Paradyn Parallel Performance Tools

DataflowAPI Programmer's Guide

9.2 Release
Apr 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email dyninst-api@cs.wisc.edu
Web www.dyninst.org

Contents

1 Introduction

DataFlowAPI aggregates a collection of dataflow analysis algorithms that are useful in Dyninst development into a single library. These algorithms can also be foundations for users to build customized analyses. Currently, these algorithms include:

- Slicing takes a program location as input and can either slice backward to determine which instructions affect the results of the given program location, or slice forward to determine which instructions are affected by the results of the given program location. One key feature of our slicing implementation is that users can control where and when to stop slicing through a set of call back functions.
- Stack Analysis determines whether a register points to the stack and which stack location the register points to, when it does point to the stack.
- Symbolic Expansion and Evaluation convert instructions to several symbolic expressions. Each symbolic expression represents the overall effects of these instructions on a register or a memory location.
- Register Liveness determines whether a register is live or not at a program location. A register is live at a program location if it will be used later in the program before its content is overwritten.

2 Abstractions

DataflowAPI starts from the control flow graphs generated by ParseAPI and the instructions generated by InstructionAPI. From these, it provides dataflow facts in a variety of forms. The key abstractions used by DataflowAPI are:

- Abstract Location represents a register or memory location in the program. DataflowAPI provides three types of abstract locations: registers, stack, and help. A register abstract location represents a register and a register at two different program locations is treated as the same abstract location. A stack abstract location consists of the stack frame to which it belongs and the offset within the stack frame. A heap abstract location represents the virtual address of the heap variable.
- Abstract Region represents a set of abstract locations of the same type. If an abstract region contains only a single abstract location, the abstract location is precisely represented. If an abstract region contains more than one abstract locations, the region contains the type of the locations. In the cases where it represents memory (either heap or stack), an abstract region also contains the memory address calculation that gives rise to this region.
- Abstract Syntax Tree (AST) represents a symbolic expression of an instruction's semantics. Specifically, an AST specifies how the value of an abstract location is modified by this instruction.
- Assignment represents a single data dependency of abstract regions in an instruction. For example, xchg eax, ebx creates two assignments: one from pre-instruction eax to post-instruction ebx, and one from pre-instruction ebx to post-instruction eax.

- Stack Height represents the difference between a value in an abstract location and the stack pointer at a function's call site.

3 Examples

We show several examples of how to use DataflowAPI. In these examples, we assume that the mutatee has been parsed and we have a function and block to analyze. Users can refer to ParseAPI manual for how to get these functions and blocks.

3.1 Slicing

The following example uses DataflowAPI to perform a backward slice on an indirect jump instruction to determine the instructions that affect the calculation of the jump target. The goal of this example is to show (1) how to convert an instruction to assignments; (2) how to perform slicing on a given assignment; (3) how to extend the default `Slicer::Predicates` and write call back functions to control the behavior of slicing.

```
#include "Instruction.h"
#include "InstructionDecoder.h"
#include "slicing.h"

using namespace Dyninst;
using namespace ParseAPI;
using namespace InstructionAPI;
using namespace DataflowAPI;

// We extend the default predicates to control when to stop slicing
class ConstantPred : public Slicer::Predicates {
  public:
    // We do not want to track through memory writes
    virtual bool endAtPoint(Assignment::Ptr ap) {
        return ap->insn()->writesMemory();
    }

    // We can treat PC as a constant as its value is the address of the instruction
    virtual bool addPredecessor(AbsRegion reg) {
        if (reg.absloc().type() == Absloc::Register) {
            MachRegister r = reg.absloc().reg();
            return !r.isPC();
        }
        return true;
    }
};

// Assume that block b in function f ends with an indirect jump.
void AnalyzeJumpTarget(Function *f, Block *b) {
    // Decode the last instruction in this block, which should be a jump
```

```
    const unsigned char * buf =
         (const unsigned char*) b->obj()->cs()->getPtrToInstruction(b->last());
    InstructionDecoder dec(buf,
                           InstructionDecoder::maxInstructionLength,
                           b->obj()->cs()->getArch());
    Instruction::Ptr insn = dec.decode();

    // Convert the instruction to assignments
    AssignmentConverter ac(true);
    vector<Assignment::Ptr> assignments;
    ac.convert(insn, b->last(), f, b, assignments);

    // An instruction can corresponds to multiple assignment.
    // Here we look for the assignment that changes the PC.
    Assignment::Ptr pcAssign;
    for (auto ait = assignments.begin(); ait != assignments.end(); ++ait) {
        const AbsRegion &out = (*ait)->out();
        if (out.absloc().type() == Absloc::Register && out.absloc().reg().isPC()) {
            pcAssign = *ait;
            break;
        }
    }

    // Create a Slicer that will start from the given assignment
    Slicer s(pcAssign, b, f);

    // We use the customized predicates to control slicing
    ConstantPred mp;
    GraphPtr slice = s.backwardSlice(mp);
}
```

