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

PART A : Preliminary Technical Report

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

This technical report introduces IIT Madras OpenMP framework (IMOP), a source-to-source compiler framework, which has been built exclusively for OpenMP C programs. IMOP has been written in Java, with an intention to provide a framework that can be used to efficiently implement research-specific prototypes for large-scale OpenMP C (or serial C) specific analysis and transformation tools. In this preliminary report, we briefly discuss various design and key features of IMOP, along with an appropriate level of implementation detail, wherever required.

Contents

1 INTRODUCTION

IIT **M**adras **O**penM**P** (IMOP) is a compiler infrastructure that aims to enable easy implementation of source-to-source transformation and analysis tools for large-scale OpenMP C programs. This infrastructure helps in querying and rewriting serial and parallel programs by providing various fundamental utilities that are needed to implement a majority of analysis and optimization techniques.

IMOP follows the object-oriented visitor design pattern, and has been written in Java. The parser and certain fundamental visitors of the Abstract Syntax Tree (ASTs), Control Flow Graphs (CFGs), etc., have been written using JavaCC/JTB.

This report looks into the design and key features of IMOP, along with an appropriate level of implementation detail, wherever required. [1]

1.1 Intended Audience

- *Compiler researchers for OpenMP C parallel programs.* IMOP provides various basic building-blocks that one can use to implement new optimization tools for OpenMP C programs. Since IMOP works at the source-code level, implementation of source-to-source transformations and of source code analyses for OpenMP C programs is feasible. IMOP is specifically designed for shared memory OpenMP C programs, and thereby it already contains various elementary analyses (e.g., concurrency analysis) that are specific to such programs. We hope that these reusable modules will result in significant reduction in the development time of new tools.
- *Compiler researchers for sequential C programs.* Along with the OpenMP specific utilities, IMOP also provides various standard analyses and transformations for sequential C programs. Kindly refer to Section 3 to know what is new in IMOP as compared to the other existing frameworks.
- *OpenMP C programmers (end-users).* One aim of IMOP is to provide end-users with enough command-line switches to enable required analyses/transformations, to ease the debugging, comprehension, and profiling of their OpenMP C programs. Furthermore, IMOP also aims to ease the creation of new tools to such an extent that even the end-users can write basic customized tools as per their requirements.

[1] Currently, this manuscript is under preparation. Many important sections might be missing or incomplete. Its latest version can be obtained from http://www.cse.iitm.ac.in/~amannoug/imop

2 STRUCTURE AND SEMANTICS OF IMOP

2.1 Abstract syntax tree

The IMOP Grammar. A stream of valid tokens, generated from a piece of text, is considered as a syntactically valid program for a language if it can be parsed using the grammar of that language. The grammar recognized by IMOP can parse C programs, which may optionally contain OpenMP constructs and directives. This grammar has been derived from a pair of publicly available grammars originally written by Doug South (for C), and Christian Brunschens (for OpenMP). We have added certain missing OpenMP constructs and directives, and made various required changes in production rules of these grammars, to obtain the IMOP grammar. The exact grammar used by the parser of IMOP, is present in Appendix C. From this grammar, which also contains token specifications, we obtained our parser (and default depth-first visitors) using JavaCC/JTB. Corresponding to each non-terminal, there exists an AST class (subclass of class `Node`). All tokens are represented in the AST by using objects of class `NodeToken`.
The AST. Upon parsing an input program, or a code snippet (which may not represent a complete program), by using overloaded static methods in `FrontEnd.parse*(*)`, we can obtain a reference to the root node of the generated abstract syntax tree for the program/snippet. In this section, we do not discuss the meaning of various AST classes in detail, since almost all the classes correspond to well-defined symbols in the grammars specified in the ISO C99 standard and OpenMP standard.

In this section, we briefly discuss some key digressions that the IMOP parser takes from the expected parsing process, while generating the AST for given program/snippet.

2.1.1 Statements. **Labels as annotations.** In C, labels are used to specify targets of goto-statements, and targets of predicate of switch-statements. In IMOP, labels and their corresponding target statements are parsed using the classes SimpleLabeledStatement, CaseLabeledStatement, and DefaultLabeledStatement (all wrapped in objects of LabeledStatement class). While such representation is in accordance with the ANSI C grammar, it is not intuitive to work with. If we wish to obtain the list of labels for a given statement, we need to traverse upwards in the AST to obtain relevant objects of type `LabeledStatement`. Logically, information about labels of a statement should be present within the object representing that statement, rather than as separate AST ancestors for the object. Furthermore, if labels are represented by certain AST ancestors of a statement, then the processes of modifying the labels of the statement, and of adding and removing the statement in the AST, get complicated. Hence, in IMOP, we represent labels as annotations on the target statement.

Whenever a parsed `LabeledStatement` is requested to be attached as a child of a `Statement` object in the AST, the corresponding setter method performs the following : (i) it extracts the target statement for the given `LabeledStatement` object, (ii) it saves the associated label as an annotation on the target statement, and (iii) it attaches the target statement directly as the child of the `Statement` object. The three types of labels, mentioned above, are represented using `SimpleLabel`, `CaseLabel`, and `DefaultLabel` annotations (of type `Label`). Given a statement, its list of labels can be accessed and modified using setter and getter methods of the field `StatementInfo::annotatedLabels`.

2.1.2 Synchronization Constructs. **Flush Directive** In OpenMP, a `flush` directive is used to synchronize a thread's temporary view of the memory with the shared memory. Flush operations can be explicitly specified using

```
#pragma omp flush
```

(optionally with a list of variables to be flushed). Such explicit flushes are represented by FlushDirective class. Apart from these explicit flushes, various implicit flush operations are present at various program points, as specified in the OpenMP standard. In order to handle all types of flushes (explicit or implicit) uniformly, we insert special internal nodes, of class DummyFlushDirective, at those places in the AST where an explicit or implicit flush operation exists as per the OpenMP standard. Each `DummyFlushDirective` node contains a set of memory locations that may get flushed at that node. This set is represented by the field `idSet`. (When `idSet` points to `null`, we assume that it refers to the universal set – set of all shared locations.) The field `dummyFlushType` is used to indicate the type of implicit/explicit flush that a node denotes.

In IMOP, we insert `DummyFlushDirective` nodes at following program points :

(1) As an immediate predecessor of each FlushDirective (Type : `FLUSH_START`). If the directive contains a list of memory locations to be flushed, the set of all those locations is saved as `idSet`.
(2) As an immediate predecessor of each BarrierDirective (Type : `BARRIER_START`).
(3) As immediate predecessor and successor of each CriticalConstruct (Types : `CRITICAL_START` and `CRITICAL_END`).
(4) As immediate predecessor and successor of each OrderedConstruct, if : (i) no OpenMP clause is present, or (ii) `threads` or `depends` clause is present (Types : `ORDERED_START` and `ORDERED_END`).
(5) As immediate predecessor and successor of each AtomicConstruct (Types : `ATOMIC_START` and `ATOMIC_END`). If the atomic construct is present within a non-sequentially consistent region, then the set of memory locations on which atomicity is guaranteed, is saved as `idSet`.
(6) As immediate predecessor and successor of each call to the following OpenMP locking routines : `omp_set_lock()`, `omp_test_lock()`, `omp_set_nest_lock()`, and `omp_test_nest_lock()` (Types : `LOCK_START` and `LOCK_END`.)
(7) As immediate successor of each call to the following OpenMP (un)locking routines : `omp_unset_lock()`, and `omp_unset_nest_lock()` (Type : `LOCK_END`.)
(8) As an immediate predecessor of each explicit task, represented as TaskConstruct (Type : TASK_START).
(9) As an immediate successor of each implicit task (Type : TASK_END).
(10) As an immediate predecessor of each TaskwaitDirective (Type : TASKWAIT_START).
(11) As an immediate predecessor of each TaskyieldDirective (Type : TASKYIELD_START).

Note that all nodes of type `DummyFlushDirective`, are added/removed automatically by the `FrontEnd` parser (when called with *normalization* enabled, using methods like `FrontEnd.parseAndNormalize(*)`). We need to ensure that the `DummyFlushDirective` nodes should remain to be present across various program points, as per the list above, while transforming the program. (Such invariants are preserved automatically if users use appropriate update methods, discussed in Section 2.9, to specify all their higher-level transformations.)

2.2 Simplified AST

The IMOP front-end runs various simplification and normalization passes on the AST of the input program, with an aim to guarantee certain simplifying assumptions about the structure of the AST for the rest of the passes. Such assumptions reduce the number of cases that various analysis and transformation passes need to handle.

In this section, we briefly discuss about these passes, which lead to our simplified assumptions about the transformed AST, which we term as a *Simplified AST*. Note that each Simplified AST still represents a valid C program.

(1) *Simplifying expressions.* In C, an expression is said to have *side-effects* if its evaluation can modify the value of a memory location (or a file), or can access a `volatile` location. For example, `x++` is an expression with a side-effect – it increments the value of `x` by 1. The ISO C99 standard defines the notion of a *sequence point* as follows – when a sequence point exists between two expressions, then the evaluation and side-effects of the first expression must take place before the evaluation of the second expression. For example, sequence points lie between first and second operands of && and `||` operators. (An exhaustive list of sequence points can be found at Annex C of the ISO C99 standard.)

    A single Expression object in the AST of the input program may contain two (or more) operands, separated by sequence point(s), such that one operand updates the value of a variable, and second operand reads from that variable. For example, consider the following expression (assume `x = 1`, initially) :

```
(x++ == 1) && (x == 2)
```

    In this expression, a sequence-point exists between both the operands of && operator. Hence, the read of `x` by the right operand must see the value written by the left operand [2].

    For any data-flow analysis, representation of data-flow facts would be difficult if an expression within a leaf node in the control-flow graph may read from a variable *after* writing to it. (Leaf nodes are those nodes which do not contain any other CFG node within them.) For example, consider constant propagation pass, where a flow-fact of `(x:1)` at the entry of the above expression would not apply to the read of `x` in the right operand of &&. To avoid such unintuitive scenarios in IMOP, we rewrite expressions to remove sequence points from within them. In Table 1, we summarize the relevant transformations [3].

(2) *Simplifying declarations.* Each Declaration object may have more than one declarators with initializers in the AST of the input program. Every initializer immediately precedes a sequence point. Hence, for reasons explained above, we split each `Declaration` with

[2]To compare, note that `(x++) + x` does not contain any sequence point. The value read by the right operand of the + operator is implementation-defined.

[3]Note that these transformations are not applied when the enclosing expression is not present in an evaluation context. For instance, when the enclosing expression is an operand of `sizeof` operator, no simplifications are performed.

Table 1. Summary of expression simplifications.

