InstructionAPI Reference Manual 1.1

Generated by Doxygen 1.4.7

2 Dec 2009

Contents

1   InstructionAPI Introduction

When analyzing and modifying binary code, it is necessary to translate between raw binary instructions and an abstract form that describes the semantics of the instructions. As a part of the Dyninst project, we have developed the Instruction API, an API and library for decoding and representing machine instructions in a platform-independent manner. The Instruction API includes methods for decoding machine language, convenient abstractions for its analysis, and methods to produce disassembly from those abstractions. The current implementation supports the IA32, IA-64, AMD-64, SPARC, POWER, and PowerPC instruction sets. The Instruction API has the following basic capabilities:

- Decoding: interpreting a sequence of bytes as a machine instruction in a given machine language.

- Abstract representation: representing the behavior of that instruction as an abstract syntax tree.

- Disassembly: translating an abstract representation of a machine instruction into a string representation of the corresponding assembly language instruction.

Our goal in designing the Instruction API is to provide a representation of machine instructions that can be manipulated by higher-level algorithms with minimal knowledge of platform-specific details. In addition, users who need platform-specific information should be able to access it. To do so, we provide an interface that disassembles a machine instruction, extracts an operation and its operands, converts the operands to abstract syntax trees, and presents this to the user. A user of the Instruction API can work at a level of abstraction slightly higher than assembly language, rather than working directly with machine language. Additionally, by converting the operands to abstract syntax trees, we make it possible to analyze the operands in a uniform manner, regardless of the complexity involved in the operand's actual computation.

2   InstructionAPI Modules and Abstractions

The Instruction API contains three major components: the top-level instruction representation, the abstract syntax trees representing the operands of an instruction, and the decoder that creates the entire representation. We will present an overview of the features and uses of each of these three components, followed by an example of how the Instruction API can be applied to binary analysis.

2.1   Instruction Interface

The Instruction API represents a machine language instruction as an Instruction object, which contains an Operation and a collection of Operands. The Operation contains the following items:

- The mnemonic for the machine language instruction represented by its associated Instruction

- The number of operands accepted by the Operation

- Which Operands are read and/or written by the associated machine operation

- What other registers (if any) are affected by the underlying machine operation

Each Operand contains flags to indicate whether it is read, written, or both by the machine instruction represented by its parent Instruction, and contains a Expression abstract syntax tree

representing the operations required to compute the value of the operand. Figure 1 depicts these ownership relationships within an Instruction.
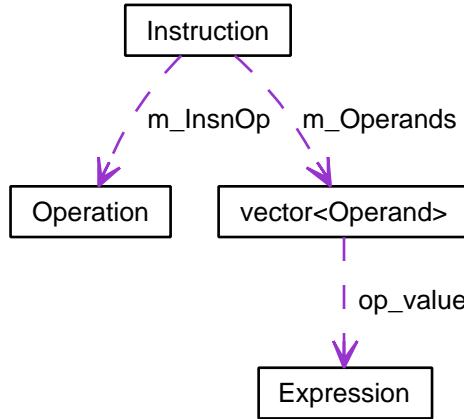

Figure 1: An Instruction and the objects it owns

Instruction objects provide two types of interfaces: direct read access to their components, and common summary operations on those components. The first interface allows access to the Operation and Operand data members, and each Operand object in turn allows traversal of its abstract syntax tree. More details about how to work with this abstract syntax tree can be found in **InstructionAST Hierarchy** (Section 2.3). This interface would be used, for example, in a data flow analysis where a user wants to evaluate the results of an effective address computation given a known register state.

The second interface allows a user to get the sets of registers read and written by the instruction, information about how the instruction accesses memory, and information about how the instruction affects control flow, without having to manipulate the Operands directly. For instance, a user could implement a register liveness analysis algorithm using just this second interface (namely the `getReadSet` and `getWriteSet` functions).

2.2   Instruction Decoding

An InstructionDecoder interprets a sequence of bytes according to a given machine language and transforms them into an instruction representation. It determines the opcode of the machine instruction, translates that opcode to an Operation object, uses that Operation to determine how to decode the instruction's Operands, and produces a decoded Instruction.

Figure 2: The InstructionDecoder's inputs and outputs

Instruction decoders are built from the following elements:

- A function to find and extract an opcode given a pointer into a buffer that points to the beginning of a machine instruction

- A table that, for a particular architecture, maps opcodes to Operations and functions that decode Operands

From these elements, it is possible to generalize the construction of Instructions from Operations and Operands to an entirely platform-independent algorithm. Likewise, much of the construction of the ASTs representing each operand can be performed in a platform-independent manner.

2.3   InstructionAST Hierarchy

The AST representation of an operand encapsulates the operations performed on registers and immediates to produce an operand for the machine language instruction.

