may have been redefined in the node, referred as redefinedCells, is obtained from the collected definitions.

A new OUT flow map is created using the IN flow map. Depending upon the nature of the set redefinedCells, there are three cases, under which we modify the internal map of the OUT flow map as follows :

* *Case 1. The set redefinedCells is universal.* For each key in the non-generic key set of the internal map, we add a new definition (the only definition from this node) to the set of definitions corresponding to that key, by creating a new ImmutableDefinitionSet object. If universal cell exists as a key in the internal map, we perform the same operation for its set as as well.
* *Case 2. The set redefinedCells is not universal, and is not singleton.* In this scenario, we traverse over the cells that are present in the redefinedCells. For each such cell that may have been (re)defined in this node, we first fetch the set of definitions corresponding to it from the internal map. If no definition exists, we assume the initial set to be empty. To this set, we add the definition corresponding to that cell. Then, we create a new ImmutableDefinitionSet from this set, and put it as the value for that cell in the internal map.
* *Case 3. The set redefinedCells is not universal, but singleton.* In this case, we create a singleton set of definitions, containing only the sole definition that exists within this node. In the internal map, we add an entry from the sole cell that is present in the set redefinedCells, to a new ImmutableDefinitionSet object that contains the newly created singleton set of definitions.

***UPDATE:* "Note that now we do not recreate and store set values for a mapping if the existing set, if any, is equal to the new set to be stored."**

– **writeToParameter()**. This method is used to model the effects of the assignment of an argument to its parameter, on the flow map. Given an IN flow map (of type ReachingDefinitionFlowMap), a new map (of type ExtensibleCellMap) is created using the internal map of the IN flow map. To this newly created map, this method adds an entry from the parameter to a set that contains the definition corresponding to the write of the argument to the parameter. Finally, this map is used to obtain the new OUT flow map, which is then returned by this method.

22.3 Copy-propagation analysis

The class **CopyPropagationAnalysis** provides an inter-thread flow-sensitive context-insensitive sve-sensitive copy propagation analysis. The identifier AnalysisName for this analysis is COPY-PROPAGATION.

The structure of flow maps for this analysis is defined by the type CopyPropagationFlowMap, which extends CellularFlowMap<Cell> (Secion 20.1). At any given node, each entry in the flow map of the form $x \rightarrow y$ denotes the fact that the only reaching definition(s) of x at the node is/are some copy statement(s) of the form $x = y$;, and that the variable y has not been redefined in any of the paths between any of those definition(s) and the given node. The implicit copy relation modelled by such maps is clearly transitive, and commutative in nature. Following are some key methods in a CopyPropagationFlowMap :

- There are two constructors. One of the constructors takes an ExtensibleCellMap<Cell> as an argument and uses it as the internal map of the newly constructed flow map; the another constructor takes another flow map as an argument, and uses an extension of the internal map of that flow map as the internal map of this newly constructed flow map.
- The unique debug string corresponding to this analysis is *"copy"*.
- Given two cells, the **meet()** operation returns a cell as follows : If both the cells are `null`, then `null` is returned. If either of the cells is `null`, then the other cell is returned. Otherwise, the universal cell is returned.

Next, we note some key observations concerning CopyPropagationAnalysis :

- **The TOP flow map.** The getTop() method returns a new CopyPropagationFlowMap upon each invocation. This flow map contains an empty ExtensibleCellMap<Cell>.
- **The entry flow map.** The entry flow map is obtained by an invocation to the method getEntryFact(), which returns a new CopyPropagationFlowMap object, with its internal map (of type ExtensibleCellMap<Cell>) populated as explained next.

  Corresponding to each global Declaration that satisfies the following constraints, an entry is made into the internal map.
  - The declaration is not that of a `typedef`.
  - The declaration does not declare more than one symbol.
  - The declaration contains an explicit initialization of the declared symbol.
  - There is exactly one Assignment in the declaration, as obtained from AssignmentGetter.getInterProceduralAssignments() (refer to Section 23), and that assignment satisfies the following properties :

    * The LHS of the assignment, obtained via Assignment::getLHSLocations(), corresponds to only one cell, which is a Symbol.
    * The RHS of the assignment, obtained via Assignment::getRHSLocations(), corresponds to only one cell, which is a Symbol.

  For each such definition, a mapping from its LHS location to the RHS location is added to the internal map, which is used to create the new CopyPropagationFlowMap to be returned from this method.

- **The transfer functions.** Given an IN flow map and a node, transfer functions are used to obtain the OUT flow map, by modelling the effects of the node on the IN flow map. For copy propagation analysis, the transfer functions are specified using following two methods :
  - **initProcess().**

    Given a node and its IN flow map, this method looks into all assignments within the node, and returns an OUT flow map accordingly.

    ***UPDATE:* "If the node is a BeginNode of a FunctionDefinition, or an EndNode of a FunctionDefinition, or of a CompoundStatement, then this method simply returns a copy of the IN flow map, to ensure that the scope-relevance code does not break. "**

    If there are no writes in the node, then the IN flow map is returned as the OUT flow map; ***UPDATE:* "however, if the node is a PostCallNode, then a copy of the IN flow map is returned."**

    Otherwise, we create a new OUT flow map, with its internal map set as an extension to the internal map of the IN flow map.

    We obtain the list of Assignments in the node, using AssignmentGetter.getInterProceduralAssignments(). The node is marked as a copy instruction if there is only one Assignment corresponding to the node, and that Assignment is itself a copy instruction, as checked using Assignment::isCopyInstruction() (Section 23).

    If this node is not a copy instruction, and if it may be writing to the universal cell, then we empty the internal map of the OUT flow map, and return the map. Otherwise, if this node does not write to the universal map, then for each cell that is written in the node, if there exists any mapping which maps any other cell to the written cell, then we change that mapping to map the other cell to the universal cell. We also add a mapping from the written cell to the universal cell, and then return the OUT flow map.

    If this node is a copy node, then we proceed as follows. Consider that an assignment assigns a cell rhs to the cell lhs. First of all, we iterate over all the keys of the internal map of the OUT flow map, and see if there is any entry that maps rhs to the lhs. If so, we set a flag

foundMirror (initially `false`). For every other entry, if the value is same as the lhs cell, then we replace the mapping of that key to map to the universal cell. After iterating over all the entries, if we find that the flag foundMirror was never set, then we add/update an entry such that the lhsCell maps to the rhsCell. Finally, we return the OUT flow map.

- **writeToParameter().**
  For a given IN flow map at a ParameterDeclaration, this method updates the flow map to model the effect of writing the specified argument to the parameter. If the argument is a constant, or if the cell corresponding to the argument is not a Symbol, then the IN flow map is returned as the OUT flow map. Similarly, if a mapping already exists in the internal map from the parameter to the argument, this method returns the IN flow map. Otherwise, a new OUT flow map is constructed with its internal map as an extension of the internal map of the IN flow map. To this newly created flow map, we add an entry which maps the parameter to the argument. Finally, we return this map.

22.4 Dominance analysis

IMOP provides sve-sensitive dominance analysis for the intra-thread super control-flow graph (i.e., one where the call statements are connected to the called definitions, and where edges exist between flushes to indicate flow of shared data) using the class **DominanceAnalysis**. A node *A* *dominates* a node *B*, if there does not exist any path from the entry node to *B* that does not pass through *A*.

Unlike most of the other instantiations of the generic IDFA pass, this analysis does not have a flow fact which derives from CellularFlowMap<?>. Instead the flow facts of this analysis are of type DominatorFlowFact which extends directly from the FlowFact class. Important members of the class DominatorFlowFact are as follows :

- Each flow fact contains a set of nodes, named dominators. At any given node, say *A*, if the flow fact contains a node, say *B*, then it would imply that *B* dominates *A*.
- The constructor of a flow fact takes a set of dominators, and sets the same object as its internal dominators set.
- Two flow facts are considered equal only of their internal sets are equal.
- The **merge()** method takes a flow fact, and merges it into the receiver flow fact, returning `true` if the receiver flow fact was changed as a result of the operation. The merge operation is defined as follows : Note that an empty internal set represents the universal set of nodes. With that interpretation, the merge operation on a receiver flow fact makes its internal set contain only those elements which are also present in the internal set of argument flow fact (i.e., we take the intersection of the two sets).

Various important methods of the class DomninanceAnalysis are explained below :

- **The TOP flow fact.** The method getTop() is used to obtain the TOP flow fact, which returns a new object containing `null` internal set of dominators (interpreted as the universal set of nodes) upon each invocation.
- **The entry flow fact.** The entry flow fact is obtained by invoking getEntryFact(), which always returns a new object containing a singleton internal set of dominators with the BeginNode of the `main()` as its only element.
- **The transfer functions.** The transfer functions, which are used to model the effect of a node on the IN flow fact to generate the OUT flow fact, are represented using the following two methods :
  - **initProcess().** The IN flow fact is returned as the OUT flow fact, if the internal set of the IN flow fact is (i) `null` (i.e., the universal set), or (ii) already contains the visited node. Otherwise, a new OUT flow fact is created and returned. Its internal set (new object) is obtained by taking the union of the internal set of the IN flow fact, and the singleton set containing the visited node.
  - **writeToParameter().** If the internal set of the IN flow fact is `null`, or already contains the parameter, then the IN flow fact is returned as the OUT flow fact. Otherwise, a new OUT flow fact, which has its internal set obtained by adding the visited parameter to a copy of the internal set of the IN flow fact, is created and returned.