Simplification	**Input Form**	**Output Form**
Removal of logical AND operator (&&)	`e1 && e2`	Prelude : `int t;` `t = e1;` `if (t) {` `    t = e2;` `}` Replacement : `t`
Removal of logical OR operator (||)	`e1 || e2`	Prelude : `int t;` `t = e1;` `if (!t) {` `    t = e2;` `}` Replacement : `t`
Removal of comma operator (,)	`e1, e2`	Prelude : `e1;` Replacement : `e2`
Removal of conditional operator (?:)	`e1 ? e2 : e3`	Prelude : `int t1;` `int t2;` `t1 = e1;` `if (t1) {` `    t2 = e2;` `} else {` `    t2 = e3;` `}` Replacement : `t2`

multiple declarators to multiple `Declaration`'s with single declarator each. Below is a sample transformation :

```
                                       T x1 = e1;
                                       T x2 = e2;
T x1 = e1, x2 = e2, x3, ..., xn = en;  ⟹  T x3;
                                       ...
                                       T xn = en;
```

(3) *Simplifying call-sites.* An expression representing a function call (using ArgumentList operator) can be present deeply nested within other expressions, which possibly themselves are function calls. Each function call represents a complex flow of control, starting with the evaluation of actual arguments and function designator, to execution of the called function, and ending with an optional return of the computed value. While modelling the effect of a function call statically, most of the flow-sensitive analyses would require to maintain

flow information before and after each function call. It can be difficult to maintain such information for function calls that are deeply nested within complex expressions. To enable easy implementation and representation of flow-sensitive analyses, we simplify expressions involving function calls, such that all function calls are present in one of the two forms :

- A *message-send* : `foo(s1, s2, ..., sn);`
- A *function-call* : `x = foo(s1, s2, ..., sn);`

where `foo` represents the evaluated function name, `x` represents a temporary that holds the return value of the call, and `s1, s2, ..., sn` all are SimplePrimaryExpressions, which can either be an `Identifier`, or a Constant. Note that both the above forms are represented by objects of a subclass of ExpressionStatement, called CallStatement, which is an internal node type (i.e., is not part of the base parser). A `CallStatement` object comprises mainly of these three fields, which represent the structure of the corresponding function-call :

(a) `functionDesignatorNode` : represents a `NodeToken` that represents the identifier representing the called function.
(b) `preCallNode` : an object that contains a list of `SimplePrimaryExpression` representing the actual arguments for the function call. Note that in case of variadic functions, the size of this list will not be same as the size of parameters.
(c) `postCallNode` : an object that optionally contains a reference to the `Identifier` which holds the return value of the function call, if any.

Following example demnostrates how simplification of call-sites is done by the front-end of IMOP :

```
                                                   fp = fptr[3];
                                                   t1 = p[3];
                                                   t2 = bar(2, t1);
i[3] = fptr[3](a * 10, bar(2, p[3]));      ⟹      t3 = a * 10;
                                                   t4 = fp(t3, t2);
                                                   i[3] = t4;
```

Note that the order of evaluation of arguments and function designator is undefined as per the standard. [4]

(4) *Naming anonymous data-types.* While performing code transformations, we often need to declare new temporaries of a given type. If the requested type is an anonymous structure or union, a separate declaration can not be created, due to lack of a name (or tag) for the specified structure/union type. Similarly, if an anonymous structure/union is present as part of a typedef definition, the associated typedef name can not be expanded to its definition, if required. Anonymous stuctures and unions also create complications in representation of data structures involving type information, where most of the times, a user-defined type is referred to by its declared name.

To avoid such issues, the front-end of IMOP provides names for all anonymous structures and unions. Below is an example for the same :

[4]In IMOP, we use the same order of evaluation of arguments and function designator, which is used by MacPorts GCC 6.2.0, our test bed.

```
struct {                          struct __imopSt_s1 {
    int a;                            int a;
    float b;             ⟹            float b;
} v1, v2;                         } v1, v2;
```

Note that if an anonymous struct/union is present as a member of an enclosing structure or union, and if the definition of that anonymous struct/union does not define any objects, then all its members are considered to be members of the enclosing structure or union. For example, in

```
struct s {
        struct {
        int a;
    };
} v;
v.a = 10;
```

`a` is a member of the structure `s`, and can be accessed directly as `v.a` as shown in the example. We do not provide names to such struct/unions since they act as syntactic sugars, rather than as types.

(5) *Encapsulating bodies.* A Statement can be either a CompoundStatement (which encloses other statements and declarations within a pair of braces) or a single statement with no enclosing braces. The body of various syntactic constructs in C and OpenMP is generally represented by a `Statement`. Often, transformations may need to insert more statements in the body of constructs. To simplify such transformations, the front-end of IMOP ensures that the body of each construct is enclosed in a pair of braces, if none exists already. Note the following example transformation :

```
                                  #pragma omp task
                                  {
#pragma omp task                      while (x < 1000) {
    while (x < 1000)     ⟹                x += y;
        x += y;                       }
                                  }
```

(6) *Splitting combined constructs.* OpenMP provides combined constructs as shortcuts to specify one OpenMP construct within the other, with no other statement in the body of the outer construct. Two famous combined constructs of OpenMP are ParallelForConstruct (used when a ForConstruct is immediately nested inside a ParallelConstruct), and ParallelSectionsConstruct (used when a SectionsConstruct is immediately nested inside a ParallelConstruct).

While combined constructs are handy shortcuts to use for commonly occuring programming patterns, they add to the number of cases that an analysis or transformation may need to handle. Hence, we replace all combined constructs with their corresponding nested equivalents. Below is an example replacement of a `ParallelForConstruct` :

```
#pragma omp parallel for \\                  #pragma omp parallel \\
firstprivate(y) ordered(1)                          firstprivate(x)
    for (x = 0; x < N; x++) {                {
       ...                              ⟹   #pragma omp for ordered(1)
    }                                            for (x = 0; x < N; x++) {
                                                         ...
                                                 }
                                             }
```

Note that when a combined construct contains clauses (e.g., a OmpFirstPrivateClause, such as `firstprivate(y)`) we attach those clauses to one or both of the split constructs, as applicable.

(7) *Making barriers explicit.* In OpenMP, a barrier is defined as that program point where each thread in the team of threads suspends further execution, until all other threads in its team encounter that or some other barrier. Barriers can be present either explicitly as `#pragma omp barrier` (BarrierDirective), or implicitly at the end of various worksharing constructs, like ForConstruct, SectionsConstruct, and SingleConstruct, unless an explicit `nowait` clause (NowaitClause) is applied to these constructs.

Any analysis or transformation that gets affected by presence of barriers, would have to handle less number of cases if all the barriers are present explicitly. Hence, in the front-end, we make all implicit barriers as explicit, by adding `nowait` clause wherever required. Following is an example transformation :

```
                                             #pragma omp single nowait
#pragma omp single                      ⟹   {...}
{...}                                        #pragma omp barrier
```

Note that there also exists an implicit barrier at the end of each ParallelConstruct, which acts as a *join* point for each *forked* thread in the associated team of the construct. However, OpenMP does not provide us with any means to make this barrier explicit. Since this barrier will always invariably be present as the last instruction in a `ParallelConstruct`, no analysis or transformation within the construct is expected to be affected by the implicit nature of this barrier.

(8) *Other minor simplifications.* Following is a list of other minor changes that we perform in the front-end after parsing of the input program :

- *Making return types explicit.* In C, when the signature of a FunctionDefinition does not explicitly specify any return type, then the return type is implicitly considered to be `int`. In cases where an explicit return type is missing, we add an explicit `int` return type to the signature of functions.

  ```
  foo (...) {...}   ⟹   int foo (...) {...}
  ```

- *Handling chained assignments.* Each AssignmentExpression in C is a value, which is same as the value being assigned. Hence, it is possible to have chained assignment expressions of the form $e_1 = e_2 = \cdots = e_n$. Such chained assignments are syntactic sugars, and can unnecessarily complicate various analysis and transformation passes. Hence we replace them with multiple statements and expression with single assignment operation in each.

```
                                        en' = en;
                                        ...
e1 = e2 = e3 = ... en' = en     ⟹      e2 = e3;
                                        e1 = e2
```

- *Removing old-style function signature.* In old-style function prototypes, the paramater list does not contain type information for the parameters. Instead, parameter types are indicated separately as declarations, which immediately precede the body of the function. When a declaration does not exist for a parameter, the type of that parameter is implicitly assumed to be `int`.
  The old-style function signatures are not recommended by the ANSI C standard. Although IMOP does allow usage of old-style function signatures to allow handling of legacy code, yet the front-end implicitly replaces such definitions with their corresponding new style. For example, note the following transformation :

```
int foo(a, b, c)
char *a;                  int foo(char *a, int b, float c) {
float c;          ⟹          ...
{...}                     }
```

2.3 Symbol Tables

An identifier in C may represent a variable (also termed as *object*) or a function name, among other things [5]. With each variable and function, we associate a *symbol* (represented by the `Symbol` class), which contains various important attributes about that variable or function. Following are some key attributes :

(1) *Name.* This attribute is represented by the field `name:String`. As its name suggests, it holds the name of the variable/function represented by this symbol.

(2) *Scope and visibility.* A symbol can be used (i.e., is *visible*) only lexically within a region of program, called as its *scope*. The ISO C99 standard defines four scopes : file, function, block and function prototype scope. In IMOP, scope of a symbol is represented by the value of its attribute `definingScope:Scopeable`. For a file (global) scope, the `definingScope` field would refer to the corresponding TranslationUnit object, for a function scope, it would refer to the corresponding FunctionDefinition object, and for a block scope, it would refer to the corresponding CompoundStatement object. [6] (All these three classes implement the `Scopeable` interface.)

    Scope for a symbol representing a global variable will be of type `TranslationUnit`. For a local variable, the scope would refer to that object of `CompoundStatement`, in which the

[5]Specifically, tags and members of a structure or union, enumeration constants, typedef names, and label names.

[6]By definition, a function prototype scope applies to those variables which are declared in the parameter list of a function declaration (not definition). For example, variables a, b, and c have function prototype scope in the following :

```
int foo (int a, char * b, float c);
```

We do not handle function prototype scopes explicitly, since no symbols associated with them can be used in any executable code (like statements or predicates).

declaration for the variable appears. In case of a parameter of a function, the scope would refer to the `FunctionDefinition` object which represents that function.

Note that nested functions are an extension of GCC, and are not allowed as per the ISO C99 standard. IMOP does not accept nested functions. Hence the scope of a symbol that represents a function is always a `TranslationUnit` object.

(3) *Type.* The type of a symbol defines the meaning of values stored/represented by the symbol. Type information about a symbol is represented by the field `type:Type`. More details about the class `Type` can be found in Section 2.4.

(4) *Storage class.* The storage class of a symbol (representing a variable) is used to indicate the nature of memory allocated for that variable, as well as the lifetime of the allocated memory. In IMOP, we store the storage classes (represented by the enumeration `StorageClass`) of a variable as a part of its type information.