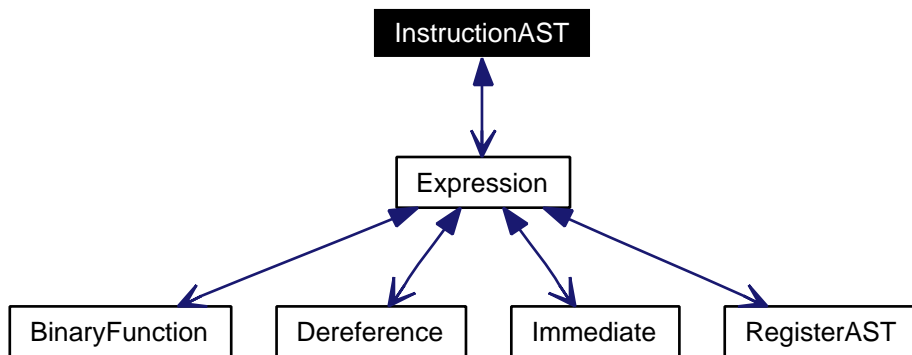The inheritance hierarchy of the AST classes is shown in Figure 3.

Figure 3: The InstructionAST inheritance hierarchy

The grammar for these AST representations is simple: all leaves must be RegisterAST or Immediate nodes. These nodes may be combined using a BinaryFunction node, which may be constructed as either an addition or a multiplication. Also, a single node may descend from a Dereference node, which treats its child as a memory address. Figure 4 shows the allowable parent/child relationships within a given tree, and Figure 5 shows how an example IA32 instruction is represented using these objects.
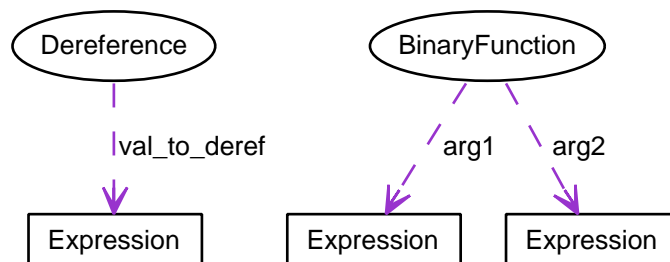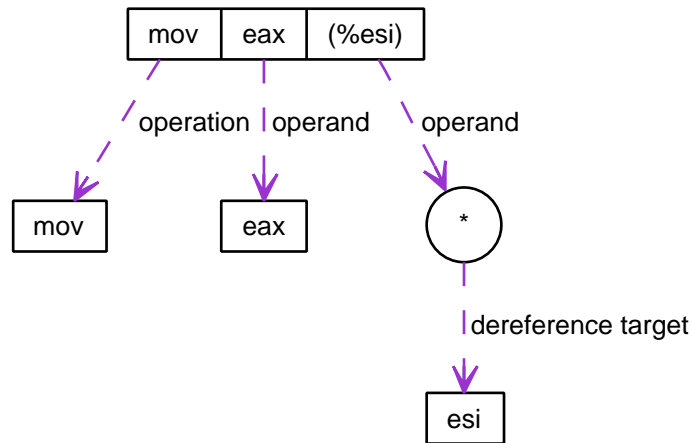
Figure 4: InstructionAST intermediate node types and the objects they own

Figure 5: The decomposition of `mov %eax, (%esi)`

These ASTs may be searched for leaf elements or subtrees (via `getUses` and `isUsed`) and traversed breadth-first or depth-first (via `getChildren`).

Any node in these ASTs may be evaluated. Evaluation attempts to determine the value represented by a node. If successful, it will return that value and cache it in the node. The tree structure, combined with the evaluation mechanism, allows the substitution of known register and memory values into an operand, regardless of whether those values are known at the time an instruction is decoded. More details on this mechanism may be found in **Dyninst::InstructionAPI::Expression** (Section 3.6).

3 InstructionAPI Class Reference

3.1 Instruction Class

The Instruction class is a generic instruction representation that contains operands, read/write semantic information about those operands, and information about what other registers and memory locations are affected by the operation the instruction performs.

The purpose of an Instruction object is to join an Operation with a sequence of Operands, and provide an interface for some common summary analyses (namely, the read/write sets, memory access information, and control flow information).

The Operation contains knowledge about its mnemonic and sufficient semantic details to answer the following questions:

- What Operands are read/written?
- What registers are implicitly read/written?
- What memory locations are implicitly read/written?
- What are the possible control flow successors of this instruction?

Each Operand is an AST built from RegisterAST and Immediate leaves. For each Operand, you may determine:

- Registers read

- Registers written

- Whether memory is read or written

- Which memory addresses are read or written, given the state of all relevant registers

Instructions should be constructed from an `unsigned char*` pointing to machine language, using the InstructionDecoder class. See **InstructionDecoder** (Section 3.4) for more details.

Public Member Functions

- **Instruction** (Operation::Ptr what, const std::vector< **Expression::Ptr** > &operand-Source, size_t size, const unsigned char *raw, unsigned int opsema)
- const **Operation** & **getOperation** () const
- void **getOperands** (std::vector< **Operand** > &operands) const
- **Operand getOperand** (int index) const
- unsigned char **rawByte** (unsigned int index) const
- const void * **ptr** () const
- void **getWriteSet** (std::set< **RegisterAST::Ptr** > &regsWritten) const
- void **getReadSet** (std::set< **RegisterAST::Ptr** > &regsRead) const
- bool **isRead** (**Expression::Ptr** candidate) const
- bool **isWritten** (**Expression::Ptr** candidate) const
- bool **readsMemory** () const
- bool **writesMemory** () const
- void **getMemoryReadOperands** (std::set< **Expression::Ptr** > &memAccessors) const
- void **getMemoryWriteOperands** (std::set< **Expression::Ptr** > &memAccessors) const
- **Expression::Ptr getControlFlowTarget** () const
- bool **allowsFallThrough** () const
- std::string **format** () const
- bool **isValid** () const
- bool **isLegalInsn** () const
- InsnCategory **getCategory** () const