22.5 Control predicate analysis

In class **PredicateAnalysis**, we implement an intra-thread intra-procedural control flow analysis (derived from IntraProceduralControlFlowAnalysis) The unique constant corresponding to this analysis is AnalysisName.PREDICATE_ANALYSIS.

The flow facts of this analysis are of type PredicateFlowFact, which is composed up of an ImmutableSet of ReversePaths; a ReversePath consists of a BeginPhasePoint, and an ImmutableList of BranchEdges; finally, a BranchEdge is made up of a predicate expression, and an integer which is used to identify a specific branch of the predicate. As its name suggests, a ReversePath denotes a path that starts at the last seen branch while traversing backwards from the given node, and ends at either a BeginPhasePoint, or at the entry point of the enclosing function. Note that a ReversePath may possibly be a non-continuous subsequence of the actual path that it denotes. One of the key methods in ReversePath is getNewList(BranchEdge):List<BranchEdge>, which returns a new sub-list object of the list edgesOnPath of the receiver object, such that none of the elements from starting index up til (and including) the first occurrence (if any) of the predicate

of the given BranchEdge in the list is present in the sub-list, and that the given BranchEdge is prepended to the sub-list.

At any program point (node), its PredicateFlowFact provides a set of all those non-cyclic paths to the node which start at some BeginPhasePoint, or entry point of the enclosing function, and do not contain any other BeginPhasePoint in them. Each edge of the path belongs to type BranchEdge, indicating which branch was taken from a given predicate, in order to reach the node.

Note 22.5.1

While a ReversePath is allowed to not contain some of the branches that need to be taken in order to reach a given node via the path represented by that ReversePath, the flow fact PredicateFlowFact at the node must contain at least one ReversePath corresponding to each incoming edge to the node.
Hence, given a set of constraints, if none of the paths present in PredicateFlowFact at a given node is feasible, then the node cannot get executed under those constraints.

Note that no update methods are allowed on any ImmutableSet or ImmutableList; we use them in the construction of PredicateFlowFact so that we can enable reuse of various components of a flow fact.
Some key points concerning PredicateFlowFact are :

- There are two constructors. One of the constructors takes another PredicateFlowFact as its argument and sets the internal controlPredicatePaths:ImmutableSet<ReversePath> of the newly constructed object to be same as the controlPredicatePaths of the argument. Note that two flow fact objects can share the same ImmutableSet object as an ImmutableSet object is, as its name suggests, immutable. Another constructor takes directly takes an ImmutableSet<ReversePath> as its argument, and store it in the field controlPredicatePaths of the newly constructed object.
- The overridding of method FlowFact::**isEqualTo(FlowFact)**:boolean returns `true` if and only if the field controlPredicatePath for the receiver object is equal to that of the argument.
- The getString() method, which is used to print debug information, uses the key *"predFlowFact"*, and prints the string of all elements of the controlPredicatePath within curly braces.
- One of the most important methods in the implementation of PredicateFlowFact is **merge(FlowFact, CellSet):boolean**, which takes an argument flow fact, and changes the receiver flow fact such that the receiver flow fact reflects the merge of its old state with that of the given argument. If the state of receiver flow fact changes as a result of this operation, then this method returns `true`; else `false`. This method works as follows :
  - First of all, this method constructs a union of the sets represented by controlPredicatePath of the receiver and the argument flow facts. On the resulting set of ReversePaths, this method invokes PredicateFlowFact.simplifyPaths(Set<ReversePath>):Set<ReversePath>,

and checks if the set returned by that invocation is equal to the set of ReversePath of the receiver (i.e., whether the set is equal to one represented by controlPredicatePath of the receiver). If the sets are equal, then this method returns `false`. Otherwise, it sets the field controlPredicatePath of the receiver to the set returned by simplifyPaths(), and returns `true`.

– The method simplifyPaths() internally invokes two recursive methods, PredicateFlowFact.fusePredicateBranches(Set<ReversePath>):Set<ReversePath>, and PredicateFlowFact.obtainPrefixPaths(Set<ReversePath>):Set<ReversePath>.
Given a set of ReversePaths, the method fusePredicateBranches() returns another set which is obtained by performing following simplification on the argument set : For any given path, if all the branches of the predicate of the first element (i.e., a branch) of the path are present as the first elements of any of the paths in the set, then the first elements of all those paths are removed where the first element corresponds to that predicate. Note that this process is applied recursively, until a fixed-point is reached.
The method obtainPrefixPaths() takes a set of ReversePaths, and returns another set which is obtained by performing following simplification on the argument set : From the argument set, only those ReversePaths are taken to create the return set for which there does not exist any suffix path in the argument set.

Next, we look into the key methods of PredicateAnalysis :

- **The TOP flow fact.** The TOP flow fact is obtained by invoking getTop() method, which returns a new object upon each invocation, composed up of an empty set of ReversePaths.
- **The entry flow fact.** Since PredicateAnalysis is an intra-procedural analysis, the entry flow fact is required to process the entry point of not just the main() function, but also for that of every reachable function. To obtain the entry flow fact, we override the method getEntryFact() which returns a new object upon each invocation, denoting a singleton set of ReversePaths, containing an empty ReversePath that starts at `null` (i.e., the BeginPhasePoint is set to `null`).
- **The transfer functions.** The transfer functions of a flow analysis are used to model the effect of the execution of a given node on the given flow fact, and return the resulting flow fact. As before, such functions should never change the internal states of the argument flow fact; however, they are allowed to return their argument flow fact as it is. Various transfer functions corresponding to this analysis are described next.

  The method edgeTransferFunction(PredicateFlowFact, Node, Node):PredicateFlowFact is used to model the effect of taking an edge between the two nodes, on the given flow-fact. It works as follows :

– If the predecessor node is a predicate expression, then first of all, we obtain the BranchEdge corresponding to this predicate and the given successor node. For any given path in the argument flow fact, if the path already contains this BranchEdge, then we simply add that path to the set corresponding to the flow fact to be returned, in order to ensure that we only maintain a set of non-cyclic paths. If a path does not contain this BranchEdge, then we prepend the BranchEdge to that path, by invoking ReversePath::getNewList(BranchEdge):Set<ReversePath> on the path (explained earlier in this section), to ensure that no cycles exist in the resulting path.
– Otherwise, the argument flow fact is returned as it is.

In order to specify the transfer function of ParameterDeclarations, we implement the method assignBottomToParameter() as an identity function.

The visits of a BarrierDirective, and of BeginNode of any ParallelConstruct, return a new flow fact, which is composed up of a singleton set of ReversePath that contains a ReversePath which contains only the BeginPhasePoint corresponding to the visited node.

Finally, in the visit of an EndNode of any ParallelConstruct, we simply return a copy of the IN flow fact of the corresponding BeginNode. If the BeginNode has not yet been processed, then we simply add the EndNode to the workList, and return a flow fact with empty set.

23 GETTING ASSIGNMENTS IN A NODE

An **Assignment** comprises of two Nodes – lhs and rhs, and represents assignment of the rhs to the lhs. To obtain the set of cells that may be represented by the lhs or the rhs, the methods getLHSLocations() and getRHSLocations() are used. In getLHSLocations(), if the node lhs is a Declarator or a NodeToken, then the corresponding Symbol is found and added to the singleton set to be returned; if lhs a UnaryExpression, then the set of locations represented by lhs is obtained using ExpressionInfo::getLocationsOf() and returned. Similarly, the sets are obtained from getRHSLocations(), which looks into the node rhs, which could either be an Expression or a NodeToken.

The method **Assignment::isCopyInstruction()** is used to check whether the assignment is a copy instruction. It returns `true` if assignment is a copy instruction. An assignment is not considered to be a copy instruction if :

- there is any typecast in the rhs,
- the number of cells that can be represented by lhs or the rhs is not exactly one each,
- either of the cells represented by lhs and rhs are not Symbols,
- either of those cells are summary nodes (i.e., when the cell is a HeapCell or FieldCell, or if the type of the cell is ArrayType, StructType or UnionType), or

- the rhs cell is an ArrayType, such that the rhs expression contains a BracketExpression or an AdditiveOptionalExpression.

Otherwise, the method returns `true`.

Given a node, in order to obtain all the Assignments that may get executed within the node, we use the class AssignmentGetter, which provides two key methods: (i) **AssignmentGetter.getLexicalAssignments()**, which captures only those assignments which are present lexically within the given node, and (ii) **AssignmentGetter.getInterProceduralAssignments()**, which captures all the assignments that are present anywhere within the given node, or in the functions called from within the node. Both these methods take all the lead nodes present within the given node (lexically, or inter-procedurally), and call the visitor AssignmentExtractor, to get a list of Assignments. Following are the key visits in this visitor:

- **InitDeclarator.** If any initializer exists in an InitDeclarator, its visit creates an Assignment with the initializer as the rhs, and the declarator as the lhs.
- **OmpForInitExpression.** In this visit, an assignment is created to model `id = expression`.
- **NonConditionalExpression.** If the operator is =, then an assignment is created for the LHS and RHS of the expression.
- **PreCallNode**. This visit is called only via getInterProceduralAssignments(). For each possibly called definition of the corresponding CallStatement, one assignment is created for each pair of parameter and argument.
- **PostCallNode.** This visit is called only via getInterProceduralAssignments(). If the node has an identifier to receive the returned value, then an assignment to the identifier is created for expression of each ReturnStatement of the each possibly called definition of the corresponding CallStatement.