3.2 Symbolic Evaluation

The following example shows how to expand a slice to ASTs and analyze an AST. Suppose we have a slice representing the instructions that affect the jump target of an indirect jump instruction. We can get the expression of the jump targets and visit the expression to see if it is a constant.

```
#include "SymEval.h"
#include "DynAST.h"

// We extend the default ASTVisitor to check whether the AST is a constant
class ConstVisitor: public ASTVisitor {
  public:
    bool resolved;
    Address target;
    ConstVisitor() : resolved(true), target(0){}

    // We reach a constant node and record its value
    virtual AST::Ptr visit(DataflowAPI::ConstantAST * ast) {
```

```
        target = ast->val().val;
        return AST::Ptr();
    };

    // If the AST contains a variable
    // or an operation, then the control flow target cannot
    // be resolved through constant propagation
    virtual AST::Ptr visit(DataflowAPI::VariableAST *) {
        resolved = false;
        return AST::Ptr();
    };
    virtual AST::Ptr visit(DataflowAPI::RoseAST * ast) {
        resolved = false;

        // Recursively visit all children
        unsigned totalChildren = ast->numChildren();
        for (unsigned i = 0 ; i < totalChildren; ++i) {
            ast->child(i)->accept(this);
        }
        return AST::Ptr();
    };
};

Address ExpandSlice(GraphPtr slice, Assignment::Ptr pcAssign) {
    Result_t symRet;
    SymEval::expand(slice, symRet);

    // We get AST representing the jump target
    AST::Ptr pcExp = symRet[pcAssign];

    // We analyze the AST to see if it can actually be resolved by constant propagation
    ConstVisitor cv;
    pcExp->accept(&cv);
    if (cv.resolved) return cv.target;
    return 0;
}
```

3.3 Liveness Analysis

The following example shows how to query for live registers.

```
#include "Location.h"
#include "liveness.h"
#include "bitArray.h"
using namespace std;
using namespace Dyninst;
using namespace Dyninst::ParseAPI;

void LivenessAnalysis(Function *f, Block *b) {
```

```
    // Construct a liveness analyzer based on the address width of the mutatee.
    // 32-bit code and 64-bit code have different ABI.
    LivenessAnalyzer la(f->obj()->cs()->getAddressWidth());

    // Construct a liveness query location
    Location loc(f, b);

    // Query live registers at the block entry
    bitArray liveEntry;
    if (!la.query(loc, LivenessAnalyzer::Before, liveEntry)) {
        printf("Cannot look up live registers at block entry\n");
    }

    printf("There are %d registers live at the block entry\n", liveEntry.count());

    // Query live register at the block exit
    bitArray liveExit;
    if (!la.query(loc, LivenessAnalyzer::After, liveExit)) {
       printf("Cannot look up live registers at block exit\n");
    }

    printf("rbx is live or not at the block exit: %d\n", liveExit.test(la.getIndex(x86_64::rbx)))
}
```

3.4 Stack Analysis

The following examples shows how to use stack analysis.

```
#include "stackanalysis.h"

void StackHeight(Function *f, Block *b) {
    Address addr = block->start();

    // Get the stack heights at that address
    StackAnalysis sa(func);
    std::vector<std::pair<Absloc, StackAnalysis::Height>> heights;
    sa.findDefinedHeights(block, addr, heights);

    // Print out the stack heights
    for (auto iter = heights.begin(); iter != heights.end(); iter++) {
        const Absloc &loc = iter->first;
        const StackAnalysis::Height &height = iter->second;
        printf("%s := %s\n", loc.format().c_str(), height.format().c_str());
    }
}
```

4 API Reference

4.1 Class Assignment

Defined in: `Absloc.h`

An assignment represents data dependencies between an output abstract region that is modified by this instruction and several input abstract regions that are used by this instruction. An instruction may modify several abstract regions, so an instruction can correspond to multiple assignments.

```
typedef boost::shared_ptr<Assignment> Ptr;
```

Shared pointer for Assignment class.

```
const std::vector<AbsRegion> &inputs() const;
std::vector<AbsRegion> &inputs();
```

Return the input abstract regions.

```
const AbsRegion &out() const;
AbsRegion &out();
```

Return the output abstract region.

```
InstructionAPI::Instruction::Ptr insn() const;
```

Return the instruction that contains this assignment.

```
Address addr() const;
```

Return the address of this assignment.

```
ParseAPI::Function *func() const;
```

Return the function that contains this assignment.

```
ParseAPI::Block *block() const;
```

Return the block that contains this assignment.

```
const std::string format() const;
```

Return the string representation of this assignment.