Constructors & Destructors

Instruction (Operation::Ptr *what*, const std::vector< Expression::Ptr > & *operand-Source*, size_t *size*, const unsigned char * *raw*, unsigned int *opsema*)

Parameters:

what Opcode of the instruction

operandSource Contains the Expressions to be transformed into Operands

size Contains the number of bytes occupied by the corresponding machine instruction

raw Contains a pointer to the buffer from which this instruction object was decoded.

Construct an Instruction from an Operation and a collection of Expressions. This method is not intended to be used except by the InstructionDecoder class, which serves as a factory class for producing Instruction objects. While an Instruction object may be built "by hand" if desired, using the decoding interface ensures that the operation and operands are a sensible combination, and that the size reported is based on the actual size of a legal encoding of the machine instruction represented. In the course of constructing an Instruction, the Expressions in `operandSource` will be transformed to Operand objects. This transformation will map the semantic information about which operands are read and written in the Operation object `what` to the value computations in `operandSource`.

Member Functions

const Operation & getOperation () const

Returns:

The Operation used by the Instruction

See **Operation** (Section 3.2) for details of the Operation interface.

void getOperands (std::vector< Operand > & *operands*) const

The vector `operands` has the instruction's operands appended to it in the same order that they were decoded.

Operand getOperand (int *index*) const

The `getOperand` method returns the operand at position `index`, or an empty operand if `index` does not correspond to a valid operand in this instruction.

unsigned char rawByte (unsigned int *index*) const

Returns a pointer to the buffer from which this instruction was decoded.

size_t size () const

Returns the size of the corresponding machine instruction, in bytes.

const void ∗ ptr () const

Returns a pointer to the raw byte representation of the corresponding machine instruction.

void getWriteSet (std::set< RegisterAST::Ptr > & *regsWritten*) const

Parameters:

regsWritten Insert the set of registers written by the instruction into `regsWritten`.

The list of registers returned in `regsWritten` includes registers that are explicitly written as destination operands (like the destination of a move). It also includes registers that are implicitly written (like the stack pointer in a push or pop instruction). It does not include any registers used only in computing the effective address of a write. `pop ∗eax`, for example, writes to `esp`, reads `esp`, and reads `eax`, but despite the fact that ∗`eax` is the destination operand, `eax` is not itself written.

For both the write set and the read set (below), it is possible to determine whether a register is accessed implicitly or explicitly by examining the Operands. An explicitly accessed register appears as an operand that is written or read; also, any registers used in any address calculations are explicitly read. Any element of the write set or read set that is not explicitly written or read is implicitly written or read.

void getReadSet (std::set< RegisterAST::Ptr > & *regsRead*) const

Parameters:

> ***regsRead*** Insert the set of registers read by the instruction into `regsRead`.

If an operand is used to compute an effective address, the registers involved are read but not written, regardless of the effect on the operand.

bool isRead (Expression::Ptr *candidate*) const

Parameters:

> ***candidate*** Subexpression to search for among the values read by this Instruction object.

Returns true if `candidate` is read by this Instruction.

bool isWritten (Expression::Ptr *candidate*) const

Parameters:

> ***candidate*** Subexpression to search for among the values written by this Instruction object.

Returns true if `candidate` is written by this Instruction.

bool readsMemory () const

Returns:

> Returns true if the instruction reads at least one memory address as data.

If any operand containing a Dereference object is read, the instruction reads the memory at that address. Also, on platforms where a stack pop is guaranteed to read memory, `readsMemory` will return true for a pop operation.

bool writesMemory () const

Returns:

> Returns true if the instruction writes at least one memory address.

If any operand containing a Dereference object is written, the instruction writes the memory at that address. Also, on platforms where a stack push is guaranteed to write memory, `writesMemory` will return true for a push operation.

void getMemoryReadOperands (std::set< Expression::Ptr > & *memAccessors*) const

Parameters:

> ***memAccessors*** Addresses read by this instruction are inserted into `memAccessors`

The addresses read are in the form of Expressions, which may be evaluated once all of the registers that they use have had their values set. Note that this method returns ASTs representing address computations, and not address accesses. For instance, an instruction accessing memory through a register dereference would return a Expression tree containing just the register that determines the address being accessed, not a tree representing a dereference of that register.

void getMemoryWriteOperands (std::set< Expression::Ptr > & *memAccessors***) const**

Parameters:

> *memAccessors* Addresses written by this instruction are inserted into `memAccessors`

The addresses written are in the same form as those returned by `getMemoryReadOperands` above.

Expression::Ptr getControlFlowTarget () const

Returns:

> An expression evaluating to the non-fallthrough control flow targets, if any, of this instruction.

When called on an explicitly control-flow altering instruction, returns the non-fallthrough control flow destination. When called on any other instruction, returns `NULL`.