Note that expressions within UnarySizeofExpression and SizeofUnaryExpression do not get executed. Hence, the visitor does not visit within such nodes.

Note 23.0.1

Currently, the extractor does not model following kinds of assignments: (i) those which take a short-hand operator (e.g., +=, -=, etc.), (ii) those which use pre/post increment/decrement operators (++, and --), and (iii) any assignments that occur within OmpForReinitExpression. Hence, the list of assignments returned by the methods of class AssignmentGetter is not exhaustive.

24 SINGLE-VALUED EXPRESSIONS, AND CO-EXISTENCE CHECKS

An expression is termed as a *single-valued expression*, if in any given runtime phase, each thread would observer same value of the expression.

With the help of SVE information for the predicates, one can make the phase information more precise. However, the number of SVE-sensitive phases can be exponential in terms of number of predicates in the program, in the worst case. Hence, we instead provide methods that can emulate the SVE-sensitivity for phases, on demand.

Two nodes are said to *co-exist* in an (abstract/static) phase if there may exist a runtime phase of the given phase such that both the nodes may get executed in that runtime phase.

In this section, we look into some relevant methods that are defined in the classes **SVEChecker**, and **CoExistenceChecker**.

SVEChecker.isSingleValuedPredicate(Expression):boolean. Given a predicate (such as predicate of an if-else statement), this method checks if the predicate is single-valued in all its possible runtime phases. An expression is termed as single-valued in a runtime phase if all threads would evaluate that expression to the same value in that phase. (Note that this value may, and frequently does, change across phases.)

If the expression is not a predicate (checked using Misc.isAPredicate()), then this method returns false, conservatively assuming that the expression is not single-valued. Otherwise, this method calls its recursive variant SVEChecker.isSingleValuedPredicate(), by passing it the expression, along with two empty sets. The value returned by this invoked method, is returned back.

The method SVEChecker.isSingleValuedPredicate() takes three arguments – (i) a predicate expression, exp, to be tested, (ii) a set of expressions, expSet, upon which the SVE check is ongoing, and (iii) a set of NodePair, nodePairs, which contains the pair of Nodes which are being checked for *co-existence* (explained later in this section).

If the expression exp is `null`, or if the static flag disable is set to `true`, then, conservatively, this method returns `false`.

Otherwise, if the expression exp is already under testing in the recursion chain, then this method returns `true`.

For efficiency purposes, we maintain two sets of nodes, singleValuedExpressions and multiValuedExpressions, which are used to cache the results of invocation of this method. These sets are static fields of SVEChecker. If an expression is present in singleValuedExpressions, then it implies that the expression has already been checked and found to be single-valued; whereas, if an expression is present in the set multiValuedExpressions, then it implies that the expression has already been found to be multi-valued (conservatively).

Hence, if exp exists in singleValuedExpressions, this method returns `true`, whereas it returns `false`, if exp is present in multiValuedExpressions.

Otherwise, the expression exp is added to the set expSet, to indicate that its testing for single-valuedness has begun.

If exp does not contain any reads of cells, then its value must be same across all its executions. Hence, we add it to singleValuedExpressions, remove it from expSet to mark the completion of its testing, add it return `true` from this method.

If exp may read and write to the same location, then conservatively we assume it to be multi-valued. We add the exp to multiValuedExpressions, remove it from expSet, and return `false`.

Next, we process each cell that may have been read within exp as follows :

- If the cell is an AddressCell which belongs to a private variable, then all threads would read different values for the cell. Hence, we add exp to multiValuedExpressions, remove it from expSet, and return `false`. If the AddressCell belongs to a shared variable, we ignore the cell, as all threads would observe the same address for a shared variable.
- Otherwise, if the cell is a FieldCell or a HeapCell, then conservatively we assume that different threads may access different parts of the cell. Hence, we add exp to multiValuedExpressions, remove it from expSet, and return `false`.
- Similarly, if the cell is a FreeVariable, we conservatively assume that different threads may read different values from the cell. Hence, we add exp to multiValuedExpressions, remove it from expSet, and return `false`.
- Otherwise, if the cell is a Symbol, we proceed as follows : If the symbol is a FunctionType, then its value (the function that it denotes) would be constant for all the threads. In such a case, we ignore this read.

  Otherwise, if the cell is an ArrayType, then its value (which is same as the address of its first element) would be same for all threads, if the array is shared at the program point where exp exists; otherwise, it will be different for all the threads. Hence, if the array is a shared variable, we ignore this read. Otherwise, we add exp to multiValuedExpressions, remove it from expSet, and return `false`.

  Otherwise, if the symbol is a shared variable at exp, and if it has been written anywhere in the phase (checked using CoExistenceChecker.isWrittenInPhase(), which is explained later in this section), then different threads may observe different values, depending upon whether they encounter exp before or after any of the write(s) of the symbol. Hence, in this case, we add exp to multiValuedExpressions, remove it from expSet, and return `false`. Otherwise, we ignore this read of the shared variable.

  If the symbol is a private variable, then we invoke the method SVEChecker.ensureSameValue() (explained later in this section) for the symbol,

and exp, passing the expSet and nodePairs arguments, to check whether all threads would indeed read same value for the thread. If this invocation returns `false`, then we add exp to multiValuedExpressions, remove it from expSet, and return `false`. Otherwise, we ignore the read.

In this manner, after processing each cell that may have been read within exp, if we have not yet returned, then we consider exp to be single-valued. We add it to singleValuedExpressions, remove it from expSet, and return `true`.

SVEChecker.ensureSameValue(Symbol,Node,Set<Expression>, Set<NodePair>):boolean. This method takes a private variable sym at a given expression, exp, along with two sets – (i) a set of expressions, expSet, which contains all those expressions which are undergoing the single-valuedness test, and (ii) a set of NodePairs, nodePairs, which contains all those pairs of nodes for which the *co-existence* check is ongoing. Given these arguments, this method tests if in each runtime phase, all threads would observe the same value for the private variable (i.e., whether all private copies of the variable would contain the same value, for a given runtime phase). (Note that the value may differ across the runtime phases.)

The set of reaching definitions for sym at exp (or rather, at the CFG leaf node that contains exp), will be `null` or empty, if the reaching-definition analysis is under-way and had not completed yet. In such cases, we conservatively return `false`, declaring the variable may take different values for different threads.

If the set of reaching definitions is singleton, then for all the threads, same definition must have been executed for writing the last value to the variable. Hence, we check whether the expression that denotes the value written by the defining node is singe-valued or not, by invoking the method SVEChecker.writesSingleValue() (described later in this section); if it is, we return `true`, else `false`.

If the set of reaching definitions is not `null`, empty, or singleton, then we process each definition as follows :

- If the definition does not write a single-valued expression to the variable, then we return `false`.
- Since we have multiple reaching definitions for the given variable, all threads would read the same value only if there does not exist any definition which can be executed by only some of the threads and not all, in any given runtime phase.
  Hence, we invoke the method CoExistenceChecker.existsForAll() on the defining node, which would return `true` only if all control-predicates of the defining node are single-valued expressions. If this invocation returns `false`, then we return `false`; otherwise, we ignore this reaching definition and test the next one.

Finally, after having checked all the reaching definitions, if we have not yet returned, we return `true`, declaring that the private variable at exp would have same value for all the threads, for any given phase. (Again, note that this value may differ across phases.)

SVEChecker.writesSingleValue(Node, Set<Expression>, Set<NodePair>):boolean. This method takes a node, and two sets – (i) a set of expressions that are under single-valuedness checks, and (ii) a set of NodePairs that are under co-existence checks. It checks whether the given nodes writes a single-valued expression to some variable.

If the node is a Declaration that contains no Initializer, then the value written would be same for all threads if this declaration is that of a shared variable; if the declaration is that of a private variable, then all threads may see different (garbage) value for the declared variable. Conservatively (and as per the call-sites of this method within this class), we simply assume that the declaration belongs to a private variable, and hence, return `false`. If the declaration contains an Initializer, then `true` is returned only if the initializer itself is a single-valued expression (checked by invoking SVEChecker.isSingleValuedPredicate(), while passing forward both the sets that are used in breaking out of recursive calls).

If the node is a ParameterDeclaration (parameters are always local), OmpForInitExpression, OmpForCondition, or OmpForReinitExpression, then this method returns `false`.

If the node is an ExpressionStatement, we consider it to write a single-valued expression to a variable if the Expression of this expression statement is single-valued in itself (checked by invoking SVEChecker.isSingleValuedPredicate(), and the argument sets).

