Canal

a static code analysis tool

Karel Klíč

August 20, 2012

Contents

Chapter 1

Overview

For a sufficiently complex software system, its maintainability and extensibility is limited by our ability to understand and correctly approximate the behaviour of the system, trace the impact of system parts to each other, control the impact of modifications, ensure correctness of the critical parts, and fixing bugs before they cause serious consequences in production.

The maintainability and extensibility is affected by the programming language of implementation. Efficient low-level languages such as C and C++ increase the complexity of the system by being closely aligned with hardware. Systems must handle memory management, operate with machine-dependent integers and floating point numbers, and use system calls with complex invariants and interdependencies.

Canal is a framework combining existing static analysis techniques improving maintainability, understanding, traceability and correctness in a coherent manner. The purpose of the framework is to make existing techniques accessible and evaluable, and to support the implementation of new techniques and allowing high quality experiments. Currently, techniques are often presented without proper experiments on real-world complex systems, or just with a proprietary implementation that cannot be investigated, so actual applicability and impact of many techniques is unknown. We hope to change this situation.

The implementation can be used for a static analysis of real-world complex software systems written in efficient low-level languages such as C and C++.

1.1 Use cases

1.1.1 Analysis of program behaviour

You can hook on the fixpoint of function calls to inspect the calculated abstract values. You can get abstract values of function call parameters.

1.1.2 Comparison with a specification

A set of pre- and post-conditions for functions, and variable-based or module-based automata. This can be defined for certain function or library, and library/function users are watched to conform to the specification.

1.1.3 Conformance to environment constraints

Double free, memory leaks, buffer overflow and underflow, division by zero, invalid access to memory, locking and concurrency errors, uncaught exceptions.

1.2 Artifacts

1.2.1 Source code

1.2.2 Specification

1.2.3 Models

Chapter 2

Installation

2.1 Installation from source code on Red Hat Enterprise Linux 6

Canal can be built and installed on Red Hat Enterprise Linux 6, but some additional packages must be installed from third-party repositories prior to building it. Documentation can be build on Red Hat Eneterprise Linux 6 as well.

Packages required by the build process:

- The core library requires `llvm-devel` and `clang` packages, which can be obtained from Extra Packages for Enterprise Linux (or EPEL) software repository.
- The command-line user interface tool requires `elfutils-devel`and `readline-devel` packages.
- The documentation requires `doxygen`, `texlive-collection-basic`, `texlive-import`, `texlive-preprint`, `texlive-fullpage`, `texlive-a4wide`, `texlive-epstopdf` packages. The `texlive-*` packages can be obtained from Jindřich Nový's TeXLive software repository, which is available at `http://jnovy.fedorapeople.org/texlive/packages.el6/`.

The installed program requires `llvm`, `clang`, `elfutils`, `readline` packages.

Part I

Concepts

Chapter 3

Preliminaries

Definitions from the order theory. More details can be found in [4].

A binary relation $\sqsubseteq$ is *reflexive* on a set $\mathcal{D}$ if every element is related to itself: $a \sqsubseteq a$ for all $a \in \mathcal{D}$. A binary relation $\sqsubseteq$ is *antisymmetric* on a set $\mathcal{D}$ if the following implication holds: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$. A binary relation $\sqsubseteq$ is *transitive* on a set $\mathcal{D}$ if whenever an element a is related to an element b, and b is in turn related to an element c, then a is also related to c: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$.

A *partial order* $\sqsubseteq$ is a binary relation on a set $\mathcal{D}$ which is reflexive, antisymmetric and transitive. A *partial ordered set* or *poset* for short is an ordered pair $(\mathcal{D}, \sqsubseteq)$ of a set $\mathcal{D}$ together with a partial ordering $\sqsubseteq$.

An element a in a poset $(\mathcal{D}, \sqsubseteq)$ is called *maximal* if it is not less than any other element in $\mathcal{D}$: $\nexists b \in \mathcal{D}, a \sqsubset b$. If there is an unique maximal element, we call it the *greatest element* and denote it by $\top$. Similarly, an element a in a poset $(\mathcal{D}, \sqsubseteq)$ is called *minimal* if it is not greater than any other element in $\mathcal{D}$: $\nexists b \in \mathcal{D}, b \sqsubset a$. If there is an unique minimal element, we call it the *least element* and denote it by $\bot$.