(5) *Defining node.* The attribute `definingNode:Node` represents the AST node which defines (or declares) this symbol. When a symbol represents a function, this attribute refers to the defining FunctionDefinition object, if function definition is present; otherwise, this attribute refers to the associated Declaration which represents the function prototype. When a symbol represents a parameter variable, this attribute refers to the associated ParameterDeclaration object. For local and global variables, this attribute refers to the declaring `Declaration` object.

Each Scopeable object contains an attribute, `symbolTable:HashMap<String, Symbol>`, which contains a symbol table for all the symbols declared in that scope.

Nesting of scopes. In C, scopes can be nested within each other. A scope can have an object with same name as an object of its enclosing scope. In this scenario, an identifier in the inner scope refers to the object in the inner scope, *shadowing* the object with same name from the enclosing scope. Hence, to ensure correct binding of identifiers to their symbols, the symbol tables should be stored in a nested manner, and lookup should be done from inside out, starting with the scope that immediately encloses the use of the identifier. In IMOP, since we associate each symbol table directly to the AST node representing its corresponding scope, a natural nesting of the symbol tables exists, defined by the nesting of the corresponding scopes.

2.4 Types

Types are used to define meaning of values that an object (or variable) may hold, or that a function may return. The ISO C99 standard defines various primitive and user-defined types in detail in Section 6.2.5. The type system of IMOP closely matches that of the ISO C99 standard.

2.4.1 Type hierarchy. Every variable, constant, and expression of the input program has a type. All these types are represented by various subclasses of `Type`. Variables of these types can be declared in various different ways, as specified by the TypeSpecifier component of the corresponding Declaration. Below, we list different C types supported by IMOP, along with the relevant type specifiers (Section 6.7.2.2, ISO C99). (Note that all concrete classes are written in `monospaced fonts`.)

- `VoidType` : represents an empty set of values.
  Type specifier : **void**.
- *ArithmeticType*
  - *IntegerType*
    - `CharType`[7]
      Type specifier : **char**.
    - `EnumType`
      Type specifier : EnumSpecifier.
    - `_BoolType` : stores either a 0 or 1.
      Type specifier : **_Bool**.
    - *SignedIntegerType*
      - `SignedCharType`
        Type specifier : **signed char**.
      - `SignedShortIntType`
        Type specifier : **short**, **signed short**, **short int**, or **signed short int**.
      - `SignedIntType` Type specifier : **int**, **signed**, or **signed int**.
      - `SignedLongIntType` Type specifier : **long**, **signed long**, **long int**, or **signed long int**.
      - `SignedLongLongIntType` Type specifier : **long long**, **signed long long**, **long long int**, or **signed long long int**.
    - *UnsignedIntegerType*
      - `UnsignedCharType` Type specifier : **unsigned char**.
      - `UnsignedShortIntType` Type specifier : **unsigned short**, or **unsigned short int**.
      - `UnsignedIntType` Type specifier : **unsigned**, or **unsigned int**.
      - `UnsignedLongIntType` Type specifier : **unsigned long**, or **unsigned long int**.
      - `UnsignedLongLongIntType`
  - *FloatingType*
    - `FloatType` Type specifier : **float**
    - `DoubleType` Type specifier : **double**
    - `LongDoubleType` Type specifier : **long double**
- *DerivedType*
  - `ArrayType`
    Represents C arrays; attribute `elementType:Type` refers to the type of elements; attribute `size:ConstantExpression` refers to the declared size for the array.
  - `FunctionType`
    Represents the types for C functions; return type is represented by the attribute `returnType:Type`; type information for parameters is stored in the attribute `parameterList:ArrayList<Parameter>`, where each `Parameter` object contains reference to following information : (i) type of the parameter (`parameterType:Type`), (ii) name of the parameter (`parameterName:String`),

[7]Note that `CharType` can be same as either `SignedCharType` or `UnsignedCharType`, depending upon the implementation. However, none of the three types are considered as compatible with one another.

and (iii) reference to the ParameterDeclaration AST node corresponding to the parameter (`parameterDeclaration:ParameterDeclaration`).

– `PointerType`
  Represents C pointers; attribute `pointeeType:Type` is used to indicate the type of the referenced object/function.
– `StructType`
  Represents C structures; name of the structure is denoted by `tag:String`; type information for various members of the structure is represented by the attribute `elementList:ArrayList<StructOrUnionMember>`, where each `StructOrUnionMember` contains following key attributes : (i) type of the member (`elementType:Type`), (ii) name of the member (`elementName:String`), and (iii) size of the bit-field, if any (`bitFieldSize:int`) (Value of this field is -1 for those members which are not a bit-field.).
  Type specifier : StructOrUnionSpecifier.
– `UnionType`
  Represents C unions; name of the union is denoted by `tag:String`; type information for various members of the union is denoted by the attribute `elementList:ArrayList<StructOrUnionMember>`.
  Type specifier : `StructOrUnionSpecifier`.

Given a declaration for any object (variable) or function (e.g., Declaration, FunctionDefinition, etc.), its associated type can be obtained by using any of the overloaded methods in `Type::getTypeTree(*)`.

For any given Expression node in the AST, its type can be obtained by using the method `Type::getType(Expression)`, which may look into the type tables (Section 2.4.4) and apply type conversion rules (Section 2.4.3), wherever applicable.

2.4.2 Further categorization of types. Note that the ISO C99 standard also provides these additional classifications for certain subclasses of `Type` :

- *Basic types* : `CharType`, *SignedIntegerType*, *UnsignedIntegerType*, and *FloatingType*. The method `Type::isBasicType()` returns `true` when called on any object of a basic type.
- *Integer types* : `CharType`, *SignedIntegerType*, *UnsignedIntegerType*, and *EnumType*.
- *Scalar types* : *ArithmeticType* and `PointerType`. The method `Type::isScalarType()` returns `true` when called on any object of a scalar type.
- *Aggregate types* : `ArrayType` and `StructType`. The method `Type::isAggregateType()` returns `true` when called on any object of an aggregate type.
- *Derived declarator types* : `ArrayType`, `FunctionType`, and `PointerType`. The method `Type::isDerivedDeclaratorType()` returns `true` when called on any object of these types.

2.4.3 Type conversions. IMOP follows all the *implicit* type conversion rules stated by the ISO C99 standard, including :

Table 2. Integer conversion ranks for various integer types.

Integer type	**Integer conversion rank**
`SignedLongLongIntType` and `UnsignedLongLongIntType`	6
`SignedLongIntType` and `UnsignedLongIntType`	5
`SignedIntType`, `UnsignedIntType`, and `EnumType`	4
`SignedShortIntType` and `UnsignedShortIntType`	3
`SignedCharType`, `UnsignedCharType`, and `CharType`	2
`SignedLongLongIntType` and `UnsignedLongLongIntType`	1
`_BoolType`	0

- *integer promotions* (Section 6.3.1.1.2, ISO C99 standard), which converts all integer types to `int` or `unsigned int` depending upon the representation size (code at `Type::getIntegerPromotedType()`),
- *usual arithmetic conversions* (Section 6.3.1.8, ISO C99), which finds a common real type for the operands and result of operations on `ArithmeticTypes` (code at `Type.getUsualArithmeticConvertedType(Type, Type)`), and
- other operator-specific rules (Section 6.5, ISO C99).

Note that some rules of the usual arithmetic conversions are defined in terms of *integer conversion ranks*, which provide a rank to each integer type (Section 6.3.1.1.1, ISO C99). An integer type with higher integer conversion rank can hold all the values that an integer type with lower integer conversion rank can. Since no absolute ranks are provided in the ISO C99, we use the ranks specified in Table 2 for performing arithmetic conversions. Note that these absolute values preserve all the constraints specified in the standard.

2.4.4 Type and typedef tables. As discussed in Section 2.4.1, type specifiers are used to declare variables, and to specify type information about functions. Users may create new type specifiers either by creating new user-defined types, like sturctures, unions, or enumerations, or be creating `typedefs`, which are generally used to give names to complicated type trees. Definitions for typedefs and types can be specified in any scope, like TranslationUnit, CompoundStatement or FunctionDefinition. For each scope, information about user-defined types and typedefs, defined in that scope, is stored in the attributes `typeTable:HashMap<String, Type>` and `typedefTable:HashMap<String, Typedef>`. This information is generally used in obtaining type for any given expression or function, as well as while creating new temporaries/functions.

2.5 Memory abstractions

Stack and *heap* are two main components of the data memory that are generally modelled during semantic analysis of a program. We term each abstract element of the memory as a *cell* (represented by class `Cell`). Note that a compile-time abstract element may refer to one or more elements of the memory during runtime. For example, in case of field-insensitive analyses, a variable of a structure type may refer to all its members, and a variable of an array type may refer to all the elements of the array.

In IMOP, we model a total of four types of cells, represented by following four subclasses of `Cell`:

(1) **Symbol** A `Symbol` object may represent a variable or a function (Section 2.3). We assume that each variable corresponds to a memory cell in stack (in case of local variables), and in global memory (in case of global variables). Similarly, since it is possible to use function designators in expressions (consider function pointers), we need to model a memory cell (in global memory) for each symbol that represents a function. Since there exists a one-to-one mapping between a symbol and its associated memory cell, we overload each `Symbol` object to also act as an abstraction for memory cell on stack (or global memory).

   Various key attributes of a `Symbol` object have been discussed in Section 2.3. We also maintain a static attribute `genericCell:Symbol` for the `Symbol` class, which represents an abstraction of all the memory cells in the stack (, global memory, and heap). This abstraction is useful while representing the universal set of stack cells.[8]

(2) **HeapCell** As its name implies, a `HeapCell` object is used to model a memory cell in the heap. In IMOP, by default, we associate a unique heap cell with each syntactic occurrence of a call to following procedures – `, malloc, calloc, and realloc`. Various attributes of a `HeapCell` object are :

   (a) `allocatorType:Allocator` represents the type of method which was used to create this heap element in the program. The possible types are represented by an enumerator `Allocator`, which can take following values – `MALLOC`, `CALLOC`, `REALLOC`, and `NONE`. Note that value `NONE` is used to model those heap cells for which the allocation statement is not known, e.g., heap cells pointed to by parameters of a function, in case of intra-procedural analyses.[9]

   (b) `allocatorNode:CallStatement` refers to the AST node of type `CallStatement` (Section 2.2) that represents the call-site where this cell was allotted on the heap.

   (c) `heapId:int` refers to a unique integral ID, which is used to identify heap cells.

   Furthermore, similar to class `Symbol`, the `HeapCell` class also contains a static field `genericCell`, which is used to represent all the memory cells on heap (and stack).

(3) **FreeVariable** During program transformations, there may exist transient states of programs or snippets which may be incomplete, and hence semantically undefined. When an identifier in an expression does not have a corresponding declaration in the enclosing scopes, then instead of a `Symbol`, the memory cell corresponding to the identifier is represented by an object of `FreeVariable`. For example, upon creation of a new expression, the variables used within the expression do not have any corresponding declarations, unless the newly created expression is attached within a proper scope. In such cases, for each identifier, we maintain a `FreeVariable` object.