For direct absolute branch instructions, `getControlFlowTarget` will return an immediate value. For direct relative branch instructions, `getControlFlowTarget` will return the expression `PC +` offset. In the case of indirect branches and calls, it returns a dereference of a register (or possibly a dereference of a more complicated expression). In this case, data flow analysis will often allow the determination of the possible targets of the instruction. We do not do analysis beyond the single-instruction level in the Instruction API; if other code performs this type of analysis, it may update the information in the Dereference object using the setValue method in the Expression interface. More details about this may be found in **Expression** (Section 3.6) and **Dereference** (Section 3.11).

bool allowsFallThrough () const

Returns:

> False if control flow will unconditionally go to the result of `getControlFlowTarget` after executing this instruction.

std::string format () const

Returns:

> The instruction as a string of assembly language

`format` is principally a helper function; Instructions are meant to be written to output streams via `operator<<`. `format` is included in the public interface for diagnostic purposes.

bool isValid () const

Returns true if this Instruction object is valid. Invalid instructions indicate that an Instruction-Decoder has reached the end of its assigned range, and that decoding should terminate.

bool isLegalInsn () const

Returns true if this Instruction object represents a legal instruction, as specified by the architecture used to decode this instruction.

InsnCategory getCategory () const

ALPHA: Returns the category that an instruction falls into. This feature is presently incomplete, and we welcome feedback on ways to extend it usefully.

Currently, the valid categories are c_CallInsn, c_ReturnInsn, c_BranchInsn, c_CompareInsn, and c_NoCategory, as defined in InstructionCategories.h.

3.2 Operation Class

An Operation object represents a family of opcodes (operation encodings) that perform the same task (e.g. the MOV family). It includes information about the number of operands, their read/write semantics, the implicit register reads and writes, and the control flow behavior of a particular assembly language operation. It additionally provides access to the assembly mnemonic, which allows any semantic details that are not encoded in the Instruction representation to be added by higher layers of analysis.

As an example, the CMP operation on IA32/AMD64 processors has the following properties:

- Operand 1 is read, but not written

- Operand 2 is read, but not written

- The following flags are written:
    - Overflow
    - Sign
    - Zero
    - Parity
    - Carry
    - Auxiliary

- No other registers are read, and no implicit memory operations are performed

Operations are constructed by the InstructionDecoder as part of the process of constructing an Instruction.

Public Member Functions

- std::string **format** () const
- entryID **getID** () const

Member Functions

const Operation::registerSet & implicitReads () const

Returns the set of registers implicitly read (i.e. those not included in the operands, but read anyway).

const Operation::registerSet & implicitWrites () const

Returns the set of registers implicitly written (i.e. those not included in the operands, but written anyway).

std::string format () const

Returns the mnemonic for the operation. Like `instruction::format`, this is exposed for debugging and will be replaced with stream operators in the public interface.

entryID getID () const

Returns the entry ID corresponding to this operation. Entry IDs are enumerated values that correspond to assembly mnemonics.

bool isRead (Expression::Ptr *candidate*) const

Returns true if the expression represented by `candidate` is read implicitly.

bool isWritten (Expression::Ptr *candidate*) const

Returns true if the expression represented by `candidate` is written implicitly.

const Operation::VCSet & getImplicitMemReads () const

Returns the set of memory locations implicitly read.

const Operation::VCSet & getImplicitMemWrites () const

Returns the set of memory locations implicitly written.

3.3   Operand Class

An Operand object contains an AST built from RegisterAST and Immediate leaves, and information about whether the Operand is read, written, or both. This allows us to determine which of the registers that appear in the Operand are read and which are written, as well as whether any memory accesses are reads, writes, or both. An Operand, given full knowledge of the values of the leaves of the AST, and knowledge of the logic associated with the tree's internal nodes, can determine the result of any computations that are encoded in it. It will rarely be the case that an Instruction is built with its Operands' state fully specified. This mechanism is instead intended to allow a user to fill in knowledge about the state of the processor at the time the Instruction is executed.

Public Member Functions

- **Operand** (**Expression::Ptr** val, bool read, bool written)
- void **getReadSet** (std::set< **RegisterAST::Ptr** > &regsRead) const
- void **getWriteSet** (std::set< **RegisterAST::Ptr** > &regsWritten) const
- void **addEffectiveReadAddresses** (std::set< **Expression::Ptr** > &memAccessors) const
- void **addEffectiveWriteAddresses** (std::set< **Expression::Ptr** > &memAccessors) const
- std::string **format** () const

Constructors & Destructors

Operand (Expression::Ptr *val*, bool *read*, bool *written*)

Create an operand from a Expression and flags describing whether the ValueComputation is read, written or both.

Parameters:

val Reference-counted pointer to the Expression that will be contained in the Operand being constructed

read True if this operand is read

written True if this operand is written

Member Functions

void getReadSet (std::set< RegisterAST::Ptr > & *regsRead*) const

Get the registers read by this operand.

Parameters:

regsRead Has the registers read inserted into it

void getWriteSet (std::set< RegisterAST::Ptr > & *regsWritten*) const

Get the registers written by this operand.