Let $(\mathcal{D}, \sqsubseteq)$ be a poset and $A \subseteq \mathcal{D}$. An element $u \in \mathcal{D}$ is an *upper bound* of A if $a \sqsubseteq u$ for all elements $a \in A$. The *least upper bound* or *lub* for short is an element x that is an upper bound on a subset A and is less than all other upper bounds on A; such an element is denoted by $\bigsqcup A$. Similarly, an element $l \in \mathcal{D}$ is a *lower bound* of A if $l \sqsubseteq a$ for all elements $a \in A$. The *greatest lower bound* or *glb* for short is an element x that is a lower bound on a subset A and is greater than all other lower bounds on A; such an element is denoted by $\bigsqcap A$.

A *lattice* $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set in which any two elements $a, b \in \mathcal{D}$ have both a least upper bound, denoted by $a \sqcup b$, and a greatest lower bound, denoted by $a \sqcap b$. A *complete lattice* $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partially ordered set in which every subset $A \subseteq \mathcal{D}$ has a least upper bound and a greatest lower bound.

A *fixpoint* of a function F is an element X such that $F(X) = X$.

A function $F \in \mathcal{D}_1 \to \mathcal{D}_2$ between two posets $(\mathcal{D}_1, \sqsubseteq_1)$ and $(\mathcal{D}_2, \sqsubseteq_2)$ is *monotonic* if $X \sqsubseteq_1 Y \implies F(X) \sqsubseteq_2 F(Y)$. A function $F \in \mathcal{D}_1 \to \mathcal{D}_2$ is *strict* if $F(\bot_1) = \bot_2$. A function $F \in \mathcal{D}_1 \to \mathcal{D}_2$ is *continuous* if it preserves the existing limits of increasing chains $(X_i)_{i \in I}$: $F(\bigsqcup_1 \{X_i \mid i \in I\}) = \bigsqcup_2 \{F(X_i) \mid i \in I\}$ whenever $\bigsqcup_1 \{X_i \mid i \in I\}$ exists.

Chapter 4

LLVM

LLVM was first presented in [5].

Formalized in [9].

Chapter 5

Abstract Interpretation

Our abstract interpreter comes in four flavours:

Context-insensitive flow-insensitive For every function in a program, the fixpoint is calculated with a single set of abstract values that encompasses all function calls.

Context-insensitive flow-sensitive For every function in a program, the fixpoint is calculated with a single set of abstract values that encompasses all function calls, but every possible path through the function is calculated separately.

Context-sensitive flow-insensitive The fixpoint is calculated with a set of abstract values specifically created for every function call.

Context-sensitive flow-sensitive The fixpoint is calculated with a set of abstract values specifically created for every possible path in a function call. Path conditions are taken into account.

Abstract interpreter can be either operational or equation-based. Our interpreter is operational.

5.0.1 Tuning

The precision of abstract interpreter is greately tunable. Here are the aspects to consider:

Interpreter flavour Context-sensitivity and flow-sensitivity increase both precision and complexity.

Widening and narrowing Selection and parameters of widening and narrowing operators affect both precision and complexity.

Relations in abstract domains Type and number of relations in abstract domains affect both precision and complexity.

Memory for abstract domains Parameters of some abstract domains allow to trade memory for better precision.

Maximal precision of abstract interpreter is same as for symbolic executor, but abstract interpreter is more tunable.

5.0.2 Context sensitivity

Context sensitivity is achieved by keeping a function call stack. Every stack frame keeps the complete state of a function fixpoint calculation (all local and global variables). When a function call is reached during the fixpoint computation and function call parameters are already initialized, a new frame is placed on the top of stack and the called function is interpreted with the provided parameters.

5.0.3 Flow sensitivity

Chapter 6

Abstractions

6.1 Multi threading

Multi-threading abstraction for Abstract Interpretation appeared in [7].

6.2 Memory

Memory abstraction appeared in [6].

Our memory abstraction for abstract interpretation recognizes four kinds of memory:

Register-like stack memory This is function-level memory that is released automatically when function returns. We denote such a memory by LLVM-style names starting with the percent sign `%`. Memory either has a name (e.g. `%result`) or a number is generated to serve as a name (e.g. `%32` denotes thirty-second unnamed instruction call in a function).