   Each `FreeVariable` object has a key attribute `nodeToken:NodeToken`[10], which represents the AST node representing the identifier.

[8]Since we represent the universal set of certain elements by a specific element, we need to modify the associated set operations accordingly.

[9]For intra-procedural analyses, we assume that all parameters of pointer types, point to the `genericCell` element.

[10]Each token in the grammar is represented by `NodeToken` objects. We parse each identifier as a token in the input grammar.

Note that for any identifier in an expression, a `FreeVariable` object acts as a placeholder for the actual `Symbol` object which would be known only when the expression is enclosed in a scope with required declarations.

(4) **AddressCell** In certain analyses, we may need to model the address of variables. Each address is an unknown constant at compile time. We represent the address of a variable as its read-only attribute. This attribute is modelled by the field `addressCell:AddressCell` of the `Symbol` object corresponding to the variable. Note that the variable can be considered as a pointer to its associated `addressCell`.

An expression may read and/or write to multiple memory cells. Given an expression, we can obtain the list (or set) of cells that may be read or written in the expression by calling `NodeInfo::getReads()` and `NodeInfo::getWrites()` methods on the information object corresponding to the AST node representing the expression.

2.6 Control-flow graphs

A control-flow graph (CFG) is used to model the flow of control among executable AST nodes. The control flow in a program is dictated by the semantics of various syntactic constructs (like CompoundStatement, which specifies a sequential flow of control, WhileStatement, which specifies a looping flow of control, TaskConstuct, which specifies a possibly parallel flow of control, etc.) and jump statements (like GotoStatement, ReturnStatement, etc.).

2.6.1 Nested CFG. In general, various syntactics constructs are used nested within one another to express the desired control flow. In most of the low-level representations of a program, these higher-level syntactic constructs get replaced by various conditional and unconditional branches (jumps) and labels. Such representations are non-nested in nature. Hence, most of the frameworks that work at low-level representations (e.g., three-address code form) of a program, use non-nested flat CFG structure. However, in case of IMOP, since we work close to the source-code level, where the control flow is expressed with the help of nested syntactic constructs, it is natural for us to use a nested CFG structure. Furthermore, specially in the context of OpenMP programs, where a programmer specifies OpenMP constructs around blocks of code (e.g., specifying body of a ParallelConstruct, SingleConstruct, etc.), nested structure of the CFG can help us in easily representing the facts about which group of leaf nodes specify a parallel region, which group of leaf nodes need to be executed by a single thread, etc.

In order to ease the task of ensuring consistency between the AST and CFG abstractions of a program during transformations, we simply treat some of the AST nodes as CFG nodes, instead of creating new nodes for representing the CFG. Our nested CFG is comprised up of two set of nodes – non-leaf CFG nodes, and leaf CFG nodes. Non-leaf CFG nodes are those AST nodes which contain other (leaf or non-leaf) CFG nodes within them.

The execution of a non-leaf CFG node starts at its specific constituent `begin:BeginNode`, executes various constituents of the node in an order dictated by the semantics of the node, and finally finishes at its another specific constituent `end:EndNode`. However, there may be multiple entry and exit points for a non-leaf node, apart from its `begin` node and `end` node, respectively, owing to various JumpStatements and labels. In fact, the end node may or may not be reachable from the `begin` node.

Following is a list of all types of leaf CFG nodes in IMOP : Declaration[11], ParameterDeclaration, UnknownCpp, UnknownPragma, OmpForInitExpression, OmpForCondition, OmpForReinitExpression, FlushDirective, DummyFlushDirective, BarrierDirective, TaskwaitDirective, TaskyieldDirective, ExpressionStatement, GotoStatement, ContinueStatement, BreakStatement, ReturnStatement, Expression[12], IfClause, NumThreadsClause, FinalClause, BeginNode, EndNode, PreCallNode, and PostCallNode.

Following is a list of all types of non-leaf CFG nodes in IMOP : FunctionDefinition, ParallelConstruct, ForConstruct, SectionsConstruct, SingleConstruct, TaskConstruct, MasterConstruct, CriticalConstruct, AtomicConstruct, OrderedConstruct, CompoundStatement, IfStatement, SwitchStatement, WhileStatement, DoStatement, ForStatement, and CallStatement.

The flow of control among the components of these non-leaf nodes is specified as per the semantics of the non-leaf node. For example, the control-flow in a `WhileStatement` starts with its `begin` node, goes to its constituent `Expression` predicate, from where it either terminates after jumping to the end node, or goes into the constituent `CompoundStatement` body, from where it comes back again to the `Expression` predicate iff the control can reach the end node the body. The constituents and control-flow for other types of non-leaf CFG nodes are defined in a similar way, as per the standard. However, since `CallStatement` (Section 2.2) is not a standard statement in C language, we explicitly specify the control flow within a `CallStatement` in Section 2.6.2.

Note that the successors of a node are specified by the attribute `succBlocks:ArrayList<Node>`, and predecessors of a node are specified by the attribute `predBlocks:ArrayList<Node>`. A `begin` node will have no predecessors, and an end node will have no successors. When control flows into a non-leaf node, it is assumed to flow into the `begin` node of the non-leaf node. Similarly, when the control flows out of an end node of a non-leaf node, it is assumed to flow out of the non-leaf node.

2.6.2 Control flow in a `CallStatement`. We introduced the notion of a `CallStatement` in Section 2.2. A `CallStatement` represents a call-site, in simplified form – each argument is either an identifier or a constant; the function designator is an identifier; and the return value, if any, is assigned to a simple scalar variable. Each `CallStatement` non-leaf CFG node comprises of following four leaf CFG nodes :

(1) `begin:BeginNode` is the first node to get executed within a `CallStatement`.
(2) `preCallNode:PreCallNode` represents reads of all the arguments.
(3) `postCallNode:PostCallNode`, optionally, emulates the write of the return value to a scalar temporary.
(4) `end:EndNode` is the last node to get executed before the control exist the `CallStatement`.

[11]In general, a `Declaration` is not considered as a CFG node. However, since we can have **const** qualified declarations with initializers in the input program, for now we consider an implicit or explicit write at all the `Declaration` nodes.

[12]All expressions that represent predicates (e.g., branch expression of an `IfStatement`) will be replaced by type `Predicate` in the release version.

Note that the *intra-procedural* control-flow for this node can be represented simply as `begin` → `preCallNode` → `postCallNode` → `end`. In case of intra-procedural analyses, the analyses do not (or can not) look into the body of the called function. Hence, the analyses need to model the side-effect of the function call in global and heap memory. We assume that these effects take place in the `preCallNode`.

In case of *inter-procedural* analysis, if the body (or bodies, when the function designator may refer to more than one functions statically of the called function is available, the control flows from `preCallNode` to the `FunctionDefinition` of the target function. After executing the target function, the control returns back from the `FunctionDefinition` to `postCallNode`. Note that since we manage call-stacks (Section 2.7), we traverse on only *valid paths* in IMOP, i.e., when the control returns back from a function, it returns back to that program point from where the function was called. (Note that if the traversal did not start from the `main()` function, or if the call-stack is empty, then the traversal from `end` node of a function can continue to all the possible call-sites for which that function may be a target.)

In IMOP, we support both intra- and inter-procedural (both context-sensitive, as well as -insensitive) valid traversals of the CFG.

2.6.3 CFG traversals. Although the nested nature of the CFG may help in ensuring its close proximity to the AST, and thereby ease various transformations, yet many analyses might want to traverse only from one leaf to another, bypassing the wrappings of all the non-leaf nodes. Similarly, while traversing the CFG of a parallel region, an analysis might not want to traverse inside the nested parallel regions, if any. To handle a variety of such cases, we provide various types of CFG traversal mechanisms. In this section, we look at some of those traversals. Note that all control-flow related information for a node is stored in the corresponding `CFGInfo` object which is accessed by calling `getInfo().getCFGInfo()` method on the node. Following traversal methods are specified for a CFG node using its `CFGInfo` objects :

- `getSuccBlocks()` and `getPredBlocks()`
  - These methods are used to obtain intra-procedural intra-task successors and predecessors at the same nesting level of a CFG node.
  - For an `EndNode`, `getSuccBlocks()` always returns an empty list.
  - For a `BeginNode`, `getPredBlocks()` always returns an empty list.
  - `CallStatement` are treated as any other leaf node, i.e., these methods provide only intra-procedural traversals.
  - `DummyFlushDirective` nodes too are treated as an empty node, i.e., these methods do not provide intra-task traversals (Section 2.11).
  - These methods read from the attributes `succBlocks:ArrayList<Node>` and `predBlocks:ArrayList<Node>` of the `CFGInfo` object, respectively,
- `getSuccessors()`, and `getPredecessors()`
  - The list returned by `getSuccessors()` is same as that returned by `getSuccBlocks()`, except for a subtle difference : when `getSuccessors()` is called on an `EndNode`, the successors of the enclosing non-leaf node are returned. These semantics are useful while traversing the control-flow graph in forward direction across the nesting boundaries.

  - The list returned by `getPredecessors()` is same as that returned by `getPredBlocks()`, except for a subtle difference : when `getPredecessors()` is called on a `BeginNode`, the predecessors of the enclosing non-leaf node are returned. These semantics are useful while traversing the control-flow graph in backward direction across the nesting boundaries.
  - For an `EndNode` of a `FunctionDefinition`, an empty list is returned by `getSuccessors()`. Similarly, an empty list is returned by `getPredecessors()` when called on the `BeginNode` of a `FunctionDefinition`. This implies that the traversals do not move across the function boundaries, and are intra-procedural in nature.
  - Like `getSuccBlocks()` and `getPredBlocks()`, these methods do not traverse across inter-procedural or inter-task edges from `CallStatement` and `DummyFlushDirective` nodes.
- `getLeafSuccessors()` and `getLeafPredecessors()`
  - These methods are used to emulate the behavior of a non-nested flat CFG. None of the non-leaf nodes are ever returned as an element by these traversal methods. These traversal methods bypass the nesting of the CFG and help in working directly at the leaf nodes.
  - The list returned by `getLeafSuccessors()` and `getLeafPredecessors()` is same as that returned by `getSuccessors()` and `getPredecessors()` with all the non-leaf nodes replaced by their corresponding `BeginNode` and `EndNode` nodes, respectively.
  - Note that the traversal using these methods remain intra-procedural and intra-task in nature.
- `getInterProceduralLeafSuccessors()` and `getInterProceduralLeafPredecessors()`
  - These methods are used to obtain context-insensitive inter-procedural intra-task leaf successors and predecessors of a CFG node.
  - Given the `EndNode` of a `FunctionDefinition`, the returned list of the method `getInterProceduralLeafSuccessors()` contains the `PostCallNode` of all those call-sites (`CallStatement`) whose target can be that function.
  - Given the `BeginNode` of a `FunctionDefinition`, the returned list of the method `getInterProceduralLeafPredecessors()` contains the `PreCallNode` of all those call-sites (`CallStatement`) whose target can be that function.
  - For the `PreCallNode` of any `CallStatement`, the returned list of the method `getInterProceduralLeafSuccessors()` contains the `BeginNode` of all those `FunctionDefinition` which can be a possible target of that `CallStatement`.
  - For the `PostCallNode` of any `CallStatement`, the returned list of the method `getInterProceduralLeafPredecessors()` contains the `EndNode` of all those `FunctionDefinition` which can be a possible target of that `CallStatement`.
  - In all other cases, the return of `getInterProceduralLeafSuccessors()` and `getInterProceduralLeafPredecessors()` will be same as the return of `getLeafSuccessors()` and `getLeafPredecessors()`, respectively.
  - Note that these methods do not traverse across the inter-task edges of `DummyFlushDirective` nodes.