If the node is a PostCallNode, then we return `false` from the method if no target for the corresponding CallStatement exists, or if the method name is known to return different values for same input (e.g., omp_get_thread_num()). List of all such methods is supposed to be populated in the static set of strings, SVEChecker.variableFunctions. Otherwise, if the list of arguments is empty, then we know that the returned value would be same for all the threads (except for the functions in variableFunctions). Hence, in that case, we return `true`. Otherwise, we return `false` only if there exists at least one argument which is not a single-valued expression; else, we return `true`.

If the nodes is a PreNode, then it implies that there were no known targets for the corresponding CallStatement, and conservatively we had assumed that PreCallNode has written to all the cells that are reachable from the arguments and globals. Hence, we return `false` from this method, assuming that different values may have been written by different threads.

Otherwise, if the node is an Expression, then we conservatively assume that the value written to a variable within the Expression is single-valued only if the Expression itself is single-valued.

CoExistenceChecker.canCoExistInAnyPhase(Node, Node):boolean Given a pair of nodes, say n_1 and n_2, this method checks whether there exists any common phase of n_1 and n_2 in which both the nodes may get executed (in any order, or concurrently).

If the flag Program.sveSensitive is set to SVEDimension.SVE_INSENSITIVE, then this method returns `true` conservatively. Otherwise, if a NodePair corresponding to n_1 and n_2 is present in the set CoExistenceChecker.knownCoExistingNodes, then this method returns `true`; if the pair is present in the set `CoExistenceChecker.knownNonCoExistingNodes`, then it returns `false`.

Otherwise, for each common phase, say *ph* of the nodes, we perform the check of co-existence by invoking CoExistenceChecker.canCoExistInPhase() on *ph* and both the nodes. If the result of any of these invocations is `true`, then we save the pair to knownCoExistingNodes and return `true`. Otherwise, we return `false` after adding the pair to knownNonCoExistingNodes.

CoExistenceChecker.canCoExistInPhase(Node, Node, Phase):boolean Given a pair of nodes, and a phase, this method invokes its recursive counterpart for these arguments by passing empty sets of NodePair and set of Expressions. This method is also utilized to calculate the amount of time spent in SVE-related tasks.

CoExistenceChecker.canCoExistInPhase(Node, Node, Phase, Set<NodePair>, Set<Expression>):boolean This method takes two nodes, say n_1 and n_2, and a phase *ph*, to check if the nodes can co-exist in *ph*. It also maintains two sets to handle recursive queries of co-existence and SVEness – (i) nodePairs:Set<NodePair> is a set of unordered pairs of nodes, on which the co-existence queries are underway, and (ii) expSet:Set<Expression>, is a set of expressions which are undergoing SVE checks.

If the flag Program.sveSensitive is set to SVEDimension.SVE_INSENSITIVE, then this method conservatively returns `true`. Otherwise, we perform a sanity check which ensures that both n_1 and n_2 must be present in the phase *ph*.

Note 24.0.1

Currently, as a temporary fix for some unknown bug(s), we conservatively return true from this method, if either of n_1 and n_2 does not belong the phase *ph*.

First of all, we check if the NodePhasePair corresponding to n_1, n_2, and *ph* is already present in knownCoExistingNodesInPhase; if so, this method returns `true`. Otherwise, if the NodePhasePair is present in knownNonCoExistingNodesInPhase, or if the NodePair corresponding to n_1 and n_2 is present in knownNonCoExistingNodes, then this method returns `false`.

If the NodePair corresponding to n_1 and n_2 is already present in the argument set nodePairs, then we return `true`, to ignore recursive constraints. (Note that we should not cache the result

during this return.) Otherwise, we add the NodePair to the argument set, indicating that now we are starting the processing of co-existence check on n_1 and n_2.

In order to perform the co-existence check, we first obtain the PredicateFlowFacts corresponding to both the nodes. For each node, its PredicateFlowFact contains a set of ReversePaths, such that for each barrier-free path from any entry point of any phase to the node, there must exist at least one ReversePath in the set which contains at least one of the branches from that path. Since the co-existence query is asked in the context of phase *ph*, we consider only those

Note 24.0.2

Currently, due to some unknown bug, we need to handle the scenario where one or both of the nodes have not been processed by PredicateAnalysis. We do so by conservatively returning `true` from this method.

ReversePath elements from a PredicateFlowFact whose BeginPhasePoint :

(i) is `null`,

(ii) is one of the entry points of *ph*, OR

(iii) contains *ph* in its set of phases (this case would occur when the node corresponding to the BeginPhasePoint is present in some nested parallel construct of the parallel construct of *ph*).

After selecting the sets of relevant ReversePaths for both the nodes, we invoke the method CoExistenceChecker.haveAnyValidPathPairs() on both the sets to check if any *valid* path pair may exist between these set (as explained later in this section). If so, then we add the NodePhasePair of the arguments in knownCoExistingNodesInPhase, the NodePair in knownCoExistingNodes, and return `true`; otherwise, we add the NodePhasePair to knownNonCoExistingNodesInPhase and return `false`. Note that before returning from either of these paths, we also remove the NodePair from the argument set nodePairs.

CoExistenceChecker.haveAnyValidPathPairs(Set<ReversePath>, Set<ReversePath>, Phase, Set<NodePair>, Set<Expression>):boolean. Given two sets of paths, say path1Set and path2Set, this method checks if there exists any pair $(p1, p2) \in$ path1Set $\times$ path2Set, such that (p_1, p_2) contains no contradicting valuations of any of the SVE predicates.

We test each pair of ReversePaths, $(p_1)$for validity as follows :

- If the BeginPhasePoints of both the ReversePaths exist, then we check if they can co-exist in the phase *ph* (or its predecessor, as the case may be), by invoking CoExistenceChecker.canBarriersCoExistInPhase() (explained later in this section). If the BeginPhasePoints cannot co-exist, we ignore this pair of paths.
- Otherwise, if there exists any pair of unequal branches, say $(b_1, b_2) \in p_1 \times p_2$, such that both the branches belong to the same predicate, and the predicate is a single-valued expression

(checked using SVEChecker.isSingleValuedPredicate()), then it implies that the paths are inconsistent. In such a case, we ignore this pair of paths.

- Otherwise, we consider this pair of paths to be valid, and return `true` from this method.

Finally, if no valid pair of paths is found, this method returns `false`.

CoExistenceChecker.canBarriersCoExistInPhase(BeginPhasePoint, BeginPhasePoint, Phase, Set<NodePair>, Set<Expression>):boolean. This method is used to check if the given BeginPhasePoints may co-exist in the context of the phase *ph* (or its predecessor, as the case may be). If either of the BeginPhasePoints is `null`, we return `true` from this method.

Otherwise, there are three possible scenarios :

Case 1: *Both the BeginPhasePoints are entry points of the phase.* In this case, we check if the nodes corresponding to the BeginPhasePoints can co-exist in any of the predecessor phases of *ph*, by invoking CoExistenceChecker.canCoExistInPhase() on both the nodes and a predecessor phase. If so, we return `true` from this method; else `false`.

Case 2: *Only one of the BeginPhasePoints is an entry point of the phase.* In this case, the BeginPhasePoint which is not an entry point of *ph*, must be present in some nested parallel construct within *ph*. We test for its co-existence with any of the successors of the other BeginPhasePoint in *ph*, using CoExistenceChecker.canCoExistInPhase(). If so, we return `true` from this method; else `false`.

Case 3: *Neither of the BeginPhasePoints are entry points of the phase.* In this case, the nodes corresponding to both the BeginPhasePoints should be present within *ph*; if not, we conservatively return true. Otherwise, we return the result of co-existence check of both the nodes in *ph*.

CoExistenceChecker.existsForAll(Node, Set<Expression>, Set<NodePair>):boolean. Given a node, this method is used to check whether it is guaranteed that the node will either be encountered (executed) by all the threads or none of them, in any given runtime phase. For this to be the case, all the control predicates of the node must be single-valued expressions.

This method inspects each phase in which the leaf CFG node, which contains (or is) the given node (termed as *node* in the rest of this section), as follows :

- If there exists any entry point of the phase (i.e., a BeginPhasePoint), from which the node is not reachable, but whose successor [11] may co-exist with the node in the phase, then it implies that a thread may as well take any of the paths that start at that successor, and never encounter the given node, whereas other threads might. Hence, we return `false` in this case.

[11] We take successor of a BeginPhasePoint, as the BeginPhasePoint itself would not be a part of the phase under consideration.