Stack memory allocated by `alloca` This is also a function-level memory that is released automatically when function returns. The difference to register-like stack memory is that this memory is accessed by LLVM exclusively via pointers. We denote such a memory by names starting with `%^`. Every piece of memory has a name corresponding to the place where the memory has been allocated (`alloca` has been called). So if the memory has been allocated by an instruction call `%ptr = alloca i32, align 4`, it can be denoted by `%^ptr`.

Global variables Global variables are module-wise and are valid for the whole program run. We denote such a memory by LLVM-style names starting with `@`.

Heap memory Heap memory is also valid for the whole program run. We denote such a memory by names starting by `@^`. Every piece of memory has a name corresponding to the place where the memory has been allocated (`malloc` or similar function has been called). Name of the function is also included in the place name, so if a function `createString` contains an instruction call `%result = call i8* @malloc(i32 1)`, we can denote the memory allocated on this place by `@^createString:result`.

As it can be seen from the style of memory denotation, every piece of memory is associated with a place in the program. This means all operations affecting a memory block allocated at certain place forms a single abstract value. Context-sensite abstract interpretation helps to increase the precision of this memory abstraction.

6.3 Arrays

6.4 Structures

6.5 Integers

Precise machine integer abstraction appeared in [8].

6.6 Floating-point numbers

Precise machine floating-point abstraction appeared in [8].

6.7 Pointers

Pointer can be casted to a number via the `ptrtoint` instruction. Usually, the resulting memory offset is used to achieve pointer arithmetics that are not available via `getelementptr` semantics.

Chapter 7

Wishlist

Lazy model-checking abstract value Allow to investigate just a single function, taking into account all possible parameter values and shapes (perhaps limited by a pre-condition). Parameter values and shapes must be smartly provided depending on the boundary requirements of the checked code. This allows a kind of model checking, and use of model checking algorithms and ideas.

Widening operators Implement widening operators for integers and other abstract domains as required.

Narrowing operators Implement narrowing operators for integers and other abstract domains as required.

String abstractions Implement abstract domains specific for C strings.

Weakly relational numeric abstractions Implement weakly relational integer and floating-point abstract domains.

Basic block abstraction Implement basic block summaries that speed-up the static analysis.

Function abstraction Implement function summaries that speed-up the static analysis.

Parallelization Make abstract interpreter to use multiple threads for fixpoint calculation on the right level.

Concurrency check Add support for checking of multi-threaded programs.

Part II

Implementation

Chapter 8

Overview

Canal is implemented in the C++ language as defined in the C++98 standard (ISO/IEC 14882:1998). It uses the C++ standard library as well as some additional libraries:

- LLVM core libraries. All versions from 2.8 up to 3.1 are supported.
- Clang compiler. All versions from 2.8 up to 3.1 are supported.
- GNU readline or any of its BSD-licensed reimplementation.
- elfutils; this library is optional, and used only on Linux-based operating systems

Chapter 9

Library Class Index

9.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

9.2 Class List

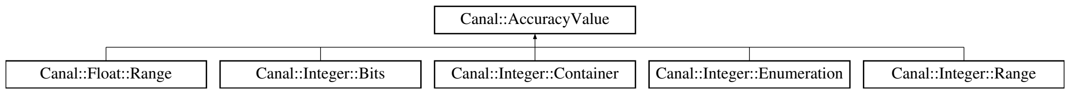
Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 10

Library Class Documentation

10.1 Canal::AccuracyValue Class Reference

Base class for abstract states that can inform about accuracy.
`#include <Value.h>`Inheritance diagram for Canal::AccuracyValue::



Public Member Functions

- virtual float accuracy () const
- virtual bool isBottom () const

  Is it the lowest value.

- virtual void setBottom ()

  Set to the lowest value.

- virtual bool isTop () const

  Is it the highest value.

- virtual void setTop ()

  Set it to the top value of lattice.

10.1.1 Detailed Description

Base class for abstract states that can inform about accuracy.

10.1.2 Member Function Documentation

float Canal::AccuracyValue::accuracy () const `[virtual]`

Get accuracy of the abstract value (0 - 1). In finite-height lattices, it is determined by the position of the value in the lattice.

Accuracy 0 means that the value represents all possible values (top). Accuracy 1 means that the value represents the most precise and exact value (bottom).

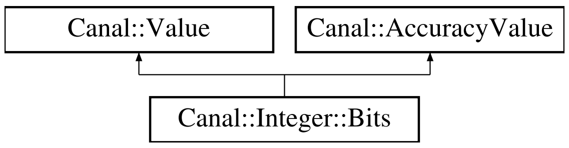
Reimplemented in Canal::Float::Range, Canal::Integer::Bits, Canal::Integer::Container, Canal::Integer::Enumeration, and Canal::Integer::Range.