Apart from the traversal methods described above, we also have a couple of collector methods – `getLexicalIntraProceduralContents()` `getLexicalIntraProceduralLeafContents()` that are used to obtain the set of all the intra-procedural CFG and leaf CFG nodes, contained lexically within the receiver object, respectively.

2.6.4 CFG visualizer. For debugging or visualization purposes, the intra-procedural nested control-flow graph for any given function or snippet can be visualized using the method `Misc.generateDotGraph(Node, String)`. This method dumps the nested control-flow graph of given `Node` in DOT format, in a file with the specified name. This generated file can be viewed graphically by any dot graph visualizers.

2.7 Call graphs

A call graph represents the flow of control among various functions of a program. In IMOP, we emulate the edges of a call graph as follows :

- In order to obtain the set of call-sites within a `FunctionDefinition` (or any snippet), we use the method `getCallStatements():ArrayList<CallStatement>` on the corresponding information object of the AST/CFG node.
- To obtain the set of `FunctionDefinition` nodes which can be a possible target of a `CallStatement`, we use the method `getCalleeDefinitions()` on the information object of the `CallStatement` (of type `CallStatementInfo`).
- And finally, for backward traversals on the call graph, given a `FunctionDefinition`, we can obtain the set of all `CallStatement` nodes whose target can be the given function, by using the method `getCallersOfThis()` on the information object of the function (of type `FunctionDefinitionInfo`).

Note that unless we maintain a call stack, it would be difficult to know which call-statement had called a given function, since we would not know the calling context. Hence, in next section, we discuss the notion of call stacks.

2.7.1 Call stack. A CallStack object represents the notion of context-sensitive/-insensitive call stack. It is a stack of `CallStatement` nodes. While performing any context-sensitive static analysis, the call-stack is used to remember the calling context.

Emulating a context-insensitive call-stack. In recursive programs, sometimes we might need to use context-insensitive call-stack, to limit the bound on the size of the call stack statically. To represent a context-insensitive stack, we use a special object: `CallStatement.getPhantomCall()`. In such a scenario, the stack will comprise of only this object and nothing else. No attempts to push or pop will have any affect on the stack. Peek (and pop) operations would return a reference to the phantom call statement. If attempts are made to pop from an empty call-stack, we instead push a phantom call statement, and return it.

Handling recursion. While modelling the call stack in case of recursive programs, we need to put a bound on the size of the stack to ensure termination of various traversals and analyses. In case of a context-sensitive stack, if attempts are made to push a call statement that is already present in the stack, we make the stack context-insensitive by clearing the stack and pushing the phantom call statement on stack.[13]

2.7.2 Context-sensitive traversals. To complement the list of context-insensitive traversal methods on the CFG, present in Section 2.6, we discuss below a pair of traversal methods that are used for context-sensitive inter-procedural traversals, using `CallStacks` – `getInterProceduralLeafSuccessors(CallStack CS)` and `getInterProceduralLeafPredecessors(CallStack CS)`. Note that these methods return a list of pairs of `Node` and `CallStack`.

- These methods are used to obtain *context-sensitive* inter-procedural intra-task leaf successors and predecessors of a CFG node, along with the relevant calling contexts.
- Given the `EndNode` of a `FunctionDefinition`, and a context-insensitive call stack (with phantom call statement at its top), say `CS`, the list returned by the method `getInterProceduralLeafSuccessors(CS)` would contain the `PostCallNode` of all those `CallStatement` nodes whose target can be that function. The state of the stack for each returned `PostCallNode` would remain unchanged.

  In case of context-sensitive stack, the returned list would contain only one element – `PostCallNode` of the popped `CallStatement` from the stack, with a modified state of the stack – one with the top element popped from `CS`.
- Given the `BeginNode` of a `FunctionDefinition`, and a context-insensitive call stack (with phantom call statement at its top), say `CS`, the list returned by the method `getInterProceduralLeafPredecessors(CS)` would contain the `PreCallNode` of all those `CallStatement` nodes whose target can be that function. The state of the stack for each returned `PreCallNode` would remain unchanged.

  In case of context-sensitive stack, the returned list would contain only one element – `PreCallNode` of the popped `CallStatement` from the stack, with a modified state of the stack – one with the top element popped from `CS`.
- When `getInterProceduralLeafSuccessors(CS)` is called on the `PreCallNode` of a `CallStatement`, with a context-insensitive stack `CS` (i.e., `CS` contains a phantom call statement as its top element), the returned list contains the `BeginNode` of all the `FunctionDefinition` nodes which can be target of that call statement. The state of the stack remains unchanged in the returned list.

  In case of context-sensitive stacks, the returned list would contain one element each for `BeginNode` of all the possible target `FunctionDefinition` nodes with a modified state of the stack – one with the `CallStatement` pushed on top of `CS`. Note that as per the current semantics, if `CS` already contains this `CallStatement`, then the stack would be made context-insensitive by emptying it and pushing a phantom call statement.
- When `getInterProceduralLeafPredecessors(CS)` is called on the `PostCallNode` of a `CallStatement`, with a context-insensitive stack `CS` (i.e., `CS` contains a phantom call statement as its top element), the returned list contains the `EndNode` of all the

[13]This policy can be changed as per the requirements, by making appropriate changes in the class `CallStack`.

FunctionDefinition nodes which can be target of that call statement. The state of the stack remains unchanged in the returned list.

In case of context-sensitive stacks, the returned list would contain one element each for EndNode of all the possible target FunctionDefinition nodes with a modified state of the stack – one with the CallStatement pushed on top of CS. Note that as per the current semantics, if CS already contains this CallStatement, then the stack would be made context-insensitive by emptying it and pushing a phantom call statement.

- In all other circumstances, the returned lists for methods getInterProceduralLeafSuccessors(CS) and getInterProceduralLeafPredecessors(CS) would be same as the ones returned by getInterProceduralLeafSuccessors() and getInterProceduralLeafPredecessors(), respectively, with the stack for each element of the returned lists as CS.
- Note that these methods do not traverse the inter-task edges of DummyFlushDirective nodes.

2.8 Snippet builders

In order to create a new AST node, we have following two options in IMOP :

(1) Use the constructors provided with various classes that represent the AST nodes. There exists one class each for all the non-terminals of the grammar (Appendix C) at the package imop.ast.node.external. Each terminal (token) is represented by objects of class NodeToken.

Note that this way of construction of the AST puts the onus of creating a correct parse tree for the input program or snippet on the programmer. We do not recommend this approach since it is complex, time-consuming, and error-prone. For the example snippet below,

```
if (x > 0) {
   x = x + 1;
}
```

the code (approximated for simplicity) that we need to write to parse it as an IfStatement is :

```
Identifier variablePredicate = new Identifier("x");
Constant constPredicate = new Constant("1");
RelationalLTExpression relationalExpression = new RelationalLTExpression(variablePredicate,
                                                  constant);
Expression predicate = new Expression(relationalExpression);
...
// Generate "lhs" and "rhs" as expressions.
...
AssignmentStatement assignStmt = new AssignmentStatement(lhs, rhs);
...
// Generate "body" as a block from "assignStmt".
...
IfStatement newStmt = new IfStatement(predicate, body);
```

(2) IMOP provides various methods in class `FrontEnd`, which take the input program or snippet in some form of stream, (including a `String`!), and invoke the in-built parser internally to create the complete AST tree rooted at a node of specified form.[14] These methods either throw a `ParseException` if the specified stream (or `String`), can not be parsed as the specified AST node type, or return a reference to the root node of the fully constructed AST. For example, in order to parse the snippet `IfStatement` above, we need to write a single line of code, as follows :

```
IfStatement newStmt = FrontEnd.parseAlone("if(x>0){x=x+1;}", IfStatement.class);
```

At the end of this statement, node `newStmt` would refer to the root node of type `IfStatement`, of the newly created AST. Note that the input `String` for parsing methods need not be compile-time constants, and can be generated on-the-fly at runtime. (Nothing special here. It's Java.)

2.9 Elementary transformations

In this section, we look into the list of elementary transformations that : (i) are used to update various components of non-leaf CFG nodes, (ii) help us in modifying labels associated with various statements, or (iii) update the symbol table entries for scopes. Almost all higher-level transformations can be expressed in terms of these elementary transformations. (Note that an exhaustive list of all the higher-level transformations provided by IMOP, will be present in a future draft of this report.)

Below, we discuss the list of elementary transformations currently provided by IMOP, corresponding to various non-leaf CFG nodes. (Note that for each setter (or modifier) method below, there exists a corresponding getter method as well, which we do not show to save space.)

- FunctionDefinition
  - `setParameterDeclarationList(ArrayList<ParameterDeclaration>)`
  - `clearParameterDeclarationList()`
  - `removeParameterDeclaration(ParameterDeclaration)`
  - `addParameterDeclaration(ParameterDeclaration)`
  - `setBody(Statement)`
- ParallelConstruct
  - `setIfClause(IfClause)`
  - `removeIfClause()`
  - `setNumThreadsClause(NumThreadsClause)`
  - `removeNumThreadsClause()`
  - `setBody(Statement)`
- ForConstruct
  - `setInitExpression(OmpForInitExpression)`
  - `setForConditionExpression(OmpForCondition)`
  - `setReinitExpression(OmpForReinitExpression)`

[14] Many compiler frameworks can invoke the internal parser only when the input stream represents the whole program (of form `TranslationUnit`, in our case). Unlike them, the parsers created by JavaCC/JTB can invoke the internal parsers even for an input stream that may represent only a snippet of the program (e.g., for an `IfStatement`).

