IMOP: IIT Madras OpenMP

A Self-Stabilizing Compiler Framework for OpenMP C

AMAN NOUGRAHIYA, IIT Madras
V. KRISHNA NANDIVADA, IIT Madras

Part C: A Quick Start Guide

IIT Madras OpenMP (IMOP) is an open-source compiler framework designed for writing program analysis, profiling, instrumentation, and source-to-source optimization tools for OpenMP C programs. IMOP is implemented in Java and aims to provide an easy-to-use and efficient framework for implementing research prototypes of various compilation tools.

This quick-start guide aims to concretely and succinctly illustrate how to meet some of the most common requirements of any compilation pass using IMOP. It dives only as deep into any topic as might be required by a majority of the end-users (that is, compiler-pass writers). The in-depth details of any topic should be present in the first two parts (Part A: Technical Report, or Part B: Code Review Document) of this three-part documentation.

Manuscript under preparation.

HOWTO: **Prepare an input program for compilation with IMOP**

IMOP's parser conforms to the grammar of OpenMP 4.0 and ANSI C. It takes only a *preprocessed* C file as input. For example, all the header files in the input program must have been expanded and all macros preprocessed, before the program can be passed as input to IMOP.

Nearly all major compilers provide appropriate switches to obtain preprocessed files. Given an OpenMP C file `foo.c`, its preprocessed version can be obtained using GCC as follows:

```
$ gcc -P -E foo.c -o foo.i
```

The above command will create the preprocessed version of `foo.c` in file `foo.i`[1].

HOWTO: **Invoke the parser**

Following is a simple demonstration of how IMOP's parser can be invoked for parsing a given input program file:

```
public AnyClass {
  public static void main(String []args) {
    Program.parseNormalizeInput(args);
  }
}
```

When using IMOP from command-line, the input file (say `foo.i`) can be given as a command-line argument using the switch `-f`, for example, as follows:

```
$ java AnyClass -f foo.i
```

When using IMOP from within Eclipse, the full path (absolute or relative) of the input program file can be specified by storing it as string to the static member **`Program.filePath`**, in method `Program.defaultCommandLineArguments`. See this method for various such examples.

The method **`parseNormalizeInput`** performs two steps: (i) parses the given input program to generate the AST, and (ii) performs some normalizations on the program[2]. This method sets the static member **`Program.root`** (accessible using **`Program.getRoot`**) to refer to the root node (of type `TranslationUnit`) of the generated AST.

[1] Refer to Appendix A for some common issues that may occur during this step (and notes on how to handle them).

[2] See Section 2.2 from IMOP's Technical Report, for details on various normalizations performed.

HOWTO: **Understand the AST representation of the program**

The full grammar of IMOP's parser is available at [imop-home]/grammar/modified.html. All non-terminals are represented by a class of their own; all terminals are denoted by the class NodeToken. Some other special nodes, such as NodeListOptional, NodeOptional, NodeChoice, and so on, are used by the parser to support EBNF-form of the grammar. Additionally, as part of its simplification, IMOP also creates some other type of *internal* nodes, such as CallStatement, PreCallNode, PostCallNode, SimplePrimaryExpression, DummyFlushDirective, and so on. The exhaustive list of all such special nodes can be found at the package imop.ast.node.internal. All these internal nodes are explained either later in this guide or in the tech-report if they do not appear here. All nodes, internal and external, are subclass of the class Node.

In general, out of all types of AST nodes, a high-level understanding of around 35-40 types, listed in Appendix B should suffice for most situations, when writing a compiler pass. It is suggested for the user to familiarize herself with the provided set.

Since IMOP's grammar closely matches to that of ANSI C and OpenMP 4.0, the meaning of each type of the AST node remains same across the grammar specification in the standards and its implementation in IMOP.

Note that AST is the *base* representation of the program under compilation. IMOP also provides various other program representations, such as CFG, CFG components, call graphs, phase-flow graphs, and so on, which are far easier and intuitive to work with as compared to an AST. All such representations are automatically kept consistent with one another in response to any transformations performed in IMOP (under some rules, explained later).

HOWTO: **Understand Scopes in IMOP**

The notion of nested scoping, such as that constructed using nested blocks of code, where each block has its own symbol/type/typedef table, and also derives from the parent scope, is represented using the interface Scopeable in IMOP.

Three main kinds of scopes, each of which implements the Scopeable interface are: TranslationUnit, FunctionDefinition, and CompoundStatement. (Note that these are the same objects that appear as part of the AST.)

Later in this guide, we will learn how to traverse across encapsulating scopes.