The documentation for this class was generated from the following files:

- lib/Value.h
- lib/Value.cpp

10.2 Canal::Integer::Bits Class Reference

`#include <IntegerBits.h>`Inheritance diagram for Canal::Integer::Bits::



Public Member Functions

- Bits (unsigned numBits)

  Initializes to the lowest value.

- Bits (const llvm::APInt &number)

  Initializes to the given value.

- unsigned getBitWidth () const

  Return the number of bits of the represented number.

- int getBitValue (unsigned pos) const
- void setBitValue (unsigned pos, int value)
- bool signedMin (llvm::APInt &result) const
- bool signedMax (llvm::APInt &result) const
- bool unsignedMin (llvm::APInt &result) const
- bool unsignedMax (llvm::APInt &result) const
- virtual Bits ∗ clone () const
- virtual Bits ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().

- virtual void merge (const Value &value)

  Implementation of Value::merge().

- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().

- virtual std::string toString () const

  Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const

  Implementation of Value::matchesString().

- virtual void add (const Value &a, const Value &b)

  Implementation of Value::add().

- virtual void sub (const Value &a, const Value &b)

  Implementation of Value::sub().

- virtual void mul (const Value &a, const Value &b)

  Implementation of Value::mul().

- virtual void udiv (const Value &a, const Value &b)
  Implementation of Value::udiv().
- virtual void sdiv (const Value &a, const Value &b)
  Implementation of Value::sdiv().
- virtual void urem (const Value &a, const Value &b)
  Implementation of Value::urem().
- virtual void srem (const Value &a, const Value &b)
  Implementation of Value::srem().
- virtual void shl (const Value &a, const Value &b)
  Implementation of Value::shl().
- virtual void lshr (const Value &a, const Value &b)
  Implementation of Value::lshr().
- virtual void ashr (const Value &a, const Value &b)
  Implementation of Value::ashr().
- virtual void and_ (const Value &a, const Value &b)
  Implementation of Value::and_().
- virtual void or_ (const Value &a, const Value &b)
  Implementation of Value::or_().
- virtual void xor_ (const Value &a, const Value &b)
  Implementation of Value::xor_().
- virtual void icmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
  Implementation of Value::icmp().
- virtual float accuracy () const
  Implementation of AccuracyValue::accuracy().
- virtual bool isBottom () const
  Implementation of AccuracyValue::isBottom().
- virtual void setBottom ()
  Implementation of AccuracyValue::setBottom().
- virtual bool isTop () const
  Implementation of AccuracyValue::isTop().
- virtual void setTop ()
  Implementation of AccuracyValue::setTop().

Public Attributes

- llvm::APInt mBits0
- llvm::APInt mBits1

10.2.1 Detailed Description

Abstracts integers as a bitfield.

For every bit, we have 4 possible states: mBits0 mBits1 State ---------------------- 0 0 Nothing was set to the bit (lowest lattice value - bottom) 1 0 The bit is set to 0 0 1 The bit is set to 1 1 1 The bit can be both 0 and 1 (highest lattice value - top)

10.2.2 Member Function Documentation

Bits ∗ Canal::Integer::Bits::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.

Implements Canal::Value.

Bits ∗ Canal::Integer::Bits::cloneCleaned () const `[virtual]`

Implementation of Value::cloneCleaned(). Covariant return type.

Implements Canal::Value.

int Canal::Integer::Bits::getBitValue (unsigned *pos*) const

Returns 0 if the bit is known to be 0. Returns 1 if the bit is known to be 1. Returns -1 if the bit value is unknown. Returns 2 if the bit is either 1 or 0.

void Canal::Integer::Bits::setBitValue (unsigned *pos*, int *value*)

Sets the bit. If value is 0 or 1, the bit is set to represent exactly 0 or 1. If value is -1, the bit is set to represent unknown value. If value is 2, the bit is set to represent both 0 and 1.

bool Canal::Integer::Bits::signedMax (llvm::APInt & *result*) const

Highest signed number represented by this abstract domain.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Bits::signedMin (llvm::APInt & *result*) const

Lowest signed number represented by this abstract domain.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Bits::unsignedMax (llvm::APInt & *result*) const

Highest unsigned number represented by this abstract domain.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Bits::unsignedMin (llvm::APInt & *result*) const

Lowest unsigned number represented by this abstract domain.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

10.2.3 Member Data Documentation

llvm::APInt Canal::Integer::Bits::mBits0

When a bit in mBits0 is 1, the value is known to contain zero at this position.

llvm::APInt Canal::Integer::Bits::mBits1

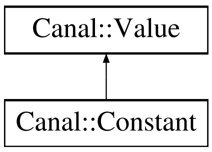
When a bit in mBits1 is 1, the value is known to contain one at this position.