  - `setBody(Statement)`
- SectionsConstruct
  - `setSectionsList(ArrayList<Statement>)`
  - `removeSection()`
  - `addSection(Statement)`
  - `clearSectionsList()`
- SingleConstruct
  - `setBody(Statement)`
- TaskConstruct
  - `setBody(Statement)`
  - `setIfClause(IfClause)`
  - `removeIfClause()`
  - `setFinalClause(FinalClause)`
  - `removeFinalClause()`
- MasterConstruct
  - `setBody(Statement)`
- CriticalConstruct
  - `setBody(Statement)`
- AtomicConstruct
  - `setExpressionStatement(ExpressionStateemnt)`
- OrderedConstruct
  - `setBody(Statement)`
- CompoundStatement
  - `setElementList(ArrayList<Statement>)`
  - `clearElementList()`
  - `removeElement(Statement)`
  - `addElement(Statement)`
- IfStatement
  - `setPredicate(Expression)`
  - `setThenBody(Statement)`
  - `setElseBody(Statement)`
  - `removeElseBody(Statement)`
- SwitchStatement
  - `setPredicate(Expression)`
  - `setBody(Statement)`
- WhileStatement
  - `setPredicate(Statement)`
  - `setBody(Statement)`
- DoStatement
  - `setPredicate(Expression)`
  - `setBody(Statement)`
- ForStatement
  - `setInitExpression()`
  - `removeInitExpression()`
  - `setTerminationExpression()`

– `removeTerminationExpression()`
– `setStepExpression()`
– `removeStepExpression()`
– `setBody()`

Note that we disallow any updates to the fields of a `CallStatement`, since any update to an existing `CallStatement` node may affect the program drastically, and can be difficult to handle. If any updates are needed in the `CallStatement` node, we instead create a new node with modified values. IMOP provides following methods to handle labels associated with a statement – `removeLabelAnnotation()` and `addLabelAnnotation()`.

To add and remove declarations from a `Scopeable` object, following two methods can be used – `addDeclaration()` and `removeDeclaration()`.

Using the methods above, various higher-level transformations can be easily specified. For example, to perform loop unrolling on a `WhileStatement`, say referred by a `whileStmt`, the code that use internally is as follows :

```
String newBodyString = "{";
newBodyString += whileStmt.getBody() + "if (!" + whileStmt.getPredicate()
                    + ") break;" + whileStmt.getBody();
newBodyString += "}";
Statement newBody = FrontEnd.parseAlone(newBodyString, Statement.class);
whileStmt.setBody(newBody);
```

2.10 Concurrency analysis

When an SPMD process is launched, various threads start executing a copy of the SPMD code (code written in an OpenMP parallel region) in parallel. Each thread runs in parallel without any interruption until it encounters a barrier, where it waits for other threads to encounter this or some other barrier. (Barriers in OpenMP are textually unaligned and unnamed). All the statements that get executed by different threads since the beginning of the program till the threads hit a set of barriers, constitute a *runtime phase*. Similarly, next runtime phase will start from this set of barriers and end at the next reachable set of barriers. We term such sets of barriers that synchronize with each other as *synchronization-set*, for context-insensitive analysis. (It is important to note that for context-sensitive analyses, the definition of *synchronization-set* takes the calling contexts as well into account.) We abstract various *runtime phases* sharing the same *synchronization-set* by a single *abstract-phase*, or a *phase* in general. Since the number of syntactic barriers in a program is finite, the number of *synchronization-sets* is also finite, and thus the number of *phases* in a program is finite (for context-insensitive definition of synchronization-sets).

Given a pair of program statements, *May Happen in Parallel* (MHP) analysis tells whether these statements may get executed in parallel by different threads in any of the possible executions of the program. Such an analysis inspects the various constraints imposed on the execution orderings of different statements by the help of various synchronization primitives and other means of communication among threads.

In the base technique that we employ for performing MHP analysis, we annotate each statement with a set of *phase numbers* (called as *phase-set*). A phase number uniquely identifies an abstraction of some runtime phases, where a runtime phase is the set of statements that are executed concurrently by different threads. Note that in the presence of loops and/or procedures, a single statement may become a part of more than one *phase*.

Context-sensitive MHP analysis is undecidable in nature [16]. To tackle this problem, we propose a k-depth context-sensitive version of MHP, where the static analysis maintains the call-stack of only a maximum size k. For all the calls that exceed the call-stack length of k, the analysis approximates the calls in a context-insensitive manner.

Given an MHP query for a pair of statements, the MHP analyzer returns *yes* if the phase-sets of these statements have a non-empty intersection.

In IMOP, we use class `Phase` to model the notion of an abstract phase. To know whether two given statements may run in parallel with each other, we use the method `Misc.mayHappenInParallel(Node, Node)`. (More details about phase information, MHP analysis, data races, atomicity, and lock-set analysis will be added later in this section.)

2.11 Inter-task communication

Data-flow facts in a serial program generally propagate along the edges of the program's control-flow graph (CFG). In case of parallel programs, however, data-flow facts at a statement may get affected even by those statements which do not precede this statement in the program's serial CFG. Seemingly unrelated statements that might be getting executed in parallel by other threads can affect the data-flow facts, in the context of shared memory. To ensure the correctness of various static analyses performed on a shared memory parallel program, it is necessary to employ concurrency analysis to ensure that all the statements that may affect the data-flow facts are taken into account while modelling flow of data.

According to the OpenMP API, communication among a pair of tasks, say (τ_1 and τ_2), may happen only when these four steps happen in order:

(1) τ_1 writes to a shared location, say s.
(2) τ_1 flushes s, implicitly or explicitly.
(3) τ_2 flushes s, implicitly or explicitly.
(4) τ_2 reads from s.

We represent this communication with the help of edges among `DummyFlushDirective` nodes. Note that these edges need to be created only between those `DummyFlushDirective` nodes which share at least one common phase. With each node of type `DummyFlushDirective`, we maintain the following two key attributes :

- `readCells:HashSet<Cell>` refers to the list of all those shared memory cells that may get read from after this dummy-flush node, on dummy-flush free paths.
- `writeCells:HashSet<Cell>` refers to the list of all those shared memory cells that might have been written to before this dummy-flush node, on dummy-flush free paths.

Now, given two `DummyFlushDirective` nodes, say d_1 and d_2, that may share a phase, we generate a directed inter-task edge from d_1 and d_2, if the intersection of d_1.`writeCells` with

d_2.`readCells` is non-empty. The intersection itself is annotated on the edge, and represents the shared locations through which communication may happen between two threads executing the tasks corresponding to d_1 and d_2.

3 RELATED WORK

Apart from IMOP, there are various other existing frameworks like GCC [17], LLVM [7], Cetus [6] and ROSE [15] that provide tools to implement compiler analyses and transformations. Following are some of the fundamental differences (and similarities) between IMOP and other frameworks :

- LLVM and GCC do not work at the source code level. If a certain analysis/transformation is defined (or should be defined) at the source code level, IMOP, ROSE and Cetus are better candidates to pick from.
- GCC, ROSE, and LLVM are written in C/C++. IMOP and Cetus are written in Java, which makes them portable and easier to use.
  Debugging and programmability are some of the various known advantages of Java over C/C++. Although programs written in C/C++ can be more efficient than Java (which is not the case always), yet the latter serves our purpose well. Our priority is to enable easier and faster development of the compiler optimization tools.
- Cetus is written in Java. However, there are various fundamental design differences between IMOP and Cetus. As compared to Cetus, IMOP has better
  - handling of OpenMP constructs and directives,
  - approach to the creation of AST nodes,
  - representation of the CFG nodes,
  - design for AST/CFG traversals,
  - implementation of the iterative data-flow analyses (both serial and parallel),
  - paradigm for implementing iterative transformations of the input programs, etc.
- ROSE is written in C/C++; IMOP is written in Java. ROSE supports OpenMP 3.0 specification; IMOP supports OpenMP 4.0. In IMOP, CFG nodes are more closely coupled with their AST counterparts. This coupling eases the AST transformations which are generally based on the output of the CFG analyses. (As an example, unlike ROSE, IMOP never needs comma operators to add new instructions to the CFG nodes.) With the help of JTB, IMOP encodes the AST traversals in a more modular way than ROSE.
- Along with the utilities provided by the other frameworks, IMOP also provides elementary analyses and transformations that are specific to OpenMP C programs.