- Next, we collect a set of all those predicates in the phase that lie on any path between starting of the phase and the node, except for those predicates whose non-leaf parents are static control parts (SCOPs) [12]and which do not contain the node within them. To obtain such a set of predicates, we invoke CollectorVisitor.collectNodeSetInGenericGraph() (Section 17), with the following arguments :
  (i) the given node (converted to a NodeStack, with an empty CallStack),
  (ii) an empty set (as we ignore the endPoinst that is populated by this method),
  (iii) a lambda for termination check, which, given a node, returns `true` if the node is a BarrierDirective, or is a BeginNode of a ParallelConstruct.
  (iv) a lambda for getting neighbours, which, given a node, returns a set of predecessors of the node by invoking CFGInfo::getParallelConstructFreeInterProceduralLeafPredecessors(), and, additionally, performs the following operation: if the collected predecessor is a predicate (Misc.isAPredicate() returns `true`), and if the predecessor's parent non-leaf node is either not SCOPped or the non-leaf node contains the node (NodeInfo::isSCOPPed(Node) returns `false`), then we add the predecessor to a special set predicatesToBeChecked.
- Once this invocation completes, we test all the predicates stored in the set predicatesToBeChecked as follows :
  – If the predicate is an OmpForCondition, we return `false`, as not all threads may evaluate the condition to same value in any given phase.
  – If the predicate contains only one successor (i.e., it is a compile-time constant), then we ignore the predicate.
  – If the predicate is a not a single-valued predicate, then we return `false`, as some threads may take that branch of the predicate from which the node is reachable, whereas other threads may take the other branch, in this phase.

After processing every phase in which the node may exist, if we have not returned yet, we return `true`, indicating that the given node is guaranteed to be executed by either all the threads, or none of them, in any given phase.

***UPDATE:* " Wed Aug 21 13:55:39 IST 2019. Now, we also maintain a cache for results of existsForAll(). "**

CoExistenceChecker.isWrittenInPhase(Node,Symbol,Set<Expression>, Set<NodePair>):boolean. Given a symbol, this method checks whether it has been written anywhere in any of the phases in which the given node may get executed.

[12]We term a non-leaf node as SCOP (or *control-confined*), if the control can enter the non-leaf node only from a single entry point, and leave it only from a single exit-point.

In this method, we iterate over all nodes in all the phases of the given node (rather, of the CFG leaf node in which the given node exists), and check if the symbol may have been written in any of the iterated nodes. If so, we invoke CoExistenceChecker.canCoExistInPhase() on the iterated node, the given node, and the phase being iterated, to check if the nodes may *co-exist* in the given phase. If the nodes pass the co-existence check, then we return `true` from this method. Otherwise, if no such node exists, then we return `false`.

25 FIXED-POINT STABILIZATION OF CFG

In Section 7, we have seen that construction of some CFG edges depends on whether the *end* of any CFG node is *reachable* or not (checked using CFGInfo::isEndReachable()). When control can flow to the immediately succeeding element of a given node, without help of any labels, then the end of the node is considered to be reachable. To recap,

(i) end of JumpStatement is never considered reachable,
(ii) end of every other leaf node is considered to be reachable, and
(iii) end of a non-leaf node is considered to be reachable if and only if its EndNode has at least one predecessor.

In other words, for a non-leaf node, its end-reachability depends on whether there are any incoming edges to the EndNode of that non-leaf node. Hence, addition or removal of a CFG edge may trigger addition or removal of other CFG edges due to changes in the end-reachability of one or more non-leaf nodes [13].

Following is a list of CFG edges that should exist only when their source nodes are end-reachable.

- The edge from body of any FunctionDefinition, ParallelConstruct SectionConstruct, SingleConstruct, TaskConstruct, MasterConstruct, CriticalConstruct, AtomicConstruct, OrderedConstruct, or SwitchStatement, to its EndNode.
- The edge between the body of ForConstruct to its step-change expression, OmpForReinitExpression.
- The edge between any two consecutive elements of a CompoundStatmenet; and the edge from the last element of a CompoundStatement to the EndNode of the CompoundStatement.

Note 25.0.1

Unlike other edges listed here, an end-unreachable element, say e_1, of a CompoundStatement may still have an edge to the succeeding element, say e_2, if e_1 is a GotoStatement whose target is a label that is annotated on e_2.

[13] Note that this section logically belongs to the Section 28; however, we discuss it here before discussing the elementary transformations as it is utilized to add and remove CFG edges, which is one of the main tasks of the elementary transformations.

- Edges from the then-body and else-body (if any) of an IfStatement to the EndNode of that IfStatement.
- Edges between body of a WhileStatement or a DoStatement to their respective predicate Expressions.
- The edge between body of a ForStatement to either the step expression, termination expression, or to itself (whichever exists, checked in that order).

In the class **EndReachabilityAdjuster**, there are following methods that help ensure that when an EndNode becomes reachable or unreachable, then the rules mentioned above are automatically triggered :

- **EndReachabilityAdjuster.updateEndReachabilityAddition(Node)** takes a non-leaf node that has recently been made end-reachable, and applies the aforementioned rules by creating new CFG edges from the non-leaf node to its successor, as per the rules, with the help of the visitor **EndReachabilityAdder**, which derives from CFGLinkVisitor. In this visitor, for each edge that needs to be added, the corresponding visit invokes CFGInfo::connectAndAdjustEndReachability() on the source and destination nodes (explained later in this section).
- **EndReachabilityAdjuster.updateEndReachabilityRemoval(Node)** takes a non-leaf node that has recently been made end-unreachable, and applies the aforementioned rules by removing CFG edges from the non-leaf node to its successor, as per the rules, with the help of the visitor **EndReachabilityRemover**, which derives from CFGLinkVisitor. In this visitor, for each edge that needs to be removed, the corresponding visit invokes CFGInfo::disconnectAndAdjustEndReachability() on the source and destination nodes (explained later in this section).

During elementary transformations, when a CFG edge needs to be added or removed, we use the methods CFGInfo::connectAndAdjustEndReachability() and CFGInfo::disconnectAndAdjustEndReachability(), respectively. In these methods, we apply the following rules inductively, until a fixed-point is reached :

- If the sole incoming edge of a node is removed, then the node is marked as unreachable, and all its outgoing edges are removed as well.

  This may lead to removal of all incoming edges of an EndNode, which may trigger removal of further edges, as mentioned above.
- If an incoming edge is added to a node that was previously unreachable, then the node becomes reachable, and as per the rules of CFG creation (Section 7), outgoing edges are created from that node.

This may lead to addition of an incoming edge to a previously unreachable EndNode, which may trigger addition of further edges, as mentioned earlier.

Note 25.0.2

In our approach for updating the CFG as a result of changes in end-reachability of a node, we are imprecise in marking an EndNode as unreachable when the corresponding *dead-code* connected to that EndNode contains a cycle.

Note 25.0.3

While updating multiple CFG edges for a set of nodes, it is recommended to first add all the desired edges, and then remove the edges that need to be removed. This would reduce the chances of having spurious toggle of end-reachability of various nodes.

We discuss both these methods in detail next.

25.1 Addition of a CFG edge

Method **CFGInfo::connectAndAdjustEndReachability(Node, Node):void** is used to add a CFG edge between the provided source and destination CFG nodes. If addition of this CFG edge may update the end-reachability of any node, then this method also performs other required changes in the CFG using EndReachabilityAdjust.updateEndReachabilityAddition().

If the given source or destination is `null`, the method connectAndAdjustEndReachability() simply returns. Otherwise, it changes their reference to the respective CFG nodes of the arguments. Then, invoking the method CFGGenerator.verifyEdgePrecision() (Section 7), it checks if construction of a CFG edge between the given nodes can be ignored due to the source being a static constant predicate. If so, then this method returns. Otherwise, it adds the source node to the list of predecessors of the destination, and the destination node to the list of successors of the source. Upon addition of this CFG edge, if the destination now has exactly one predecessor, then it implies that it was previously unreachable, but now it is reachable. In such a case, this node should then be connected, as a source, to the appropriate destination, as per CFG generation rules. We achieve so by invoking EndReachabilityAdjuster.updateEndReachabilityAddition() if the destination is an EndNode; otherwise we use **NextNodeJoiner.joinNextNode()**, which internally invokes NextNodeJoinVisitor, a subclass of CFGLinkVisitor. Note that the rules followed in various visits of NextNodeJoinVisitor are same as the ones followed during CFG generation (Section 7).

25.2 Removal of a CFG edge

Method **CFGInfo::disconnectAndAdjustEndReachability(Node, Node):void** is used to remove a CFG edge from between the provided source and destination CFG nodes. If removal of

this CFG edge may update the end-reachability of any node, then this method also performs other required changes in the CFG using EndReachabilityAdjust.updateEndReachabilityAddition().

If the given source or destination is `null`, then the method disconnectAndAdjustEndReachability() returns; otherwise, it make them refer to their respective CFG nodes instead. Then, this method removes the source node from the list of predecessors of the destination, and the destination node from the list of successors of the source. Upon removal of this CFG edge, if the destination now has no predecessors, then it implies that it was previously reachable, but now it is unreachable. In such a case, this node should then be disconnected, as a source, from the appropriate destinations, as per CFG generation rules. We achieve so by invoking EndReachabilityAdjuster.updateEndReachabilityRemoval() if the destination is an EndNode; otherwise we use **NextNodeDisjoiner.disjoinNextNode()**, which internally invokes NextNodeDisjoinVisitor, a subclass of CFGLinkVisitor. Note that the rules followed in various visits of NextNodeDisjoinVisitor are same as the ones followed during CFG generation (Section 7).

26 ELEMENTARY TRANSFORMATIONS

Any transformation that adds/modifies/removes any of the CFG components of any non-leaf CFG node, or adds/removes labels on statements, is termed as an *elementary transformation* in IMOP. The set of elementary transformations available in IMOP is quite exhaustive – *any* valid transformation within a function (i.e., the executable part of a program), can be expressed as a series of elementary transformations [14].