4.2 Class AssignmentConverter

Defined in: `AbslocInterface.h`

This class should be used to convert instructions to assignments.

```
AssignmentConverter(bool cache, bool stack = true);
```

Construct an AssignmentConverter. When `cache` is `true`, this object will cache the conversion results for converted instructions. When `stack` is `true`, stack analysis is used to distinguish stack variables at different offset. When `stack` is `false`, the stack is treated as a single memory region.

```
void convert(InstructionAPI::Instruction::Ptr insn,
             const Address &addr,
             ParseAPI::Function *func,
             ParseAPI::Block *blk,
             std::vector<Assignment::Ptr> &assign);
```

Convert instruction `insn` to assignments and return these assignments in `assign`. The user also needs to provide the context of `insn`, including its address `addr`, function `func`, and block `blk`.

4.3 Class Absloc

Defined in: `Absloc.h`

Class Absloc represents an abstract location. Abstract locations can have the following types

Type	Meaning
Register	The abstract location represents a register
Stack	The abstract location represents a stack variable
Heap	The abstract location represents a heap variable
Unknown	The default type of abstract location

```
static Absloc makePC(Dyninst::Architecture arch);
static Absloc makeSP(Dyninst::Architecture arch);
static Absloc makeFP(Dyninst::Architecture arch);
```

Shortcut interfaces for creating abstract locations representing PC, SP, and FP

```
bool isPC() const;
bool isSP() const;
bool isFP() const;
```

Check whether this abstract location represents a PC, SP, or FP.

```
Absloc();
```

Create an Unknown type abstract location.

```
Absloc(MachRegister reg);
```

Create a Register type abstract location, representing register `reg`.

```
Absloc(Address addr):
```

Create a Heap type abstract location, representing a heap variable at address `addr`.

```
Absloc(int o,
       int r,
       ParseAPI::Function *f);
```

Create a Stack type abstract location, representing a stack variable in the frame of function `f` and at offset `o` within the frame.

```
std::string format() const;
```

Return the string representation of this abstract location.

```
const Type& type() const;
```

Return the type of this abstract location.

```
bool isValid() const;
```

Check whether this abstract location is valid or not. Return `true` when the type is not Unknown.

```
const MachRegister &reg() const;
```

Return the register represented by this abstract location. This method should only be called when this abstract location truly represents a register.

```
int off() const;
```

Return the offset of the stack variable represented by this abstract location. This method should only be called when this abstract location truly represents a stack variable.

```
int region() const;
```

Return the region of the stack variable represented by this abstract location. This method should only be called when this abstract location truly represents a stack variable.

```
ParseAPI::Function *func() const;
```

Return the function of the stack variable represented by this abstract location. This method should only be called when this abstract location truly represents a stack variable.

```
Address addr() const;
```

Return the address of the heap variable represented by this abstract location. This method should only be called when this abstract location truly represents a heap variable.

```
bool operator<(const Absloc &rhs) const;
bool operator==(const Absloc &rhs) const;
bool operator!=(const Absloc &rhs) const;
```

Comparison operators

4.4 Class AbsRegion

Defined in: `Absloc.h`

Class AbsRegion represents a set of abstract locations of the same type.

```
AbsRegion();
```

Create a default abstract region.

```
AbsRegion(Absloc::Type t);
```

Create an abstract region representing all abstract locations with type `t`.

```
AbsRegion(Absloc a);
```

Create an abstract region representing a single abstract location `a`.

```
bool contains(const Absloc::Type t) const;
bool contains(const Absloc &abs) const;
bool contains(const AbsRegion &rhs) const;
```

Return `true` if this abstract region contains abstract locations of type `t`, contains abstract location `abs`, or contains abstract region `rhs`.

```
bool containsOfType(Absloc::Type t) const;
```

Return `true` if this abstract region contains abstract locations in type `t`.

```
bool operator==(const AbsRegion &rhs) const;
bool operator!=(const AbsRegion &rhs) const;
bool operator<(const AbsRegion &rhs) const;
```

Comparison operators

```
const std::string format() const;
```

Return the string representation of the abstract region.

```
Absloc absloc() const;
```

Return the abstract location in this abstract region.

```
Absloc::Type type() const;
```

Return the type of this abstract region.

```
AST::Ptr generator() const;
```

If this abstract region represents memory locations, this method returns address calculation of the memory access.

```
bool isImprecise() const;
```

Return `true` if this abstract region represents more than one abstract locations.