A QUICK START GUIDE

A.1 Front-End : Parsing, and Default Passes

A.2 Implementing analyses and transformations

A.3 Emitting the transformed code

B IMPLEMENTATION DEFINED BEHAVIOR

C PARSER GRAMMAR

EBNF for C with OpenMP

TranslationUnit	::=	(ElementsOfTranslation)+
ElementsOfTranslation	::=	ExternalDeclaration
	|	UnknownCpp
	|	UnknownPragma
ExternalDeclaration	::=	Declaration
	|	FunctionDefinition
	|	DeclareReductionDirective
	|	ThreadPrivateDirective
FunctionDefinition	::=	(DeclarationSpecifiers)? Declarator (DeclarationList)? CompoundStatement
Declaration	::=	DeclarationSpecifiers (InitDeclaratorList)? ";"
DeclarationList	::=	(Declaration)+
DeclarationSpecifiers	::=	(ADeclarationSpecifier)+
ADeclarationSpecifier	::=	StorageClassSpecifier
	|	TypeSpecifier
	|	TypeQualifier
StorageClassSpecifier	::=	<AUTO> | <REGISTER> | <STATIC> | <EXTERN> | <TYPEDEF>
TypeSpecifier	::=	<VOID> | <CHAR> | <SHORT> | <INT> | <LONG> | <FLOAT> | <DOUBLE> | <SIGNED> | <UNSIGNED> | StructOrUnionSpecifier | EnumSpecifier | TypedefName
TypeQualifier	::=	<RESTRICT> | <CONST> | <VOLATILE> | <INLINE> | <CCONST> | <CINLINED> | <CINLINED2> | <CSIGNED> | <CSIGNED2>
StructOrUnionSpecifier	::=	(StructOrUnionSpecifierWithList | StructOrUnionSpecifierWithId)
StructOrUnionSpecifierWithList	::=	StructOrUnion (<IDENTIFIER>)? "{" StructDeclarationList "}"
StructOrUnionSpecifierWithId	::=	StructOrUnion <IDENTIFIER>
StructOrUnion	::=	<STRUCT> | <UNION>
StructDeclarationList	::=	(StructDeclaration)+
InitDeclaratorList	::=	InitDeclarator ("," InitDeclarator)*
InitDeclarator	::=	Declarator ("=" Initializer)?
StructDeclaration	::=	SpecifierQualifierList StructDeclaratorList ";"
SpecifierQualifierList	::=	(ASpecifierQualifier)+
ASpecifierQualifier	::=	TypeSpecifier
	|	TypeQualifier
StructDeclaratorList	::=	StructDeclarator ("," StructDeclarator)*
StructDeclarator	::=	StructDeclaratorWithDeclarator
	|	StructDeclaratorWithBitField
StructDeclaratorWithDeclarator	::=	Declarator (":" ConstantExpression)?

```
    StructDeclaratorWithBitField  ::=  ":" ConstantExpression
                  EnumSpecifier  ::=  EnumSpecifierWithList
                                  |   EnumSpecifierWithId
          EnumSpecifierWithList  ::=  <ENUM> ( <IDENTIFIER> )? "{" EnumeratorList "}"
            EnumSpecifierWithId  ::=  <ENUM> <IDENTIFIER>
                 EnumeratorList  ::=  Enumerator ( "," Enumerator )*
                     Enumerator  ::=  <IDENTIFIER> ( "=" ConstantExpression )?
                     Declarator  ::=  ( Pointer )? DirectDeclarator
               DirectDeclarator  ::=  IdentifierOrDeclarator DeclaratorOpList
               DeclaratorOpList  ::=  ( ADeclaratorOp )*
                  ADeclaratorOp  ::=  DimensionSize
                                  |   ParameterTypeListClosed
                                  |   OldParameterListClosed
                  DimensionSize  ::=  "[" ( ConstantExpression )? "]"
        ParameterTypeListClosed  ::=  "(" ( ParameterTypeList )? ")"
         OldParameterListClosed  ::=  "(" ( OldParameterList )? ")"
         IdentifierOrDeclarator  ::=  <IDENTIFIER>
                                  |   "(" Declarator ")"
                        Pointer  ::=  ( "*" | "^" ) ( TypeQualifierList )? ( Pointer )?
              TypeQualifierList  ::=  ( TypeQualifier )+
              ParameterTypeList  ::=  ParameterList ( "," "..." )?
                  ParameterList  ::=  ParameterDeclaration ( "," ParameterDeclaration )*
           ParameterDeclaration  ::=  DeclarationSpecifiers ParameterAbstraction
           ParameterAbstraction  ::=  Declarator
                                  |   AbstractOptionalDeclarator
     AbstractOptionalDeclarator  ::=  ( AbstractDeclarator )?
               OldParameterList  ::=  <IDENTIFIER> ( "," <IDENTIFIER> )*
                    Initializer  ::=  AssignmentExpression
                                  |   ArrayInitializer
               ArrayInitializer  ::=  "{" InitializerList ( "," )? "}"
                InitializerList  ::=  Initializer ( "," Initializer )*
                       TypeName  ::=  SpecifierQualifierList ( AbstractDeclarator )?
             AbstractDeclarator  ::=  AbstractDeclaratorWithPointer
                                  |   DirectAbstractDeclarator
  AbstractDeclaratorWithPointer  ::=  Pointer ( DirectAbstractDeclarator )?
       DirectAbstractDeclarator  ::=  AbstractDimensionOrParameter DimensionOrParameterList
   AbstractDimensionOrParameter  ::=  AbstractDeclaratorClosed
                                  |   DimensionSize
                                  |   ParameterTypeListClosed
       AbstractDeclaratorClosed  ::=  "(" AbstractDeclarator ")"
       DimensionOrParameterList  ::=  ( ADimensionOrParameter )*
```

```
         ADimensionOrParameter  ::=  DimensionSize
                                 |   ParameterTypeListClosed
                   TypedefName  ::=  <IDENTIFIER>
                     Statement  ::=  LabeledStatement
                                 |   ExpressionStatement
                                 |   CallStatement
                                 |   CompoundStatement
                                 |   SelectionStatement
                                 |   IterationStatement
                                 |   JumpStatement
                                 |   UnknownPragma
                                 |   OmpConstruct
                                 |   OmpDirective
                                 |   UnknownCpp
                    UnknownCpp  ::=  "#" <UNKNOWN_CPP>
                        OmpEol  ::=  <OMP_CR>
                                 |   <OMP_NL>
                  OmpConstruct  ::=  ParallelConstruct
                                 |   ForConstruct
                                 |   SectionsConstruct
                                 |   SingleConstruct
                                 |   ParallelForConstruct
                                 |   ParallelSectionsConstruct
                                 |   TaskConstruct
                                 |   MasterConstruct
                                 |   CriticalConstruct
                                 |   AtomicConstruct
                                 |   OrderedConstruct
                  OmpDirective  ::=  BarrierDirective
                                 |   TaskwaitDirective
                                 |   TaskyieldDirective
                                 |   FlushDirective
             ParallelConstruct  ::=  OmpPragma ParallelDirective Statement
                     OmpPragma  ::=  "#" <PRAGMA> <OMP>
                 UnknownPragma  ::=  "#" <PRAGMA> <UNKNOWN_CPP>
             ParallelDirective  ::=  <PARALLEL> UniqueParallelOrDataClauseList OmpEol
UniqueParallelOrDataClauseList  ::=  ( AUniqueParallelOrDataClause )*
   AUniqueParallelOrDataClause  ::=  UniqueParallelClause
                                 |   DataClause
          UniqueParallelClause  ::=  IfClause
                                 |   NumThreadsClause
```

```
                          IfClause  ::=  <IF> "(" Expression ")"
                  NumThreadsClause  ::=  <NUM_THREADS> "(" Expression ")"
                        DataClause  ::=  OmpPrivateClause
                                     |   OmpFirstPrivateClause
                                     |   OmpLastPrivateClause
                                     |   OmpSharedClause
                                     |   OmpCopyinClause
                                     |   OmpDfltSharedClause
                                     |   OmpDfltNoneClause
                                     |   OmpReductionClause
                  OmpPrivateClause  ::=  <PRIVATE> "(" VariableList ")"
             OmpFirstPrivateClause  ::=  <FIRSTPRIVATE> "(" VariableList ")"
              OmpLastPrivateClause  ::=  <LASTPRIVATE> "(" VariableList ")"
                   OmpSharedClause  ::=  <SHARED> "(" VariableList ")"
                   OmpCopyinClause  ::=  <COPYIN> "(" VariableList ")"
               OmpDfltSharedClause  ::=  <DFLT> "(" <SHARED> ")"
                 OmpDfltNoneClause  ::=  <DFLT> "(" <NONE> ")"
                OmpReductionClause  ::=  <REDUCTION> "(" ReductionOp ":" VariableList ")"
                      ForConstruct  ::=  OmpPragma ForDirective OmpForHeader Statement
                      ForDirective  ::=  <FOR> UniqueForOrDataOrNowaitClauseList OmpEol
UniqueForOrDataOrNowaitClauseList  ::=  ( AUniqueForOrDataOrNowaitClause )*
   AUniqueForOrDataOrNowaitClause  ::=  UniqueForClause
                                     |   DataClause
                                     |   NowaitClause
                      NowaitClause  ::=  <NOWAIT>
                   UniqueForClause  ::=  <ORDERED>
                                     |   UniqueForClauseSchedule
                                     |   UniqueForCollapse
                 UniqueForCollapse  ::=  <COLLAPSE> "(" Expression ")"
           UniqueForClauseSchedule  ::=  <SCHEDULE> "(" ScheduleKind ( "," Expression )? ")"
                      ScheduleKind  ::=  <STATIC> | <DYNAMIC> | <GUIDED> | <RUNTIME>
                      OmpForHeader  ::=  <FOR> "(" OmpForInitExpression ";" OmpForCondition ";" Omp-
                                         ForReinitExpression ")"
              OmpForInitExpression  ::=  <IDENTIFIER> "=" Expression
                   OmpForCondition  ::=  OmpForLTCondition
                                     |   OmpForLECondition
                                     |   OmpForGTCondition
                                     |   OmpForGECondition
                 OmpForLTCondition  ::=  <IDENTIFIER> "<" Expression
                 OmpForLECondition  ::=  <IDENTIFIER> "<=" Expression
                 OmpForGTCondition  ::=  <IDENTIFIER> ">" Expression
```