HOWTO: **Print AST nodes**

IMOP provides various utility methods to print AST nodes (and other strings) to files. Three of the frequently used ones are:

- To print the current text-equivalent of the whole program to a file named foo-test.i:

```
DumpSnapshot.forceDumpRoot("test");
```

- To print the current state of the node, say `tempNode`, to a file named foo-test.i:

```
DumpSnapshot.forcePrintToFile(tempNode, "foo-test.i");
```

- To print the given string, say *"tempString"*, to a file named temp-string.i:

```
DumpSnapshot.forcePrintToFile("tempString", "foo-test.i");
```

Note that the `toString` method of the `Node` superclass has already been overridden to provide the pretty-printing string of the node. Hence, to print the current text of a while-statement, say `whileStmt`, to standard output stream, we can simply write:

```
System.out.println(whileStmt);
```

HOWTO: **Work with** `NodeInfo` **Objects**

With each AST node, there exists a corresponding `NodeInfo` object, which contains (directly or indirectly) information about the AST node along various dimensions. Given a node `tempNode`, its info-object can be retrieved using:

```
NodeInfo nodeInfo = tempNode.getInfo();
```

For most of the important AST nodes, a special subclass of `NodeInfo` is maintained. For example, the info-object for a while-statement will be of exact type `WhileStatementInfo`. Such specialized classes also contain various node-specific functionalities. For instance, to unroll a while-statement, we can simply invoke the following:

```
whileStmt.getInfo().unrollLoop(int);
```

To obtain the function-definition for `main` function:

```
Program.getRoot().getInfo().getMainFunction()
```

To obtain all function-definitions in the program:

```
Program.getRoot().getInfo().getAllFunctionDefinitions()
```

Corresponding to each node of interest, the user should peruse both, the `NodeInfo` class, as well as its specialization, if any, for the node, in order to understand the AST-related utilities. However, note that, as mentioned earlier, AST is not the easiest representation to work with. Other representations will be discussed later in the guide.

HOWTO: **Query Nodes Down the AST**

One common requirement during program analysis and transformations is to be able to obtain all nodes of a given AST type (such as if-statements) *within* a given base node (such as a while-statement). Here, the term *within* is used to refer to the subtree of the base node. IMOP provides a set of methods to achieve this functionality.

To obtain a set of all if-statements in the AST sub-tree of a given while-statement (say, `whileStmt`):

```
Misc.getExactEnclosee(whileStmt, IfStatement.class);
```

To obtain a list (in post-order traversal), we instead use the method `Misc.getExactPostOrderEnclosee`. Note that none of these methods in the compiler will step into the called-functions in the subtree of the base node, as these methods are applied on the AST and not CFG/CG.

Labels. In order to obtain statement with a given label (say `l1`) within any AST node, say `tempNode`:

```
Statement stmt = tempNode.getInfo().getStatementWithLabel("l1");
```

HOWTO: **Query Nodes Up the AST**

Another common requirement in various passes is to traverse up the AST, searching for nodes that meet certain criteria. IMOP provides higher-level query methods for such tasks as well.

To obtain the immediately enclosing node of some type, say if-statement, for a given base node, say `tempNode`:

```
Misc.getEnclosingNode(tempNode, IfStatement.class);
```

This method is *inclusive* in nature – in case if the base node is of the same type as requested, then the base node is returned back.

To obtain the enclosing function-definition, there is a short-hand:

```
Misc.getEnclosingFunction(tempNode);
```

Similarly, to obtain the enclosing scope:

```
Misc.getEnclosingBlock(tempNode);
```

This node is exclusive in nature.

There are other similar variants provided by IMOP; they can be found in `Misc` class. Also, note that these methods perform traversal in the AST, and not the CFG/CG. Hence, they will not capture the enclosing nodes that enclose the call-sites of the function to which a base node belongs. For such purposes, CFG traversals/queries are recommended.

HOWTO: **Create New Code Snippets in IMOP**

In IMOP, it is quite straightforward and simple to create new snippets of code through string manipulation. Once we obtain the string of the code snippet to be created, we simply need to invoke the parser for the desired AST type. For instance, the following line will create a subtree rooted at node of type `Statement`, denoting the text "`x=2+3;`":

```
Statement newStmt = FrontEnd.parseAndNormalize("x=y+2;",
                                               Statement.class);
```

HOWTO: **Understand the Nested CFG in IMOP**

In order to preserve the scoping information, as well as the nesting information present in the input program, IMOP does not use the flat CFG such as the ones used by most other compilers. Instead, it uses nested CFGs, where a CFG node may contain nested control-flow graph within itself. Specially in the context of OpenMP, let us consider two possible ways in which any construct, say a parallel construct, can be modeled in the CFG: (i) it is expressed using a single node, which contains within itself a reference to the body of the construct (and other relevant clauses), or (ii) it is broken down into two function calls – one denoting the begin, and another the end, of the parallel construct – that are inserted above and below the body, respectively. Unlike most other compilers that work with low-level IR, IMOP takes the former option by using nested CFGs, thereby simplifying various analyses and transformations, by removing any need to manually keep track of the nesting information.