4.5 Class AbsRegionConverter

Defined in: `AbslocInterface.h`

Class AbsRegionConverter converts instructions to abstract regions.

```
AbsRegionConverter(bool cache, bool stack = true);
```

Create an AbsRegionConverter. When `cache` is `true`, this object will cache the conversion results for converted instructions. When `stack` is `true`, stack analysis is used to distinguish stack variables at different offsets. When `stack` is `false`, the stack is treated as a single memory region.

```
void convertAll(InstructionAPI::Expression::Ptr expr,
                Address addr,
ParseAPI::Function *func,
ParseAPI::Block *block,
std::vector<AbsRegion> &regions);
```

Create all abstract regions used in `expr` and return them in `regions`. All registers appear in `expr` will have a separate abstract region. If the expression represents a memory access, we will also create a heap or stack abstract region depending on where it accesses. `addr`, `func`, and `blocks` specify the contexts of the expression. If PC appears in this expression, we assume the expression is at address `addr` and replace PC with a constant value `addr`.

```
void convertAll(InstructionAPI::Instruction::Ptr insn,
                Address addr,
                ParseAPI::Function *func,
                ParseAPI::Block *block,
                std::vector<AbsRegion> &used,
                std::vector<AbsRegion> &defined);
```

Create abstract regions appeared in instruction `insn`. Input abstract regions of this instructions are returned in `used` and output abstract regions are returned in `defined`.If the expression represents a memory access, we will also create a heap or stack abstract region depending on where it accesses. `addr`, `func`, and `blocks` specify the contexts of the expression. If PC appears in this expression, we assume the expression is at address `addr` and replace PC with a constant value `addr`.

```
AbsRegion convert(InstructionAPI::RegisterAST::Ptr reg);
```

Create an abstract region representing the register `reg`.

```
AbsRegion convert(InstructionAPI::Expression::Ptr expr,
                  Address addr,
                  ParseAPI::Function *func,
                  ParseAPI::Block *block);
```

Create and return the single abstract region represented by `expr`.

4.6 Class Graph

Defined in: `Graph.h`

We provide a generic graph interface, which allows users adding, deleting, iterating nodes and edges in a graph. Our slicing algorithms are implemented upon this graph interface, so users can inherit the defined classes for customization.

```
typedef boost::shared_ptr<Graph> Ptr;
```

Shared pointer for Graph

```
virtual void entryNodes(NodeIterator &begin, NodeIterator &end);
```

The entry nodes (nodes without any incoming edges) of the graph.

```
virtual void exitNodes(NodeIterator &begin, NodeIterator &end);
```

The exit nodes (nodes without any outgoing edges) of the graph.

```
virtual void allNodes(NodeIterator &begin, NodeIterator &end);
```

Iterate all nodes in the graph.

```
bool printDOT(const std::string& fileName);
```

Output the graph in dot format.

```
static Graph::Ptr createGraph();
```

Return an empty graph.

```
void insertPair(NodePtr source, NodePtr target, EdgePtr edge = EdgePtr());
```

Insert an pair of node into the graph and create a new edge `edge` from `source` to `target`.

```
virtual void insertEntryNode(NodePtr entry);
virtual void insertExitNode(NodePtr exit);
```

Insert a node as an entry/exit node

```
virtual void markAsEntryNode(NodePtr entry);
virtual void markAsExitNode(NodePtr exit);
```

Mark a node that has been added to this graph as an entry/exit node.

```
void deleteNode(NodePtr node);
void addNode(NodePtr node);
```

Delete / Add a node.

```
bool isEntryNode(NodePtr node);
bool isExitNode(NodePtr node);
```

Check whether a node is an entry / exit node

```
void clearEntryNodes();
void clearExitNodes();
```

Clear the marking of entry / exit nodes. Note that the nodes are not deleted from the graph.

```
unsigned size() const;
```

Return the number of nodes in the graph.

4.7 Class Node

Defined in: `Node.h`

```
typedef boost::shared_ptr<Node> Ptr;
```

shared pointer for Node

```
void ins(EdgeIterator &begin, EdgeIterator &end);
void outs(EdgeIterator &begin, EdgeIterator &end);
```

Iterate over incoming/outgoing edges of this node.

```
void ins(NodeIterator &begin, NodeIterator &end);
void outs(NodeIterator &begin, NodeIterator &end);
```

Iterate over adjacent nodes connected with incoming/outgoing edges of this node

.

```
bool hasInEdges();
bool hasOutEdges();
```

Return `true` if this node has incoming/outgoing edges.

```
void deleteInEdge(EdgeIterator e);
void deleteOutEdge(EdgeIterator e);
```

Delete an incoming/outgoing edge.

```
virtual Address addr() const;
```

Return the address of this node.

```
virtual std::string format() const = 0;
```

Return the string representation.

```
class NodeIterator;
```

Iterator for nodes. Common iterator operations including `++`, `-`, and dereferencing are supported.