Note 26.0.1

Note that for any given non-leaf CFG node, its BeginNode and EndNode components cannot (and should not) be updated.
Similarly, all *leaf* CFG nodes of IMOP are considered immutable via elementary transformations. In other words, there does not exist any elementary transformation which can update the AST contents of a leaf CFG node. While one can access and alter the AST components of a leaf CFG node via other means, it is not recommended, as no guarantees of automated update of program abstractions are provided in such cases.
When we need to modify the contents of a leaf node, we should instead create a new modified leaf node and replace the existing node with the new one. For example, while attempting to rename a variable in an ExpressionStatement, we should create a new ExpressionStatement with the updated variable name, and use it to replace the old ExpressionStatement [a].

[a] To replace an old node with a new node, one can use the method NodeReplacer.replaceNodes(Node, Node).

[14] While IMOP also provides methods to add/modify/remove global declarations/definitions (of variables, types, typedefs, and functions), such methods are not yet termed as *elementary transformation* as in their current state they need not provide guarantees of automated update of all program abstractions (but only few) upon their invocation.

One key guarantee provided by each elementary transformation is that the state of each program abstraction would automatically be made consistent with the modifications performed by the elementary transformation on the program. Such update may happen *eagerly*, i.e., before the elementary transformation is considered complete, or *lazily*, i.e., before the first use of any affected data structure, as explained in detail in Section 28.

As most of the elementary transformations, by definition, alter the CFG components of a non-leaf node, they are present in the subclasses of CFGInfo. The other elementary transformations, which manipulate labels of a Statement (leaf or non-leaf node), are present in the class StatementInfo.

In this section, we categorize and discuss various elementary transformations in terms of the non-leaf nodes on which they are specified.

For each transformation, we specify the steps taken to modify the AST, and CFG edges, along with label annotations of the affected nodes, wherever required. Note that any update to CFG edges also stabilizes the *end-reachability* of affected nodes implicitly. Automated update/invalidation of other program abstractions (such as, IDFA flow facts, MHP information, etc.), under any of the elementary transformations, are explained in detail later in Section 28.

Note 26.0.2

In this document, when we discuss any method of a NodeInfo or a CFGInfo object corresponding to an AST node, we refer to the AST node by the phrase *owner node*.

Common methods for updating CFG. Before starting with inspection of each non-leaf node separately, we look into some basic methods that help us in updating the CFG. In Section 25, we have already noticed how addition and removal of CFG edges using CFGInfo::connectAndAdjustEndReachability() and CFGInfo::disconnectAndAdjustEndReachability() can internally ensure the stabilization of *reachability* and *end-reachability* of any affected nodes. During any elementary transformation on a non-leaf node, we use these methods in the corresponding subclasses of CFGInfo, to add and remove CFG edges that are defined as per the semantics of the non-leaf node. For handling the update of CFG edges involving JumpStatements and Labels we use the following common methods from class IncompleteSemantics :

Note 26.0.3

For any elementary transformation that may remove a node from the program, the updates to CFG must be performed before the updates to the AST, whereas the order of updates should be reversed while adding a node to the program.

- **IncompleteSemantics::adjustSemanticsForOwnerRemoval():void**. This method is used while removing a node from the program. For each JumpStatement which is lexically present

within, or is, the node to be removed, if target of the JumpStatement (obtained via calls to getTarget() of respective subclasses of CFGInfo) is not present in the node to be removed, then we invoke CFGInfo::disconnectAndAdjustEndReachability() on the JumpStatement and its target.

If a Statement that is lexically present within, or is, the node to be removed contains any SimpleLabels, then we inspect all its predecessor GotoStatements that correspond to any SimpleLabel on the Statement. If any such GotoStatement is not present within the node to be removed, we remove the CFG edge connecting that GotoStatement to the Statement.

If there exists any Statement lexically within the node (or which is the node itself), such that it contains a CaseLabel whose corresponding SwitchStatement is not present within the node to be removed, then we remove the CFG edge connecting the predicate of that SwitchStatement to the Statement.

Similarly, if any Statement lexically within the node (or which is the node itself), contains a DefaultLabel whose SwitchStatement does not reside within the node to be removed, we perform the following two actions :

- Using connectAndAdjustEndReachability(), we add a CFG edge between the predicate and EndNode of the SwitchStatement.
- Using disconnectAndAdjustEndReachability(), we remove the CFG edge between the predicate of the SwitchStatement and the Statement.

- **IncompleteSemantics::adjustSemanticsForOwnerAddition():void**. This method is used to handle the CFG edges corresponding to JumpStatements and Labels, while adding a node to the program.

  For each JumpStatement which is lexically present within, or is, the added node, if target of the JumpStatement (obtained via calls to getTarget() of respective subclasses of CFGInfo) is not present in the added node, then we invoke CFGInfo::connectAndAdjustEndReachability() on the JumpStatement and its target.

  If a Statement that is lexically present within, or is, the added node, contains any SimpleLabel, we obtain its outer-most non-leaf CFG node, and search for all possible GotoStatements that may have this Statement as their target. If any such GotoStatement is present outside the added node, we connect it to the Statement using a CFG edge.

  If there exists any Statement lexically within the node (or which is the node itself), such that it contains a CaseLabel whose corresponding SwitchStatement is not present within the added node, then we add a CFG edge connecting the predicate of that SwitchStatement to the Statement.

Similarly, if any Statement lexically within the node (or which is the node itself), contains a DefaultLabel whose SwitchStatement does not reside within the added node, we perform the following two actions :

- Using connectAndAdjustEndReachability(), we add a CFG edge between the predicate of the SwitchStatement and the Statement.
- Using disconnectAndAdjustEndReachability(), we remove the CFG edge between the predicate and EndNode of the SwitchStatement.

- **IncompleteSemantics::adjustSemanticsForOwnerSwitchPredicateRemoval():void**. This method is used to update portions of the CFG when the predicate of a SwitchStatement is removed. It is invoked on the body of the affected SwitchStatement.

  First of all, it obtains a set of those statements which contain a CaseLabel and/or a DefaultLabel corresponding to the affected SwitchStatement, using SwitchRelevantStatementsGetter. Then, for each such collected statement, it removes the CFG edge which connects the predicate of this SwitchStatement with the collected statement, using disconnectAndAdjustEndReachability().
- **IncompleteSemantics::adjustSemanticsForOwnerSwitchPredicateAddition():void**. This method too is invoked on the body of a SwitchStatement to which a new predicate has been added. It takes care of the CFG edges that connect predicate to relevant cases and default labeled statement.

  Firstly, it collects the set of all those statements that contain a CaseLabel and/or DefaultLabel relevant to the affected SwitchStatement. To all those statements, this method creates a CFG edge from the predicate of the SwitchStatement, using connectAndAdjustEndReachability().
- **IncompleteSemantics:: adjustContinueSemanticsForOwnerForLoopExpressionRemoval():void**. This method is used to alter those CFG edges in a loop which may connect various internal ContinueStatements to loop's predicate (or step expression) that has to be removed. It is invoked on the body of the loop being affected.

  This method begins with collecting the set of all those ContinueStatements which correspond to the same loop as the one being affected. Then, it removes the CFG edges between all such ContinueStatements and their sole successors.
- **IncompleteSemantics:: adjustContinueSemanticsForOwnerForLoopExpressionAddition():void**. This method is used to alter those CFG edges in a loop which may connect various internal ContinueStatements to loop's predicate (or step expression) that has been recently added. It is invoked on the body of the loop being affected.

This method begins with collecting the set of all those ContinueStatements which correspond to the same loop as the one being affected. Then, it adds a CFG edge between all such ContinueStatements with their targets (as obtained with ContinueStatementCFGInfo::getTarget()).

Now, we discuss how various elementary transformations alter other CFG edges that are created as per the semantics of non-leaf nodes.

26.1 Labels of a statement

In this section, we discuss various methods that are used to alter the labels of a statement, and look at how we update the AST, and CFG edges under each such transformation. Note that no OpenMP statements are allowed to have labels, as they start with `pragma`'s, which cannot have labels annotated on them [15]. All these methods are present as member methods of the class StatementInfo. They are discussed next in detail :

- **addLabelAnnotation(int, Label):void**. This method takes a Label object, and adds it at the specified index (starting with zero) in the list of label annotations (annotatedLabels) of the owner statement.

Note 26.1.1

Note that all elementary transformation methods of label annotations first ensure that the owner node is not any OpenMP construct/directive, as labels cannot be applied to `#pragma` directives. Then, if the user has invoked a transformation on any non-CFG statement node (say, by mistake) then that method recursively invokes itself on the corresponding CFG statement instead, and returns the result of that invocation, if any.

If the label annotations of the CFG statement already contains the Label at the specified index, we return back from this method. Otherwise, we first need to remove the Label from its previous statement, if any, before adding it to the owner CFG statement. We do so by invoking StatementInfo::removeLabelAnnotation() on the current labeledCFGNode, if any, of the given Label.