Parameters:

regsWritten Has the registers written inserted into it

bool isRead (Expression::Ptr *candidate*) const

Returns true if this operand is read.

bool isWritten (Expression::Ptr *candidate*) const

Returns true if this operand is written.

bool readsMemory () const

Returns true if this operand reads memory.

bool writesMemory () const

Returns true if this operand writes memory.

void addEffectiveReadAddresses (std::set< Expression::Ptr > & *memAccessors*) const

Inserts the effective addresses read by this operand into memAccessors.

Parameters:

> ***memAccessors*** If this is a memory read operand, insert the `ExpressionPtr` representing the address being read into `memAccessors`.

void addEffectiveWriteAddresses (std::set< Expression::Ptr > & *memAccessors*) const

Inserts the effective addresses written by this operand into memAccessors.

Parameters:

> ***memAccessors*** If this is a memory write operand, insert the `ExpressionPtr` representing the address being written into `memAccessors`.

std::string format () const

Return a printable string representation of the operand.

Returns:

> The operand in a disassembly format

Expression::Ptr getValue () const

The `getValue` method returns an ExpressionPtr to the AST contained by the operand.

3.4   InstructionDecoder Class

The InstructionDecoder class decodes instructions, given a buffer of bytes and a length, and constructs an Instruction. The InstructionDecoder will, by default, be constructed to decode machine language on the platform on which it has been compiled. The buffer will be treated as if there is an instruction stream starting at the beginning of the buffer. InstructionDecoder objects are given a buffer from which to decode at construction. Calls to `decode` will proceed to decode instructions sequentially from that buffer until its end is reached. At that point, all subsequent calls to `decode` will return an invalid Instruction object.

An InstructionDecoder object may alternately be constructed without designating a buffer, and the buffer may be specified at the time `decode` is called. This method of use may be more convenient for users who are decoding non-contiguous instructions.

Public Member Functions

- Instruction::Ptr **decode** ()
- Instruction::Ptr **decode** (const unsigned char *buffer)

12

Constructors & Destructors

InstructionDecoder (const unsigned char ∗ *buffer*, size_t *size*)

Construct an InstructionDecoder object that decodes from `buffer`, up to `size` bytes.

InstructionDecoder ()

Construct an InstructionDecoder object with no buffer specified.

Member Functions

Instruction::Ptr decode ()

Decode the current instruction in this InstructionDecoder object's buffer, interpreting it as machine language of the type understood by this InstructionDecoder. If the buffer does not contain a valid instruction stream, a null Instruction pointer will be returned. The Instruction's `size` field will contain the size of the instruction decoded.

Instruction::Ptr decode (const unsigned char ∗ *buffer*)

Decode the instruction at `buffer`, interpreting it as machine language of the type understood by this InstructionDecoder. If the buffer does not contain a valid instruction stream, a null Instruction pointer will be returned. The Instruction's `size` field will contain the size of the instruction decoded.

3.5 InstructionAST Class

The InstructionAST class is the base class for all nodes in the ASTs used by the Operand class. It defines the necessary interfaces for traversing and searching an abstract syntax tree representing an operand. For the purposes of searching an InstructionAST, we provide two related interfaces. The first, `getUses`, will return the registers that appear in a given tree. The second, `isUsed`, will take as input another tree and return true if that tree is a (not necessarily proper) subtree of this one. `isUsed` requires us to define an equality relation on these abstract syntax trees, and the equality operator is provided by the InstructionAST, with the details implemented by the classes derived from InstructionAST. Two AST nodes are equal if the following conditions hold:

- They are of the same type
- If leaf nodes, they represent the same immediate value or the same register
- If non-leaf nodes, they represent the same operation and their corresponding children are equal

Public Member Functions

- bool **operator==** (const **InstructionAST** &rhs) const
- virtual void **getUses** (set< InstructionAST::Ptr > &uses)=0
- virtual bool **isUsed** (InstructionAST::Ptr findMe) const =0
- virtual std::string **format** () const =0

Member Functions

bool operator== (const InstructionAST & *rhs*) const

Compare two AST nodes for equality.

Non-leaf nodes are equal if they are of the same type and their children are equal. RegisterASTs are equal if they represent the same register. Immediates are equal if they represent the same value.

virtual void getChildren (vector< InstructionAST::Ptr > & *children*) const [pure virtual]

Children of this node are appended to the vector `children`.

virtual void getUses (set< InstructionAST::Ptr > & *uses*) [pure virtual]

Parameters:

 uses The use set of this node is appended to the vector `uses`

The use set of an InstructionAST is defined as follows:

- A RegisterAST uses itself

- A BinaryFunction uses the use sets of its children

- An Immediate uses nothing

- A Dereference uses the use set of its child

virtual bool isUsed (InstructionAST::Ptr *findMe*) const [pure virtual]

Returns:

 True if `findMe` is used by this AST node.

Parameters:

 findMe AST node to find in the use set of this node

Unlike `getUses`, `isUsed` looks for `findMe` as a subtree of the current tree. `getUses` is designed to return a minimal set of registers used in this tree, whereas `isUsed` is designed to allow searches for arbitrary subexpressions

virtual std::string format () const [pure virtual]

The `format` interface returns the contents of an InstructionAST object as a string. By default, `format()` (Section 3.5) produces assembly language.

3.6   Expression Class

An Expression is an AST representation of how the value of an operand is computed.