4.8 Class Edge

Defined in: `Edge.h`

```
typedef boost::shared_ptr<Edge> Edge::Ptr;
```

Shared pointer for `Edge`.

```
static Edge::Ptr Edge::createEdge(const Node::Ptr source, const Node::Ptr target);
```

Create a new directed edge from `source` to `target`.

```
Node::Ptr Edge::source() const;
Node::Ptr Edge::target() const;
```

Return the source / target node.

```
void Edge::setSource(Node::Ptr source);
void Edge::setTarget(Node::Ptr target);
```

Set the source / target node.

```
class EdgeIterator;
```

Iterator for edges. Common iterator operations including `++`, `-`, and dereferencing are supported.

4.9 Class Slicer

Defined in: `slicing.h`

Class Slicer is the main interface for performing forward and backward slicing. The slicing algorithm starts with a user provided Assignment and generates a graph as the slicing results. The nodes in the generated Graph are individual assignments that affect the starting assignment (backward slicing) or are affected by the starting assignment (forward slicing). The edges in the graph are directed and represent either data flow dependencies or control flow dependencies.

We provide callback functions and allow users controlling when to stop slicing. In particular, class `Slicer::Predicates` contains a collection of call back functions that can control the specific behaviors of the slicer. Users can inherit Predicates class to provide customized stopping criteria for the slicer.

```
Slicer(AssignmentPtr a,
       ParseAPI::Block *block,
       ParseAPI::Function *func,
       bool cache = true,
       bool stackAnalysis = true);
```

Construct a slicer, which can then be used to perform forward or backward slicing starting at the assignment `a`. `block` and `func` represent the context of assignment `a`. `cache` specifies whether the slicer will cache the results of conversions from instructions to assignments. `stackAnalysis` specifies whether the slicer will invoke stack analysis to distinguish stack variables.

```
GraphPtr forwardSlice(Predicates &predicates);
GraphPtr backwardSlice(Predicates &predicates);
```

Perform forward or backward slicing and use `predicates` to control the stopping criteria and return the slicing results as a graph

A slice is represented as a Graph. The nodes and edges are defined as below:

```
class SliceNode : public Node
```

The default node data type in a slice graph.

```
typedef boost::shared_ptr<SliceNode> Ptr;
static SliceNode::Ptr SliceNode::create(AssignmentPtr ptr,
                                        ParseAPI::Block *block,
                                        ParseAPI::Function *func);
```

Create a slice node, which represents assignment `ptr` in basic block `block` and function `func`.

Class SliceNode has the following methods to retrieve information associated the node:

Method name	Return type	Method description
block	ParseAPI::Block*	Basic block of this SliceNode.
func	ParseAPI::Function*	Function of this SliceNode.
addr	Address	Address of this SliceNode.
assign	Assignment::Ptr	Assignment of this SliceNode.
format	std::string	String representation of this SliceNode.

```
class SliceEdge : public Edge
```

The default edge data type in a slice graph.

```
typedef boost::shared_ptr<SliceEdge> Ptr;
static SliceEdge::Ptr create(SliceNode::Ptr source,
                             SliceNode::Ptr target,
                             AbsRegion const&data);
```

Create a slice edge from `source` to `target` and the edge presents a dependency about abstract region `data`.

```
const AbsRegion &data() const;
```

Get the data annotated on this edge.

4.10 Class Slicer::Predicates

Defined in: `slicing.h`

Class Predicates abstracts the stopping criteria of slicing. Users can inherit this class to control slicing in various situations, including whether or not performing inter-procedural slicing, whether or not searching for control flow dependencies, and stopping slicing after discovering certain assignments. We provide a set of call back functions that allow users to dynamically control the behavior of the Slicer.

```
Predicates();
```

Construct a default predicate, which will only search for intraprocedural data flow dependencies.

```
bool searchForControlFlowDep();
```

Return `true` if this predicate will search for control flow dependencies. Otherwise, return `false`.

```
void setSearchForControlFlowDep(bool cfd);
```

Change whether or not to search for control flow dependencies according to `cfd`.

```
virtual bool widenAtPoint(AssignmentPtr) { return false; }
```

The default behavior is to return `false`.

```
virtual bool endAtPoint(AssignmentPtr);
```