Note 26.1.2

Although statements of different FunctionDefinition may use label with same string, these labels cannot be same SimpleLabel object, as they contain a field labeledCFGNode which can only point to one statement on which the label has been annotated. Similarly, we cannot reuse the same CaseLabel or DefaultLabel objects across different SwitchStatements.

[15] Currently, we do not check whether any attempts are made by the users of IMOP to add labels to an OpenMP statement. While this is possible as per the grammar, it is not valid as per semantics of the C language. This is a minor TODO for later.

Depending upon the type of label being added, we also remove some other conflicting labels from other statements in the context, using **StatementInfo::removeSimilarLabelsFromRelevantContext()** as described next.

- In case of a SimpleLabel, we remove all those other labels that have same name as that of the SimpleLabel and are annotated on any CFG node (except the owner node) within the outer-most non-leaf CFG node that encloses the owner node.
- For CaseLabel and DefaultLabel, we first need to obtain the enclosing SwitchStatement, if any, of the owner node; if none exists, then we obtain the outer-most non-leaf CFG node that encloses the owner node. Then, we collect all the relevant statements, as defined below :
  * If an enclosing SwitchStatement of the owner node is found, we collect all those statements within that SwitchStatement which contain any CaseLabel or DefaultLabel that may be a target of the predicate of that SwitchStatement.
  * Otherwise, we collect all those statements within the outer-most non-leaf CFG encloser of the owner node which contain annotations of CaseLabel and/or DefaultLabel that do not have any corresponding (i.e., enclosing) SwitchStatement.

  Note that the owner node is not considered as a relevant statement.

  Now, if the added label is a CaseLabel, we remove all those CaseLabel's from the relevant statements which have the same case-expression string as that of the added CaseLabel; if the added label is a DefaultLabel, we remove all DefaultLabel annotations from the relevant statements.

Then, this method adds the given label to the annotatedLabels of the statement, at the specified index. It also updates the field labeledCFGNode of the Label to make it refer to the owner node.

Finally, this method invokes StatementInfo::updateUponLabelAddition() to perform automated update of various program abstractions, which is required as a result of addition of this Label. Among update of various other abstractions (as discussed in Section 28), this method also performs updates in the CFG-edges by invoking **StatementInfo::adjustSemanticsForLabelAnnotation(Label, LabelUpdateMode): Set<Node>**, which works as follows :

- This method takes a Label that has been added to the owner node, and performs required update of the CFG edges, while returning a set of those nodes which have been added as predecessors of the owner statement as a result of addition of this Label.
- If the added label is a SimpleLabel, this method searches for all those GotoStatements within the outer-most non-leaf CFG node enclosing the owner node, such that the name of label of the GotoStatement matches the name of the added SimpleLabel. From

all these collected nodes, this method adds a CFG edge to the owner node, using CFGInfo::connectAndAdjustEndReachability(). Finally, the set of collected GotoStatements is returned by this method.
  - If the added label is a CaseLabel, then this method searches for the enclosing SwitchStatement of the owner node; if no such SwitchStatement is found, this method returns an empty set. Otherwise, this method adds a CFG edge between the predicate of the SwitchStatement and the owner node, using CFGInfo::connectAndAdjustEndReachability() [16].
  - When the added label is a DefaultLabel, this method searches for an enclosing SwitchStatement for the owner node; if none is found, it returns an empty set. Otherwise, the following two edits are performed on the CFG :
    (i) a CFG edge is added using CFGInfo::connectAndAdjustEndReachability() between the predicate of the SwitchStatement and the owner node, and
    (ii) the CFG edge between predicate of the SwitchStatement and its EndNode is removed, using CFGInfo::disconnectAndAdjustEndReachability().
- **removeLabelAnnotation(Label):boolean**. This method removes the specified label from the label annotations of the owner node. If the label was present in the label annotations, this method returns `true`, else `false`.

  Before removing the label from annotatedLabels of the owner node, this method invokes StatementInfo::updateUponLabelRemoval() to trigger automated update of various program abstractions. Apart from update of other program abstractions, updateUponLabelRemoval() also invokes **StatementInfo::adjustSemanticsForLabelRemoval(Label, LabelRemovalMode): Set<Node>**, which updates the CFG edges as follows :
  - This method takes a label that has to be removed from the owner node, and performs update to the CFG edges, while returning a set of those nodes that will be removed as predecessors of the owner node as a result of removal of this label.
  - If the label to be removed is a SimpleLabel, we collect all its predecessor GotoStatements, and invoke CFGInfo::disconnectAndAdjustEndReachability() to remove the edges from GotoStatements to the owner node. Finally, the set of collected GotoStatements is returned back by this method.
  - When the label to be removed is a CaseLabel or a DefaultLabel, we first obtain the enclosing SwitchStatement; if none is found, we return an empty set. Otherwise, we proceed as follows.

[16] Note that if the predicate is a static-time constant, which does not evaluate to the case of the added CaseLabel, then no edge should be created between the predicate and the owner node. However, this check is performed internally within connectAndAdjustEndReachability(), hence we can invoke it without performing this check explicitly.

If the added label is a DefaultLabel, then we add a CFG edge between the predicate and the EndNode of the obtained SwitchStatement.

Next, we need to check whether there will be any other labels on the owner node that would make it a target of the predicate of the SwitchStatement. If so, then we should not remove the CFG edge; otherwise, we remove the edge between the predicate and the owner node using CFGInfo::disconnectAndAdjustEndReachability(). Finally, we return a set containing the predicate, if any edge was removed.

Note 26.1.3

Note that no information about a new CFG edge being created between the predicate and the end-node of the SwitchStatement is conveyed back to the callee. Check if this may create any issues in the automated update of various abstractions.

Once the update method returns, we remove the label from annotatedLabels, and set the field labeledCFGNode of the Label to `null`.

Note that there also exists an overloaded version of this method which takes a String argument. That method searches for a SimpleLabel whose name matches the given String, and then removes it using this method.

- **clearLabelAnnotations():void**. This method is used to clear the label annotations of the owner statement.

  For each label annotation of the owner node, we invoke StatementInfo::adjustSemanticsForLabelRemoval(), to perform automated update of CFG edges as explained above. Then, we set the labeledCFGNode field of that label to `null` and the label from annotatedLabels, before processing the next label annotation.

26.2 Function definition

In case of a FunctionDefinition, we do not currently support the following elementary transformations of its signature [17] :

- Setting a new parameter-declaration list.
- Clearing away the existing parameter-declaration list.
- Removing a specific parameter-declaration from the list.
- Adding a specific parameter-declaration at a specific position in the list.

The only implemented elementary transformation of a FunctionDefinition, present in FunctionDefinitionCFGInfo is

[17] All the missing transformations of a FunctionDefinition have been added as TODOs in IMOP. Until then, an inefficient way to achieve these transformations is to build a new function altogether, and replace the existing function with the intended modified one.

- **setBody(CompoundStatement):void**. This method changes the current body of the owner FunctionDefinition with the provided CompoundStatement.

  If the provided body is same as the current body, this method returns. Otherwise, we first set the owner node as the parent field of the given CompoundStatement. (We need to check whether any automated update requires this.)

  Then, using FunctionDefinitionCFGInfo::updateCFGForBodyRemoval(), we remove the CFG edges that connect the old body to the owner node. In updateCFGForBodyRemoval(), we invoke we remove all possible edges to and from the body being removed, as per the semantics of CFG generation (Section 7) for FunctionDefinition.

  After returning back from method updateCFGForBodyRemoval(), we set the AST fields of the owner node to connect it to the provided body in the AST. Finally, we add CFG edges to connect the owner node to the provided body, using FunctionDefinitionCFGInfo::updateCFGForBodyInsertion(). In updateCFGForBodyInsertion(), we create all the CFG edges to and from the added body as per the CFG generation rules of FunctionDefinition.

26.3 Omp parallel construct

Following are the methods that enable elementary transformations of a ParallelConstruct.

- **setBody(Statement):List<UpdateSideEffects>**.
- **setIfClause(IfClause):void**.
- **removeIfClause():boolean**.
- **setNumThreadsClause(NumThreadsClause):void**.
- **removeNumThreadsClause():boolean**.

Note that other applicable clauses of a ParallelConstruct do not contain any executable units (expressions) within them. Hence, they are not considered as CFG components of the ParallelConstruct. In order to update the clauses of a ParallelConstruct, one can directly use the overloaded methods ParallelConstruct::addOmpClause(); these methods do not (need to) trigger automated update of any program abstractions.

26.4 Omp for construct

Following elementary changes can be performed to the CFG components of a ForConstruct (which denotes an `omp for`).

- **setBody(Statement):List<UpdateSideEffects>**.
- **setInitExpression(OmpForInitExpression):void**.
- **setForConditionExpression(OmpForCondition):void**.
- **setReinitExpression(OmpReinitExpression):void**.

26.5 Omp sections construct

In a SectionsConstruct, following elementary transformations can be applied to update its CFG components.

- **addSection(Statement):List<UpdateSideEffects>**.
- **removeSection(Statement):boolean**.
- **clearSectionList():void**.
- **setSectionList(List<Statement>):void**.

As SectionConstruct (note, this is Section, not Sections), is not a CFG node, if we wish to change the body of a section, we can as well create a new SectionConstruct, and replace the existing one with the new one [18].