The Expression class extends the InstructionAST class by adding the concept of evaluation to the nodes of an InstructionAST. Evaluation attempts to determine the **Result** (Section 3.7) of the computation that the AST being evaluated represents. It will fill in results of as many of the nodes in the tree as possible, and if full evaluation is possible, it will return the result of the computation performed by the tree.

Permissible leaf nodes of a Expression tree are RegisterAST and Immediate objects. Permissible internal nodes are BinaryFunction and Dereference objects. An Expression may represent an immediate value, the contents of a register, or the contents of memory at a given address, interpreted as a particular type.

The Results in an Expression tree contain a type and a value. Their values may be an undefined value or an instance of their associated type. When two Results are combined using a BinaryFunction, the BinaryFunction specifies the output type. Sign extension, type promotion, truncation, and all other necessary conversions are handled automatically based on the input types and the output type. If both of the Results that are combined have defined values, the combination will also have a defined value; otherwise, the combination's value will be undefined. For more information, see **Result** (Section 3.7), **BinaryFunction** (Section 3.10), and **Dereference** (Section 3.11).

A user may specify the result of evaluating a given Expression. This mechanism is designed to allow the user to provide a Dereference or RegisterAST with information about the state of memory or registers. It may additionally be used to change the value of an Immediate or to specify the result of a BinaryFunction. This mechanism may be used to support other advanced analyses.

In order to make it more convenient to specify the results of particular subexpressions, the `bind` method is provided. `bind` allows the user to specify that a given subexpression has a particular value everywhere that it appears in an expression. For example, if the state of certain registers is known at the time an instruction is executed, a user can `bind` those registers to their known values throughout an Expression.

The evaluation mechanism, as mentioned above, will evaluate as many sub-expressions of an expression as possible. Any operand that is more complicated than a single immediate value, however, will depend on register or memory values. The Results of evaluating each subexpression are cached automatically using the `setValue` mechanism. The Expression then attempts to determine its Result based on the Results of its children. If this Result can be determined (most likely because register contents have been filled in via `setValue` or `bind`), it will be returned from `eval`; if it can not be determined, a Result with an undefined value will be returned. See Figure 6 for an illustration of this concept; the operand represented is [`EBX` + 4 ∗ `EAX`]. The contents of `EBX` and `EAX` have been determined through some outside mechanism, and have been defined with `setValue`. The `eval` mechanism proceeds to determine the address being read by the Dereference, since this information can be determined given the contents of the registers. This address is available from the Dereference through its child in the tree, even though calling `eval` on the Dereference returns a Result with an undefined value.
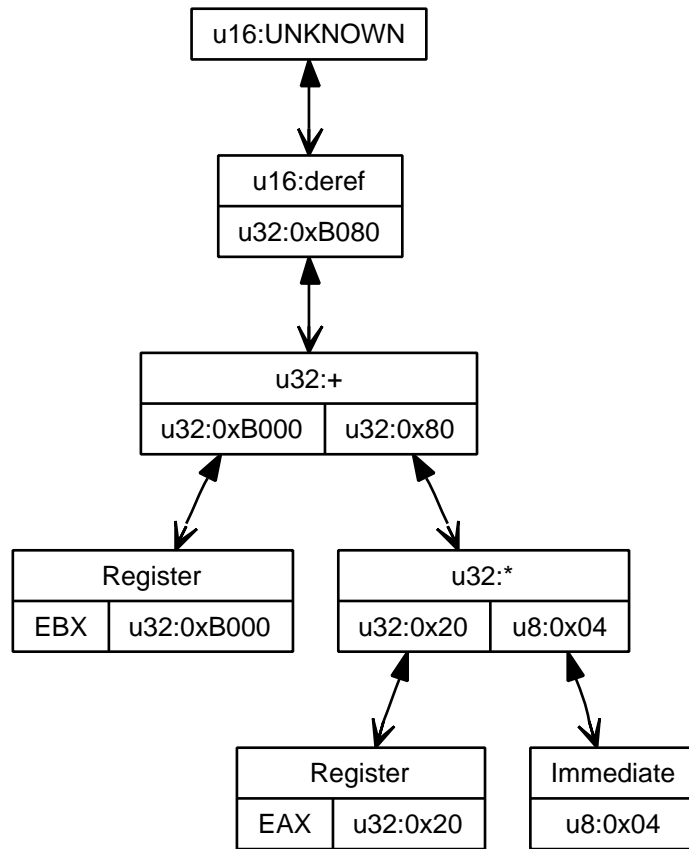
Figure 6: Applying `eval` to a Dereference tree with

the state of the registers known and the state of memory unknown"

Public Member Functions

- void **setValue** (const **Result** &knownValue)
- void **clearValue** ()
- virtual bool **bind** (**Expression** ∗expr, const **Result** &value)

Member Typedefs

typedef dyn_detail::boost::shared_ptr<Expression> Ptr

A type definition for a reference counted pointer to a Expression.

Member Functions

const Result & eval () const [virtual]

If the Expression can be evaluated, returns a Result containing its value. Otherwise returns an undefined Result.

void setValue (const Result & *knownValue***)**

Parameters:

>   *knownValue* Sets the result of `eval` for this Expression to `knownValue`

void clearValue ()

`clearValue` sets the contents of this Expression to undefined. The next time `eval` is called, it will recalculate the value of the Expression.

int size () const

`size` returns the size of this Expression's Result, in bytes.

bool bind (Expression ∗ *expr***, const Result &** *value***)** [virtual]

`bind` searches for all instances of the Expression `expr` within this Expression, and sets the result of `eval` for those subexpressions to `value`. `bind` returns true if at least one instance of `expr` was found in this Expression.