In backward slicing, after we find a match for an assignment, we pass it to this function. This function should return `true` if the user does not want to continue searching for this assignment. Otherwise, it should return `false`. The default behavior of this function is to always return `false`.

```
typedef std::pair<ParseAPI::Function *, int> StackDepth_t;
typedef std::stack<StackDepth_t> CallStack_t;
virtual bool followCall(ParseAPI::Function * callee,
                        CallStack_t & cs,
                        AbsRegion argument);
```

This predicate function is called when the slicer reaches a direct call site. If it returns `true`, the slicer will follow into the callee function `callee`. This function also takes input `cs`, which represents the call stack of the followed callee functions from the starting point of the slicing to this call site, and `argument`, which represents the variable to slice with in the callee function. This function is default to always return `false`. Note that as Dyninst currently does not try to resolve indirect calls, the slicer will NOT call this function at an indirect call site.

```
virtual std::vector<ParseAPI::Function *>
          followCallBackward(ParseAPI::Block * caller,
                             CallStack_t & cs,
                             AbsRegion argument);
```

This predicate function is called when the slicer reaches the entry of a function in the case of backward slicing or reaches a return instruction in the case of forward slicing. It returns a vector of caller functions that the user want the slicer to continue to follow. This function takes input `caller`, which represents the call block of the caller, `cs`, which represents the caller functions that have been followed to this place, and `argument`, which represents the variable to slice with in the caller function. This function is default to always return an empty vector.

```
virtual bool addPredecessor(AbsRegion reg);
```

In backward slicing, after we match an assignment at a location, the matched AbsRegion `reg` is passed to this predicate function. This function should return `true` if the user wants to continue to search for dependencies for this AbsRegion. Otherwise, this function should return `true`. The default behavior of this function is to always return `true`.

```
virtual bool addNodeCallback(AssignmentPtr assign,
                             std::set<ParseAPI::Edge*> &visited);
```

In backward slicing, this function is called when the slicer adds a new node to the slice. The newly added assignment `assign` and the set of control flow edges `visited` that have been visited so far are passed to this function. This function should return `true` if the user wants to continue slicing. If this function returns `false`, the Slicer will not continue to search along the path. The default behavior of this function is to always return `true`.

4.11 Class StackAnalysis

The StackAnalysis interface is used to determine the possible stack heights of abstract locations at any instruction in a function. Due to there often being many paths through the CFG to reach a given instruction, abstract locations may have different stack heights depending on the path taken to reach that instruction. In other cases, StackAnalysis is unable to adequately determine what is contained in an abstract location. In both situations, StackAnalysis is conservative in its reported stack heights. The table below explains what the reported stack heights mean.

Reported stack height	Meaning
TOP	On all paths to this instruction, the specified abstract location contains a value that does not point to the stack.
x (some number)	On at least one path to this instruction, the specified abstract location has a stack height of x. On all other paths, the abstract location either has a stack height of x or doesn't point to the stack.
BOTTOM	There are three possible meanings: 1. On at least one path to this instruction, StackAnalysis was unable to determine whether or not the specified abstract location points to the stack. 2. On at least one path to this instruction, StackAnalysis determined that the specified abstract location points to the stack but could not determine the exact stack height. 3. On at least two paths to this instruction, the specified abstract location pointed to different parts of the stack.

```
StackAnalysis(ParseAPI::Function *f)
```

Constructs a StackAnalysis object for function `f`.

```
StackAnalysis::Height find(ParseAPI::Block *b, Address addr, Absloc loc)
```

Returns the stack height of abstract location `loc` before execution of the instruction with address `addr` contained in basic block `b`. The address `addr` must be contained in block `b`, and block `b` must be contained in the function used to create this StackAnalysis object.

```
StackAnalysis::Height findSP(ParseAPI::Block *b, Address addr)
StackAnalysis::Height findFP(ParseAPI::Block *b, Address addr)
```

Returns the stack height of the stack pointer and frame pointer, respectively, before execution of the instruction with address `addr` contained in basic block `b`. The address `addr` must be contained in block `b`, and block `b` must be contained in the function used to create this StackAnalysis object.

```
void findDefinedHeights(ParseAPI::Block *b,
                        Address addr,
                        std::vector<std::pair<Absloc, StackAnalysis::Height>> &heights)
```

Writes to the vector `heights` all defined <abstract location, stack height> pairs before execution of the instruction with address `addr` contained in basic block `b`. Note that abstract locations with stack heights of TOP (i.e. they do not point to the stack) are not written to `heights`. The address `addr` must be contained in block `b`, and block `b` must be contained in the function used to create this StackAnalysis object.

4.12 Class StackAnalysis::Height

Defined in: `stackanalysis.h`

The Height class is used to represent the abstract notion of stack heights. Every Height object represents a stack height of either TOP, BOTTOM, or x, where x is some integral number. The Height class also defines methods for comparing, combining, and modifying stack heights in various ways.

```
typedef signed long Height_t
```

The underlying data type used to convert between Height objects and integral values.