There are two kinds of CFG nodes – (i) those that may contain a nested CFG within them (i.e., a non-leaf CFG node), and (ii) those that cannot (i.e., a leaf node). In Appendix C, we list all the current set of leaf and non-leaf CFG nodes in IMOP. The user should familiarize herself with both these sets, as most compiler passes in IMOP express their desired analyses and transformations in terms of these nodes.

HOWTO: **Obtain CFG Information Objects**

The CFG information object for a node contains all its CFG-related information, such as the list of successors and predecessors. Notice that `CFGInfo` is the superclass of all CFG information objects for various kinds of CFG nodes (such as `WhileStatementCFGInfo` for a `WhileStatement`). For any node, say `tempNode`, its CFG information object can be obtained as follows:

```
CFGInfo info = tempNode.getInfo().getCFGInfo();
```

Note that IMOP automatically creates the CFG during parsing and normalization of the input program. Further, the CFG is kept consistent automatically with all changes to the program.

HOWTO: **Understand CFG Components**

A non-leaf CFG node can be composed up of a number of other non-leaf or leaf CFG nodes. We term all such immediately nested CFG nodes as the *CFG Components* of the non-leaf node. For instance, two key CFG components of a while-statement are: (i) its predicate (an `Expression`), and (ii) its body (a `Statement`).

Given a non-leaf node, its CFG components can be obtained through following forms of invocations:

```
tempNode.getInfo().getCFGInfo().get*();
tempNode.getInfo().getCFGInfo().has*();
```

For example, if `tempNode` is a while-statement, then the predicate of that statement can be obtained as follows:

```
tempNode.getInfo().getCFGInfo().getPredicate();
```

Note that the `has*` forms are available only for those components that are optional (such as various expressions of a for-statement, else-body of an if-statement, and so on).

As mentioned earlier, CFG (and CFG Components) are the key entities on which various analyses and optimizations are expressed. The user should open and check the public methods provided by various subclasses of `CFGInfo`, given in the package `imop.lib.cfg.info`.

HOWTO: **Use Elementary Transformations for Modifying CFG Components**

In the CFG-information object of a non-leaf node, IMOP provides various required setters using which the CFG components of the non-leaf node can be modified. These setters are termed as *elementary transformations* in IMOP. They are of the following forms:

```
tempNode.getInfo().getCFGInfo().set*();
tempNode.getInfo().getCFGInfo().remove*();
tempNode.getInfo().getCFGInfo().add*();
```

For example, if `tempNode` is a parallel construct, then its body can be replaced with a new body, say `newBody`, as follows:

```
tempNode.getInfo().getCFGInfo().setBody(newBody);
```

Note that the last two forms (removers and adders) are applicable only for optional CFG components, as discussed above.

One key advantage of using elementary transformations in IMOP (either directly, or through higher-level CFG transformations discussed next) is that all program abstractions (such as points-to graphs, call graphs, etc.) are automatically kept consistent with the resulting modifications to the program. This property is termed as *self-stabilization*.

HOWTO: **Use Higher-Level CFG Transformations**

In various optimizations, it is quite common to encounter a situation where we need to express a CFG transformation that depends only upon the *contents* of a node (such as, say, a set of shared accesses happening in the node), and not upon *what* the node is present in the program as (such as whether the node is a predicate of some while-statement, or is just some expression-statement itself).

For instance, consider an arbitrary set of CFG nodes that may write to some specific shared location. Let us say we wish to instrument the program in such a manner that a print statement is executed immediately before the execution of any node from the set. Achieving this task using elementary transformations can be quite tricky and tedious – given any node from the set, we do not even know the type of its enclosing non-leaf node; we will have to enumerate all nesting possibilities, and handle each case separately.