`bind` does not operate on subexpressions that happen to evaluate to the same value. For example, if a dereference of 0xDEADBEEF is bound to 0, and a register is bound to 0xDEADBEEF, a dereference of that register is not bound to 0.

3.7   Result Class

- `m14`: a 14 byte memory value

Public Member Functions

- **Result** (Result_Type t)
- template<typename T> **Result** (Result_Type t, T v)
- bool **operator==** (const **Result** &o) const
- std::string **format** () const

Constructors & Destructors

Result (Result_Type *t***)**

A Result may be constructed from a type without providing a value. This constructor creates a Result of type `t` with undefined contents.

Result (Result_Type *t***, T** *v***)**

A Result may be constructed from a type and any value convertible to the type that the tag represents. This constructor creates a Result of type `t` and contents `v` for any `v` that is implicitly convertible to type `t`. Attempting to construct a Result with a value that is incompatible with its type will result in a compile-time error.

Member Functions

bool operator== (const Result & *o*) const

Two Results are equal if any of the following hold:

- Both Results are of the same type and undefined

- Both Results are of the same type, defined, and have the same value

Otherwise, they are unequal (due to having different types, an undefined Result compared to a defined Result, or different values).

std::string format () const

Results are formatted as strings containing their contents, represented as hexadecimal. The type of the Result is not included in the output.

int size () const

Returns the size of the contained type, in bytes.

3.8   RegisterAST Class

A RegisterAST object represents a register contained in an operand. As a RegisterAST is a Expression, it may contain the physical register's contents if they are known.

Public Member Functions

- virtual void **getChildren** (vector< InstructionAST::Ptr > &children) const
- virtual void **getUses** (set< InstructionAST::Ptr > &uses)
- virtual bool **isUsed** (InstructionAST::Ptr findMe) const
- bool **operator<** (const **RegisterAST** &rhs) const

Static Public Member Functions

- static **RegisterAST makePC** ()
- static **RegisterAST::Ptr promote** (const InstructionAST::Ptr reg)

Member Typedefs

typedef dyn_detail::boost::shared_ptr<RegisterAST> Ptr

A type definition for a reference-counted pointer to a RegisterAST.

Constructors & Destructors

RegisterAST (int *id*)

Construct a register, assigning it the ID id.

Member Functions

void getChildren (vector< InstructionAST::Ptr > & *children***) const** `[virtual]`

By definition, a RegisterAST object has no children.

Parameters:

> ***children*** Since a RegisterAST has no children, the `children` parameter is unchanged by this method.

void getUses (set< InstructionAST::Ptr > & *uses***)** `[virtual]`

By definition, the use set of a RegisterAST object is itself.

Parameters:

> ***uses*** This RegisterAST will be inserted into `uses`.

bool isUsed (InstructionAST::Ptr *findMe***) const** `[virtual]`

`isUsed` returns true if `findMe` is a RegisterAST that represents the same register as this RegisterAST, and false otherwise.

std::string format () const `[virtual]`

The `format` method on a RegisterAST object returns the name associated with its ID.

RegisterAST makePC () `[static]`

Utility function to get a Register object that represents the program counter.

`makePC` is provided to support platform-independent control flow analysis.

bool operator< (const RegisterAST & *rhs***) const**

We define a partial ordering on registers by their register number so that they may be placed into sets or other sorted containers.

unsigned int getID () const

The `getID` function returns the ID number of a register.

RegisterAST::Ptr promote (const InstructionAST::Ptr *reg***)** `[static]`

Utility function to hide aliasing complexity on platforms (IA-32) that allow addressing part or all of a register

3.9 Immediate Class

The Immediate class represents an immediate value in an operand

Since an Immediate represents a constant value, the `setValue` and `clearValue` interface are disabled on Immediate objects. If an immediate value is being modified, a new Immediate object should be created to represent the new value.

Public Member Functions

- virtual bool **isUsed** (InstructionAST::Ptr findMe) const

Member Functions

void getChildren (vector< InstructionAST::Ptr > &) const [virtual]

By definition, an Immediate has no children.

virtual void getUses (set< InstructionAST::Ptr > &) [virtual]

By definition, an Immediate uses no registers.

bool isUsed (InstructionAST::Ptr *findMe*) const [virtual]

`isUsed`, when called on an Immediate, will return true if `findMe` represents an Immediate with the same value. While this convention may seem arbitrary, it allows `isUsed` to follow a natural rule: an InstructionAST is used by another InstructionAST if and only if the first InstructionAST is a subtree of the second one.

3.10   BinaryFunction Class

A BinaryFunction object represents a function that can combine two Expressions and produce another ValueComputation.

For the purposes of representing a single operand of an instruction, the BinaryFunctions of interest are addition and multiplication of integer values; this allows a Expression to represent all addressing modes on the architectures currently supported by the Instruction API.