Method name	Return type	Method description
height	Height_t	This stack height as an integral value.
format	std::string	This stack height as a string.
isTop	bool	True if this stack height is TOP.
isBottom	bool	True if this stack height is BOTTOM.

```
Height(const Height_t h)
```

Creates a Height object with stack height `h`.

```
Height()
```

Creates a Height object with stack height TOP.

```
bool operator<(const Height &rhs) const
bool operator>(const Height &rhs) const
bool operator<=(const Height &rhs) const
bool operator>=(const Height &rhs) const
bool operator==(const Height &rhs) const
bool operator!=(const Height &rhs) const
```

Comparison operators for Height objects. Compares based on the integral stack height treating TOP as MAX_HEIGHT and BOTTOM as MIN_HEIGHT.

```
Height &operator+=(const Height &rhs)
Height &operator+=(const signed long &rhs)
const Height operator+(const Height &rhs) const
const Height operator+(const signed long &rhs) const
const Height operator-(const Height &rhs) const
```

Returns the result of basic arithmetic on Height objects according to the following rules, where x and y are integral stack heights and S represents any stack height:

- $TOP + TOP = TOP$
- $TOP + x = BOTTOM$
- $x + y = (x + y)$
- $BOTTOM + S = BOTTOM$

Note that the subtraction rules can be obtained by replacing all + signs with - signs.

The `operator+` and `operator-` methods leave this Height object unmodified while the `operator+=` methods update this Height object with the result of the computation. For the methods where `rhs` is a `const signed long`, it is not possible to set `rhs` to TOP or BOTTOM.

4.13 Class AST

Defined in: `DynAST.h`

We provide a generic AST framework to represent tree structures. One example use case is to represent instruction semantics with symbolic expressions. This AST framework include the base class definitions for tree nodes and visitors. Users can inherit tree node classes to create their own AST structure and AST visitors to write their own analyses for the AST.

All AST node classes should be derived from class AST and currently we have the following types of AST nodes.

AST::ID	Meaning
V_AST	Base class type
V_BottomAST	Bottom AST node
V_ConstantAST	Constant AST node
V_VariableAST	Variable AST node
V_RoseAST	ROSEOperation AST node
V_StackAST	Stack AST node

```
typedef boost::shared_ptr<AST> Ptr;
```

Shared pointer for class AST.

```
typedef std::vector<AST::Ptr> Children;
```

The container type for the children of this AST.

```
bool operator==(const AST &rhs) const;
bool equals(AST::Ptr rhs);
```

Check whether two AST nodes are equal. Return `true` when two nodes are in the same type and are equal according to the `==` operator of that type.

```
virtual unsigned numChildren() const;
```

Return the number of children of this node.

```
virtual AST::Ptr child(unsigned i) const;
```

Return the `i`th child.

```
virtual const std::string format() const = 0;
```

Return the string representation of the node.

```
static AST::Ptr substitute(AST::Ptr in, AST::Ptr a, AST::Ptr b);
```

Substitute every occurrence of `a` with `b` in AST `in`. Return a new AST after the substitution.

```
virtual AST::ID AST::getID() const;
```

Return the class type ID of this node.

```
virtual Ptr accept(ASTVisitor *v);
```

Apply visitor `v` to this node. Note that this method will not automatically apply the visitor to its children.

```
virtual void AST::setChild(int i, AST::Ptr c);
```

Set the `i`th child of this node to `c`.

4.14 Class SymEval

Defined in: `SymEval.h`

Class SymEval provides interfaces for expanding an instruction to its symbolic expression and expanding a slice graph to symbolic expressions for all abstract locations defined in this slice.

```
typedef std::map<Assignment::Ptr, AST::Ptr, AssignmentPtrValueComp> Result_t;
```

This data type represents the results of symbolic expansion of a slice. Each assignment in the slice has a corresponding AST.

```
static std::pair<AST::Ptr, bool> expand(const Assignment::Ptr &assignment,
                                        bool applyVisitors = true);
```

This interface expands a single assignment given by `assignment` and returns a `std::pair`, in which the first element is the AST after expansion and the second element is a bool indicating whether the expansion succeeded or not. `applyVisitors` specifies whether or not to perform stack analysis to precisely track stack variables.

```
static bool expand(Result_t &res,
                   std::set<InstructionPtr> &failedInsns,
   bool applyVisitors = true);
```

This interface expands a set of assignment prepared in `res`. The corresponding ASTs are written back into `res` and all instructions that failed during expansion are inserted into `failedInsns`. `applyVisitors` specifies whether or not to perform stack analysis to precisely track stack variables. This function returns `true` when all assignments in `res` are successfully expanded.