To handle such issues, IMOP provides five *higher-level CFG transformations* corresponding to the following cases:

- Ensuring that a CFG node (say newNode) is executed immediately before a given arbitrary CFG node (say baseNode), on all execution paths at runtime.

```
InsertImmediatePredecessor.insert(baseNode, newNode);
```

- Ensuring that a CFG node (say newNode) is executed immediately after a given arbitrary CFG node (say baseNode), on all execution paths at runtime.

```
InsertImmediateSuccessor.insert(baseNode, newNode);
```

- Ensuring that a CFG node (say newNode) is executed in between two consecutive CFG nodes (say basePred and baseSucc).

```
InsertOnTheEdge.insert(basePred, baseSucc, newNode);
```

- Ensuring that a CFG node (say baseNode) is removed from the CFG.

```
NodeRemover.remove(baseNode);
```

- Ensuring that a CFG node (say newNode) is replaced with another node (say baseNode) from the CFG.

```
NodeReplacer.remove(baseNode, newNode);
```

NOTE: *The current version of this quick-start guide does not contain information about various fundamental concepts, such as symbols, types, etc., as well as all advanced concepts such as* dummy-flush directives, inter-task edges, concurrency analysis, Z3-IMOP integration, *and so on. We plan to summarize these and other important concepts in the next version of this guide.*

A COMMON ISSUES ENCOUNTERED WHILE PREPARING A PREPROCESSED FILE FOR IMOP

Following are some key points to note when attempting to generate the preprocessed file for IMOP:

- The content of header files may differ across multiple versions and implementations of GCC. Hence, to ensure that the optimized file generated by IMOP can run with GCC on some machine, do ensure that the aforementioned step of generating the preprocessed file to be given to IMOP is also performed on the same machine.
- GCC provides various *extensions* to the C language. For example, ANSI C and ISO C do not allow declaration of an induction variable as follows:

```
for (int i = 0; i < x; i++) {}
```

  Instead, the variable `i` should have been declared before the `for` statement. However, GCC allows the above given format. IMOP handles only ANSI/ISO C grammar, and does not support such extensions allowed by GCC.
- Similarly, GCC also uses various built-in function declarations and built-in types in its header files. Since the declarations of these functions and types are not present explicitly in the expanded preprocessed file, IMOP may face issues during parsing, type checking, and so on. IMOP explicitly recognizes some of these built-in types, in order to resolve the related parsing errors.

  How to fix such issues for a new type, say `_newtype128`*, being used in the header files present in my system?* The compiler writer should search for all those methods within `FrontEnd.java` where new entries are being added to `CParser.types`. At all such functions, following line should be added:

```
CParser.types.put("_newtype128", Boolean.TRUE);
```

  This should resolve the related parsing error.
- As a result of the such issues, it may so happen that some preprocessed file is not getting parsed successfully by IMOP, despite user intervention. In such cases, using the line-number at which the parsing fails, the user can perform one-time manual task of either fixing or deleting the offending declaration (if it is not being used). If it *is* being used, please contact the developers of IMOP for a fix.

B COMMONLY USED AST NODES

Following is a set of some of the most-commonly-used types of AST nodes:

`TranslationUnit`, `FunctionDefinition`, `Declaration`, `ParameterDeclaration`, `Initializer`, `Statement`, `Expression`, `ExpressionStatement`, `CompoundStatement`, `SelectionStatement`, `IfStatement`, `SwitchStatement`, `IterationStatement`, `WhileStatement`, `DoStatement`, `ForStatement`, `CallStatement`, `PreCallNode`, `PostCallNode`, `SimplePrimaryExpression`, `GotoStatement`, `ContinueStatement`, `BreakStatement`, `ReturnStatement`, `OmpConstruct`, `ParallelConstruct`, `ForConstruct`, `SectionsConstruct`, `SingleConstruct`, `TaskConstruct`, `MasterConstruct`, `CriticalConstruct`, `OrderedConstruct`, `AtomicConstruct`, `OmpDirective`, `BarrierDirective`, `TaskwaitDirective`, `TaskyieldDirective`, `DummyFlushDirective`, and `FlushDirective`.

C CFG LEAF AND NON-LEAF NODES

Following is the list of all 23 leaf CFG nodes, currently present in IMOP:

`Declaration`, `ParameterDeclaration`, `UnknownCpp`, `UnknownPragma`, `OmpForInitExpression`, `OmpForCondition`, `OmpForReinitExpression`, `FlushDirective`, `DummyFlushDirective`, `BarrierDirective`, `TaskwaitDirective`, `TaskyieldDirective`, `ExpressionStatement`, `GotoStatement`, `ContinueStatement`, `BreakStatement`, `ReturnStatement`, `Expression`, `IfClause`, `NumThreadsClause`, `FinalClause`, `BeginNode`, and `EndNode`.

Following are the 16 non-leaf CFG nodes, currently present in IMOP:

`FunctionDefinition`, `ParallelConstruct`, `ForConstruct`, `SectionsConstruct`, `SingleConstruct`, `TaskConstruct`, `MasterConstruct`, `CriticalConstruct`, `AtomicConstruct`, `OrderedConstruct`, `CompoundStatement`, `IfStatement`, `SwitchStatement`, `WhileStatement`, `DoStatement`, and `ForStatement`.