26.6 Omp single construct

The only CFG component of a SingleConstruct that can be changed is its body, via the elementary transformation **setBody(Statement):List<UpdateSideEffects>**.

26.7 Omp task construct

The CFG components of a TaskConstruct can be modified with the help of following elementary transformations.

- **setBody(Statement):List<UpdateSideEffects>**.
- **setIfClause(IfClause):void**.
- **removeIfClause():boolean**.
- **setFinalClause(FinalClause):void**.
- **removeFinalClause():boolean**.

26.8 Omp master construct

The only elementary transformation applicable to a MasterConstruct is **setBody(Statement):List<UpdateSideEffects>**.

26.9 Omp critical construct

In case of a CriticalConstruct, the only CFG component of it that can be updated is its body, for which one can use **setBody(Statement)::List<UpdateSideEffects>**.

Note that modifications to the region name of a CriticalConstruct are not considered as elementary transformations.

[18] This is similar to how we update leaf CFG nodes; note that SectionConstruct is not a leaf CFG node, though.

26.10 Omp atomic construct

In order to update the body of an AtomicConstruct, one can use the method **setBody(Statement):List<UpdateSideEffects>**.

No other elementary transformations exist for an AtomicConstruct.

26.11 Omp ordered construct

The body of an OrderedConstruct can be modified using the elementary transformation **setBody(Statement)::List<UpdateSideEffects>**.

26.12 Compound statement

Elementary transformations of a CompoundStatement are usually the most frequently used elementary transformations. A CompoundStatement is a block that is composed up of a list of Declarations and/or Statements. Following are the elementary transformations for a CompoundStatement.

- **addDeclaration(Declaration):List<UpdateSideEffects>**.
- **addStatement(Statement):List<UpdateSideEffects>**.
- **addElement(Node):List<UpdateSideEffects>**.
- **removeDeclaration(Declaration):List<UpdateSideEffects>**.
- **removeStatement(Statement):List<UpdateSideEffects>**.
- **removeElement(Node):List<UpdateSideEffects>**.
- **clearElementList():void**.
- **setElementList(List<Node>):void**.

26.13 If statement

In case of an IfStatement, following elementary transformations exist.

- **setPredicate(Expression):void**.
- **setThenBody(Statement):List<UpdateSideEffects>**.
- **setElseBody(Statement):List<UpdateSideEffects>**.
- **removeElseBody():void**.

26.14 Switch statement

In case of a SwitchStatement, note that its various cases do not create any syntactic blocks. They are simply labels to which the control can jump from the predicate, depending upon the value of the predicate at runtime. Hence, the only CFG components of a SwitchStatement that can be modified using elementary transformations are its predicate and its body.

- **setBody(Statement):List<UpdateSideEffects>**.
- **setPredicate(Expression):void**.

26.15 While statement

Following are the elementary transformations of a WhileStatement.

- **setBody(Statement):List<UpdateSideEffects>**.
- **setPredicate(Expression):void**.

26.16 Do-while statement

In case of a DoStatement, IMOP provides the following elementary transformations :

- **setBody(Statement):List<UpdateSideEffects>**.
- **setPredicate(Expression):void**.

26.17 For statement

Following is an exhaustive list of all the elementary transformations that are applicable on a ForStatement (which represents a serial `for` loop in C).

- **setBody(Statement):List<UpdateSideEffects>**.
- **setInitExpression(Expression):void**.
- **removeInitExpression():void**.
- **setTerminationExpression(Expression):void**.
- **removeTerminationExpression():void**.
- **setStepExpression(Expression):void**.
- **removeStepExpression():void**.

26.18 Call statement

For simplicity, we do not allow modifications to the PreCallNode or PostCallNode of a CallStatement. In other words, a CallStatement is an immutable object under the context of elementary transformations. (While not recommended, one can still access and alter the AST components of a CallStatement.) In order to make any changes to a CallStatement, one should instead construct a new modified CallStatement, and use it to replace the current CallStatement.

27 MISCELLANEOUS METHODS/VISITORS

- Misc.getCFGNodeFor(Node) is used to obtain either the immediately enclosed, or the immediately enclosing CFG node for a given non-CFG node; if the provided node is a CFG node, then the node itself is returned by this method.

- Misc.getDeclarator(Declaration, String) is used to obtain the Declarator corresponding to the provided identifier in the given declaration.
- Misc.getIdNameList(Declaration) is used to obtain a list of identifier names declared in the given declaration.
- CFGInfo.getLexicalCFGLeafContents(Node) returns a set of all CFG leaf nodes that that are lexically contained within the given node.
- CFGInfo.getIntraTaskCFGLeafContents(Node) is used to collect the set of CFG leaf nodes that may be present anywhere within the given node, including the function bodies that are called from within the node. Note that the traversals are done only on valid paths.
- NodeInfo::isConnectedToProgram() is used to check whether the associated node is connected to the main AST.
- Misc.getInheritedEnclosee(Node, Class<T>) returns a set of those nodes that are present in the AST sub-tree of the provided node, and are of type T (or its subtype).
- Type.hasIncompatibleTypeCastOfPointers(Node) is used to check if the given node contains any incompatible typecasting of pointers.
- Misc.getSimplePrimaryExpression(Expression) returns true, if the provided node is a simple primary expression; otherwise, it returns false.
- RootInfo::getAllFunctionDefinitions() provides a list of all the function definitions that are present in the associated translation unit.
- Misc.getInternalFirstCFGNode() is used to obtain the first CFG node that's encountered in the DFS traversal on the AST of the base node (i.e., the argument). However, if the base node is a CFG node, the node itself is returned.
- Misc.getClauseList(Node)
- Misc.getCaseDefaultLabelStatementList()
- Misc.isAPredicate()
- DeclarationInfo::getInitializer().
- NodeInfo::getOuterMostNonLeafEncloser()
- Misc.getEnclosingNode(node:Node, className:Class<Node>) returns a node, if any, that is of type className, and encloses (with no other encloser in between of the same type) the given node.
- Misc.getEnclosingBlock(Node):Scopeable returns the enclosing CompoundStatement, FunctionDefinition, or TranslationUnit, for the given node (exclusively).
- DeclarationInfo::hasInitializer(),
- RootInfo::removeDeclarationEffects(). (See if this can fully support automated update.)
- Conversion.getUsualArithmeticConvertedType(Expression, Expression).

- Type::getIntegerPromotedType().
- Type.getTypeFromArithmeticKeys().
- Type.getTypeTree().
- Misc.getSymbolEntry().
- CFGInfo::getAllComponents().
- SectionsConstructCFGInfo::getSectionList() is used to obtain a list of CFG nodes that represent the body of all the sections in the given node.
- StatementInfo::getLabelAnnotations().
- Misc.getInheritedPostOrderEnclosee() is used to obtain a post-order traversal list of nodes that are of specified type and are present within the given node. This method relies on the visitor PostOrderInheritedCollector.
- NodeInfo::getAllCellsAtNode(), and its overridden definitions at RootInfo, FunctionDefinitionInfo, and CompoundStatementInfo.
- CompoundStatementCFGInfo::getElementList().
- CellCollection::applyAllExpanded() is used to apply the passed lambda on cells of the receiver collection.
- NodeInfo::getAllSymbolNamesAtNodeExclusively(), NodeInfo::getAllCellsAtNodeExclusively(), NodeInfo::getAllCellsAtNode(), and NodeInfo::getAllSymbolNamesAtNode().
- RootInfo::getAllFunctionDefinitions(), RootInfo::getFunctionWithName(), and RootInfo::getMainFunction().
- NodeInfo::getReachableCallStatementsInclusive(), and NodeInfo::getLexicallyEnclosedCallStatements(), as described in Section 15.
- NodeInfo::getReachableCallGraphNodes().
- FunctionDefinitionInfo::getCallersOfThis(), FunctionDefinitionInfo::getCalledDefinitions(), and FunctionDefinitionInfo::getCalledSymbol().
- NodeInfo::getEnclosedScopesInclusive() is used to obtain a set of all those scopes that are present lexically within the given node (including the node itself).
- Type::getAllTypes().
- StructType::getDeclaringNode(), UnionType::getDeclaringNode(), and EnumType::getDeclaringNode().
- Misc.getTypedefEntry().
- CFGInfo::getInterProceduralLeafSuccessors(), and CFGInfo::getInterProceduralLeafPredecessors().
- CFGInfo::getInterTaskLeafSuccessorEdges(*), and CFGInfo::getInterTaskLeafPredecessorEdges(*).
- CFGInfo::getIntraTaskCFGLeafContents().
- CFGInfo::getInterTaskLeafSuccessorList.

- Misc.isCFGNode(), Misc.isCFGLeafNode(), and Misc.isCFGNonLeafNode().
- AnalysisDimension.
- Misc.getBufferedWriter(String):BufferedWriter, takes a filename, creates it, if it does not already exist, and returns a BufferedWriter to it, which can be used to write to the file.
- NodeInfo::getSharingAttribute().
- NodeInfo::isSCOPPed().
- CFGLinkVisitor, and its subclasses.
- CFGLinkFinder.getCFGLinkFor().