Public Member Functions

- **BinaryFunction (Expression::Ptr** arg1, **Expression::Ptr** arg2, Result_Type result_-type, funcT::Ptr func)
- virtual const **Result** & **eval** () const
- virtual void **getChildren** (vector< InstructionAST::Ptr > &children) const
- virtual void **getUses** (set< InstructionAST::Ptr > &uses)
- virtual bool **isUsed** (InstructionAST::Ptr findMe) const

Constructors & Destructors

BinaryFunction (Expression::Ptr *arg1*, Expression::Ptr *arg2*, Result_Type *result_ -type*, funcT::Ptr *func*)

Parameters:

    arg1 first input to function

    arg2 second input to function

    result_ type type of the function's result

    func implementation of the function

The constructor for a BinaryFunction may take a reference-counted pointer or a plain C++ pointer to each of the child Expressions that represent its arguments. Since the reference-counted implementation requires explicit construction, we provide overloads for all four combinations of plain and reference-counted pointers. Note that regardless of which constructor is used, the pointers `arg1` and `arg2` become owned by the BinaryFunction being constructed, and should not be deleted. They will be cleaned up when the BinaryFunction object is destroyed.

The `func` parameter is a binary functor on two Results. It should be derived from `funcT`. `addResult` and `multResult`, which respectively add and multiply two Results, are provided as part of the InstructionAPI, as they are necessary for representing address calculations. Other `funcTs` may be implemented by the user if desired. funcTs have names associated with them for output and debugging purposes. The addition and multiplication functors provided with the Instruction API are named "+" and "*", respectively.

Member Functions

const Result & eval () const   [virtual]

The BinaryFunction version of `eval` allows the `eval` mechanism to handle complex addressing modes. Like all of the ValueComputation implementations, a BinaryFunction's `eval` will return the result of evaluating the expression it represents if possible, or an empty Result otherwise. A BinaryFunction may have arguments that can be evaluated, or arguments that cannot. Additionally, it may have a real function pointer, or it may have a null function pointer. If the arguments can be evaluated and the function pointer is real, a result other than an empty Result is guaranteed to be returned. This result is cached after its initial calculation; the caching mechanism also allows outside information to override the results of the BinaryFunction's internal computation. If the cached result exists, it is guaranteed to be returned even if the arguments or the function are not evaluable.

virtual void getChildren (vector< InstructionAST::Ptr > & *children*) const   [virtual]

The children of a BinaryFunction are its two arguments.

Parameters:

> ***children*** Appends the children of this BinaryFunction to `children`.

virtual void getUses (set< InstructionAST::Ptr > & *uses*)   [virtual]

The use set of a BinaryFunction is the union of the use sets of its children.

Parameters:

> ***uses*** Appends the use set of this BinaryFunction to `uses`.

virtual bool isUsed (InstructionAST::Ptr *findMe*) const   [virtual]

`isUsed` returns true if `findMe` is an argument of this BinaryFunction, or if it is in the use set of either argument.

3.11   Dereference Class

A Dereference object is a Expression that dereferences another ValueComputation.

A Dereference contains an Expression representing an effective address computation. Its use set is the same as the use set of the Expression being dereferenced.

It is not possible, given the information in a single instruction, to evaluate the result of a dereference. `eval` may still be called on a Expression that includes dereferences, but the expected use case is as follows:

- Determine the address being used in a dereference via the `eval` mechanism

- Perform analysis to determine the contents of that address

- If necessary, fill in the Dereference node with the contents of that addresss, using `setValue`

The type associated with a Dereference node will be the type of the value *read from memory*, not the type used for the address computation. Two Dereferences that access the same address but interpret the contents of that memory as different types will produce different values. The children of a Dereference at a given address are identical, regardless of the type of dereference being performed at that address. For example, the Expression shown in Figure 6 could have its root Dereference, which interprets the memory being dereferenced as a unsigned 16-bit integer, replaced with a Dereference that interprets the memory being dereferenced as any other type. The remainder of the Expression tree would, however, remain unchanged.

Public Member Functions

- **Dereference (Expression::Ptr** addr, Result_Type result_type)
- virtual void **getChildren** (vector< InstructionAST::Ptr > &children) const
- virtual void **getUses** (set< InstructionAST::Ptr > &uses)
- virtual bool **isUsed** (InstructionAST::Ptr findMe) const

Constructors & Destructors

Dereference (Expression::Ptr *addr*, Result_Type *result_ type*)

A Dereference is constructed from a Expression pointer (raw or shared) representing the address to be dereferenced and a type indicating how the memory at the address in question is to be interpreted.

Member Functions

virtual void getChildren (vector< InstructionAST::Ptr > & *children*) const   [virtual]

A Dereference has one child, which represents the address being dereferenced.

Parameters:

   children Appends the child of this Dereference to `children`.

virtual void getUses (set< InstructionAST::Ptr > & *uses*)   [virtual]

The use set of a Dereference is the same as the use set of its children.

Parameters:

   uses The use set of this Dereference is inserted into `uses`.

virtual bool isUsed (InstructionAST::Ptr *findMe*) const `[virtual]`

An InstructionAST is used by a Dereference if it is equivalent to the Dereference or it is used by the lone child of the Dereference

3.12   RegInfo Struct

3.13   IA32RegTable Struct

Index