OmpForGECondition	::=	<IDENTIFIER> ">=" Expression
OmpForReinitExpression	::=	PostIncrementId
	|	PostDecrementId
	|	PreIncrementId
	|	PreDecrementId
	|	ShortAssignPlus
	|	ShortAssignMinus
	|	OmpForAdditive
	|	OmpForSubtractive
	|	OmpForMultiplicative
PostIncrementId	::=	<IDENTIFIER> "++"
PostDecrementId	::=	<IDENTIFIER> "--"
PreIncrementId	::=	"++" <IDENTIFIER>
PreDecrementId	::=	"--" <IDENTIFIER>
ShortAssignPlus	::=	<IDENTIFIER> "+=" Expression
ShortAssignMinus	::=	<IDENTIFIER> "-=" Expression
OmpForAdditive	::=	<IDENTIFIER> "=" <IDENTIFIER> "+" AdditiveExpression
OmpForSubtractive	::=	<IDENTIFIER> "=" <IDENTIFIER> "-" AdditiveExpression
OmpForMultiplicative	::=	<IDENTIFIER> "=" MultiplicativeExpression "+" <IDENTIFIER>
SectionsConstruct	::=	OmpPragma <SECTIONS> NowaitDataClauseList OmpEol SectionsScope
NowaitDataClauseList	::=	(ANowaitDataClause)*
ANowaitDataClause	::=	NowaitClause
	|	DataClause
SectionsScope	::=	"{" (Statement)? (ASection)* "}"
ASection	::=	OmpPragma <SECTION> OmpEol Statement
SingleConstruct	::=	OmpPragma <SINGLE> SingleClauseList OmpEol Statement
SingleClauseList	::=	(ASingleClause)*
ASingleClause	::=	NowaitClause
	|	DataClause
	|	OmpCopyPrivateClause
OmpCopyPrivateClause	::=	<COPYPRIVATE> "(" VariableList ")"
TaskConstruct	::=	OmpPragma <TASK> (TaskClause)* OmpEol Statement
TaskClause	::=	DataClause
	|	UniqueTaskClause
UniqueTaskClause	::=	IfClause
	|	FinalClause
	|	UntiedClause
	|	MergeableClause
FinalClause	::=	<FINAL> "(" Expression ")"

```
                              UntiedClause  ::=  <UNTIED>
                           MergeableClause  ::=  <MERGEABLE>
                       ParallelForConstruct  ::=  OmpPragma <PARALLEL> <FOR> UniqueParallelOrUnique-
                                                  ForOrDataClauseList OmpEol OmpForHeader Statement
UniqueParallelOrUniqueForOrDataClauseList  ::=  ( AUniqueParallelOrUniqueForOrDataClause )*
   AUniqueParallelOrUniqueForOrDataClause  ::=  UniqueParallelClause
                                             |    UniqueForClause
                                             |    DataClause
                   ParallelSectionsConstruct  ::=  OmpPragma <PARALLEL> <SECTIONS> UniqueParallelOr-
                                                  DataClauseList OmpEol SectionsScope
                           MasterConstruct  ::=  OmpPragma <MASTER> OmpEol Statement
                         CriticalConstruct  ::=  OmpPragma <CRITICAL> ( RegionPhrase )? OmpEol State-
                                                  ment
                              RegionPhrase  ::=  "(" <IDENTIFIER> ")"
                           AtomicConstruct  ::=  OmpPragma <ATOMIC> ( AtomicClause )? OmpEol Statement
                              AtomicClause  ::=  <READ> | <WRITE> | <UPDATE> | <CAPTURE>
                            FlushDirective  ::=  OmpPragma <FLUSH> ( FlushVars )? OmpEol
                                 FlushVars  ::=  "(" VariableList ")"
                          OrderedConstruct  ::=  OmpPragma <ORDERED> OmpEol Statement
                          BarrierDirective  ::=  OmpPragma <BARRIER> OmpEol
                         TaskwaitDirective  ::=  OmpPragma <TASKWAIT> OmpEol
                        TaskyieldDirective  ::=  OmpPragma <TASKYIELD> OmpEol
                    ThreadPrivateDirective  ::=  OmpPragma <THREADPRIVATE> "(" VariableList ")" OmpEol
                  DeclareReductionDirective  ::=  OmpPragma <DECLARE> <REDUCTION> "(" ReductionOp
                                                  ":" ReductionTypeList ":" Expression ")" ( InitializerClause )?
                                                  OmpEol
                         ReductionTypeList  ::=  ( TypeSpecifier )*
                         InitializerClause  ::=  AssignInitializerClause
                                             |    ArgumentInitializerClause
                   AssignInitializerClause  ::=  <INITIALIZER> "(" <IDENTIFIER> "=" Initializer ")"
                 ArgumentInitializerClause  ::=  <INITIALIZER> "(" <IDENTIFIER> "(" ExpressionList ")" ")"
                               ReductionOp  ::=  <IDENTIFIER> | "+" | "*" | "-" | "&" | "^" | "|" | "||" | "&&"
                              VariableList  ::=  <IDENTIFIER> ( "," <IDENTIFIER> )*
                          LabeledStatement  ::=  SimpleLabeledStatement
                                             |    CaseLabeledStatement
                                             |    DefaultLabeledStatement
                    SimpleLabeledStatement  ::=  <IDENTIFIER> ":" Statement
                      CaseLabeledStatement  ::=  <CASE> ConstantExpression ":" Statement
                   DefaultLabeledStatement  ::=  <DFLT> ":" Statement
                       ExpressionStatement  ::=  ( Expression )? ";"
                         CompoundStatement  ::=  "{" ( CompoundStatementElement )* "}"
```

```
CompoundStatementElement ::= Declaration
                           |  Statement
      SelectionStatement ::= IfStatement
                           |  SwitchStatement
             IfStatement ::= <IF> "(" Expression ")" Statement ( <ELSE> Statement )?
         SwitchStatement ::= <SWITCH> "(" Expression ")" Statement
      IterationStatement ::= WhileStatement
                           |  DoStatement
                           |  ForStatement
          WhileStatement ::= <WHILE> "(" Expression ")" Statement
             DoStatement ::= <DO> Statement <WHILE> "(" Expression ")" ";"
            ForStatement ::= <FOR> "(" ( Expression )? ";" ( Expression )? ";" ( Expression )?
                             ")" Statement
           JumpStatement ::= GotoStatement
                           |  ContinueStatement
                           |  BreakStatement
                           |  ReturnStatement
           GotoStatement ::= <GOTO> <IDENTIFIER> ";"
       ContinueStatement ::= <CONTINUE> ";"
          BreakStatement ::= <BREAK> ";"
         ReturnStatement ::= <RETURN> ( Expression )? ";"
              Expression ::= AssignmentExpression ( "," AssignmentExpression )*
    AssignmentExpression ::= NonConditionalExpression
                           |  ConditionalExpression
NonConditionalExpression ::= UnaryExpression AssignmentOperator AssignmentExpression
      AssignmentOperator ::= "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" | "^=" |
                             "|="
   ConditionalExpression ::= LogicalORExpression ( "?" Expression ":" ConditionalExpres-
                             sion )?
      ConstantExpression ::= ConditionalExpression
     LogicalORExpression ::= LogicalANDExpression ( "||" LogicalORExpression )?
    LogicalANDExpression ::= InclusiveORExpression ( "&&" LogicalANDExpression )?
   InclusiveORExpression ::= ExclusiveORExpression ( "|" InclusiveORExpression )?
   ExclusiveORExpression ::= ANDExpression ( "^" ExclusiveORExpression )?
           ANDExpression ::= EqualityExpression ( "&" ANDExpression )?
      EqualityExpression ::= RelationalExpression ( EqualOptionalExpression )?
 EqualOptionalExpression ::= EqualExpression
                           |  NonEqualExpression
         EqualExpression ::= "==" EqualityExpression
      NonEqualExpression ::= "!=" EqualityExpression
    RelationalExpression ::= ShiftExpression ( RelationalOptionalExpression )?
```

```
     RelationalOptionalExpression  ::= RelationalLTExpression
                                     | RelationalGTExpression
                                     | RelationalLEExpression
                                     | RelationalGEExpression
           RelationalLTExpression  ::= "<" RelationalExpression
           RelationalGTExpression  ::= ">" RelationalExpression
           RelationalLEExpression  ::= "<=" RelationalExpression
           RelationalGEExpression  ::= ">=" RelationalExpression
                  ShiftExpression  ::= AdditiveExpression ( ShiftOptionalExpression )?
          ShiftOptionalExpression  ::= ShiftLeftExpression
                                     | ShiftRightExpression
              ShiftLeftExpression  ::= ">>" ShiftExpression
             ShiftRightExpression  ::= "<<" ShiftExpression
               AdditiveExpression  ::= MultiplicativeExpression ( AdditiveOptionalExpression )?
       AdditiveOptionalExpression  ::= AdditivePlusExpression
                                     | AdditiveMinusExpression
           AdditivePlusExpression  ::= "+" AdditiveExpression
          AdditiveMinusExpression  ::= "-" AdditiveExpression
         MultiplicativeExpression  ::= CastExpression ( MultiplicativeOptionalExpression )?
 MultiplicativeOptionalExpression  ::= MultiplicativeMultiExpression
                                     | MultiplicativeDivExpression
                                     | MultiplicativeModExpression
    MultiplicativeMultiExpression  ::= "*" MultiplicativeExpression
      MultiplicativeDivExpression  ::= "/" MultiplicativeExpression
      MultiplicativeModExpression  ::= "%" MultiplicativeExpression
                   CastExpression  ::= CastExpressionTyped
                                     | UnaryExpression
              CastExpressionTyped  ::= "(" TypeName ")" CastExpression
                  UnaryExpression  ::= UnaryExpressionPreIncrement
                                     | UnaryExpressionPreDecrement
                                     | UnarySizeofExpression
                                     | UnaryCastExpression
                                     | PostfixExpression
      UnaryExpressionPreIncrement  ::= "++" UnaryExpression
      UnaryExpressionPreDecrement  ::= "--" UnaryExpression
              UnaryCastExpression  ::= UnaryOperator CastExpression
            UnarySizeofExpression  ::= SizeofTypeName
                                     | SizeofUnaryExpression
            SizeofUnaryExpression  ::= <SIZEOF> UnaryExpression
                   SizeofTypeName  ::= <SIZEOF> "(" TypeName ")"
                    UnaryOperator  ::= "&" | "*" | "+" | "-" | "~" | "!"
```

PostfixExpression	::=	PrimaryExpression PostfixOperationsList
PostfixOperationsList	::=	(APostixOperation)*
APostixOperation	::=	BracketExpression
	|	ArgumentList
	|	DotId
	|	ArrowId
	|	PlusPlus
	|	MinusMinus
PlusPlus	::=	"++"
MinusMinus	::=	"--"
BracketExpression	::=	"[" Expression "]"
ArgumentList	::=	"(" (ExpressionList)? ")"
DotId	::=	"." <IDENTIFIER>
ArrowId	::=	"->" <IDENTIFIER>
PrimaryExpression	::=	<IDENTIFIER>
	|	Constant
	|	ExpressionClosed
ExpressionClosed	::=	"(" Expression ")"
ExpressionList	::=	AssignmentExpression ("," AssignmentExpression)*
Constant	::=	<INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> |
		<CHARACTER_LITERAL> | (<STRING_LITERAL>)+
BeginNode		
EndNode		
DummyFlushDirective		
SimplePrimaryExpression		
CallStatement		
PreCallNode		
PostCallNode		

REFERENCES

[1] OpenMP Application Programming Interface. Standard, OpenMP Architecture Review Board, November 2015.

[2] Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W. B., and Parady, B. SPEComp: A new benchmark suite for measuring parallel computer performance. In *OpenMP Shared Memory Parallel Programming*. Springer, 2001, pp. 1–10.

[3] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), ACM, pp. 72–81.

[4] Bronevetsky, G., and De Supinski, B. R. Formal specification of the OpenMP memory model. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming* (Berlin, Heidelberg, 2008), IWOMP'05/IWOMP'06, Springer-Verlag, pp. 324–346.

[5] Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing* (Washington, DC, USA, 2009), ICPP '09, IEEE Computer Society, pp. 124–131.

[6] ik Lee, S., Johnson, T. A., and Eigenmann, R. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing, 16th Intl. Workshop, College Station, TX, USA, Revised Papers, volume 2958 of LNCS* (2003), pp. 539–553.

[7] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.

In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[8] Ma, H., Zhao, R., Gao, X., and Zhang, Y. Barrier Optimization for OpenMP Program. In *Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing* (Washington, DC, USA, 2009), SNPD '09, IEEE Computer Society, pp. 495–500.

[9] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 21–65.

[10] Nandivada, V. K., Shirako, J., Zhao, J., and Sarkar, V. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst. 35*, 1 (Apr. 2013), 3:1–3:48.

[11] Netzer, R. H. B., and Miller, B. P. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst. 1*, 1 (Mar. 1992), 74–88.

[12] Nicolau, A. Percolation Scheduling: A Parallel Compilation Technique. Tech. rep., Ithaca, NY, USA, 1985.

[13] Nicolau, A., Li, G., and Kejariwal, A. Techniques for efficient placement of synchronization primitives. *SIGPLAN Not. 44*, 4 (Feb. 2009), 199–208.

[14] Nicolau, A., Li, G., Veidenbaum, A. V., and Kejariwal, A. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd international conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 169–180.

[15] Quinlan, D., Liao, C., Panas, T., Matzke, R., Schordan, M., Vuduc, R., and Yi, Q. ROSE User Manual: A Tool for Building Source-to-Source Translators. Tech. rep., Lawrence Livermore National Laboratory, 2013.

[16] Ramalingam, G. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22*, 2 (Mar. 2000), 416–430.

[17] Stallman, R. M., and DeveloperCommunity, G. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3.* CreateSpace, Paramount, CA, 2009.

[18] Van der Wijngaart, R. F., and Wong, P. NAS parallel benchmarks version 2.4. Tech. rep., NAS technical report, NAS-02-007, 2002.

[19] Viswakaran Sreelatha, J. K., and Balachandran, S. Compiler Enhanced Scheduling for OpenMP for Heterogeneous Multiprocessors. In 2^{nd} *Workshop on Energy Efficiency with Heterogeneous Computing* (Prague, Czech Republic, January 2016), EEHCO '16, ACM.

[20] Walsh, P. Performance of Barrier Synchronisation Algorithms on Modern Shared Memory Architectures. Master's thesis, The University of Edinburgh, Sept. 2004.