Retval_t	Meaning
FAILED	failed
WIDEN_NODE	widen
FAILED_TRANSLATION	failed translation
SKIPPED_INPUT	skipped input
SUCCESS	success

```
static Retval_t expand(Dyninst::Graph::Ptr slice, DataflowAPI::Result_t &res);
```

This interface expands a slice and return an AST for each assignment in the slice. This function will perform substitution of ASTs.

We use an AST to represent the symbolic expressions of an assignment. A symbolic expression AST contains internal node type `RoseAST`, which abstracts the operations performed with its child nodes, and two leave node types: `VariableAST` and `ConstantAST`.

`RoseAST`, `VariableAST`, and `ConstantAST` all extend class `AST`. Besides the methods provided by class `AST`, `RoseAST`, `VariableAST`, and `ConstantAST` each have a different data structure associated with them.

```
Variable& VariableAST::val() const;
Constant& ConstantAST::val() const;
ROSEOperation & RoseAST::val() const;
```

We now describe data structure `Variable`, `Constant`, and `ROSEOperation`.

```
struct Variable;
```

A `Variable` represents an abstract region at a particular address.

```
Variable::Variable();
Variable::Variable(AbsRegion r);
Variable::Variable(AbsRegion r, Address a);
```

The constructors of class Variable.

```
bool Variable::operator==(const Variable &rhs) const;
bool Variable::operator<(const Variable &rhs) const;
```

Two Variable objects are equal when their AbsRegion are equal and their addresses are equal.

```
const std::string Variable::format() const;
```

Return the string representation of the Variable.

```
AbsRegion Variable::reg;
Address Variable::addr;
```

The abstraction region and the address of this Variable.

```
struct Constant;
```

A `Constant` object represents a constant value in code.

```
Constant::Constant();
Constant::Constant(uint64_t v);
Constnat::Constant(uint64_t v, size_t s);
```

Constructors a Constant objects.

```
bool Constant::operator==(const Constant &rhs) const;
bool Constant::operator<(const Constant &rhs) const;
```

Comparison operators for Constant objects. Comparison is based on the value and size.

```
const std::string Constant::format() const;
```

Return the string representation of the Constant object.

```
uint64_t Constant::val;
size_t Constant::size;
```

The numerical value and bit size of this value.

```
struct ROSEOperation;
```

`ROSEOperation` defines the following operations and we represent the semantics of all instructions with these operations.

ROSEOperation::Op	Meaning
nullOp	No operation
extractOp	Extract bit ranges from a value
invertOp	Flip every bit
negateOp	Negate the value
signExtendOp	Sign-extend the value
equalToZeroOp	Check whether the value is zero or not
generateMaskOp	Generate mask
LSBSetOp	LSB set op
MSBSetOp	MSB set op
concatOp	Concatenate two values to form a new value
andOp	Bit-wise and operation
orOp	Bit-wise or operation
xorOp	Bit-wise xor operation
addOp	Add operation
rotateLOp	Rotate to left operation
rotateROp	Rotate to right operation
shiftLOp	Shift to left operation
shiftROp	Shift to right operation
shiftRArithOp	Arithmetic shift to right operation
derefOp	Dereference memory operation
writeRepOp	Write rep operation
writeOp	Write operation
ifOp	if operation
sMultOp	Signed multiplication operation
uMultOp	Unsigned multiplication operation
sDivOp	Signed division operation
sModOp	Signed modular operation
uDivOp	Unsigned division operation
uModOp	Unsigned modular operation
extendOp	Zero extend operation
extendMSBOp	Extend the most significant bit operation

```
ROSEOperation::ROSEOperation(Op o) : op(o);
ROSEOperation::ROSEOperation(Op o, size_t s);
```

Constructors for ROSEOperation

```
bool ROSEOperation::operator==(const ROSEOperation &rhs) const;
```

Equal operator

```
const std::string ROSEOperation::format() const;
```

Return the string representation.

```
ROSEOperation::Op ROSEOperation::op;
size_t ROSEOperation::size;
```

4.15 Class ASTVisitor

The ASTVisitor class defines callback functions to apply during visiting an AST for each AST node type. Users can inherit this class to write customized analyses for ASTs.

```
typedef boost::shared_ptr<AST> ASTVisitor::ASTPtr;
virtual ASTVisitor::ASTPtr ASTVisitor::visit(AST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::BottomAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::ConstantAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::VariableAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(DataflowAPI::RoseAST *);
virtual ASTVisitor::ASTPtr ASTVisitor::visit(StackAST *);
```

Callback functions for visiting each type of AST node. The default behavior is returning the input parameter.