The documentation for this class was generated from the following files:

- lib/IntegerBits.h
- lib/IntegerBits.cpp

10.3 Canal::Constant Class Reference

Inheritance diagram for Canal::Constant::



Public Member Functions

- **Constant** (const llvm::Constant ∗constant=NULL)
- bool **isAPInt** () const
- const llvm::APInt & **getAPInt** () const
- bool **isGetElementPtr** () const
- Value ∗ **toModifiableValue** () const
- virtual Constant ∗ clone () const

  Create a copy of this value.

- virtual Constant ∗ cloneCleaned () const
- virtual bool **operator==** (const Value &value) const
- virtual size_t memoryUsage () const

  Get memory usage (used byte count) of this abstract value.

- virtual std::string toString () const

  Create a string representation of the abstract value.

- virtual bool matchesString (const std::string &text, std::string &rationale) const

Public Attributes

- const llvm::Constant ∗ **mConstant**

10.3.1 Member Function Documentation

Constant ∗ Canal::Constant::cloneCleaned () const `[virtual]`

This is used to obtain instance of the value type and to get an empty value at the same time.
Implements Canal::Value.

bool Canal::Constant::matchesString (const std::string & *text*, std::string & *rationale*) const `[virtual]`

Checks if the abstract value internal state matches the text description. Full coverage of the state is not expected, the text can contain just partial information.
Implements Canal::Value.

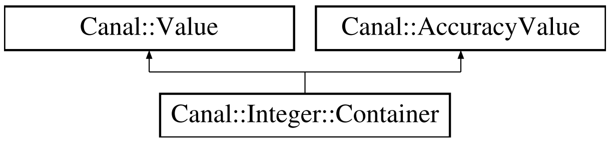
std::string Canal::Constant::toString () const `[virtual]`

Create a string representation of the abstract value. An idea for different memory interpretation. virtual Value ∗castTo(const llvm::Type ∗itemType, int offset) const = 0;
Implements Canal::Value.
The documentation for this class was generated from the following files:

- lib/Constant.h
- lib/Constant.cpp

10.4 Canal::Integer::Container Class Reference

Inheritance diagram for Canal::Integer::Container::



Public Member Functions

- **Container** (unsigned numBits)
- Container (const llvm::APInt &number)
- Container (const Container &container)

  Copy constructor. Creates independent copy of the container.

- virtual ~Container ()

  Destructor. Deletes the contents of the container.

- unsigned **getBitWidth** () const
- Bits & **getBits** ()
- const Bits & **getBits** () const
- Enumeration & **getEnumeration** ()
- const Enumeration & **getEnumeration** () const
- Range & **getRange** ()
- const Range & **getRange** () const
- bool signedMin (llvm::APInt &result) const
- bool signedMax (llvm::APInt &result) const
- bool unsignedMin (llvm::APInt &result) const
- bool unsignedMax (llvm::APInt &result) const
- virtual Container ∗ clone () const
- virtual Container ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().

- virtual void merge (const Value &value)

  Implementation of Value::merge().

- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().

- virtual std::string toString () const

  Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const
- virtual void add (const Value &a, const Value &b)

  Implementation of Value::add().

- virtual void sub (const Value &a, const Value &b)

  Implementation of Value::sub().

- virtual void mul (const Value &a, const Value &b)
  Implementation of Value::mul().
- virtual void udiv (const Value &a, const Value &b)
  Implementation of Value::udiv().
- virtual void sdiv (const Value &a, const Value &b)
  Implementation of Value::sdiv().
- virtual void urem (const Value &a, const Value &b)
  Implementation of Value::urem().
- virtual void srem (const Value &a, const Value &b)
  Implementation of Value::srem().
- virtual void shl (const Value &a, const Value &b)
  Implementation of Value::shl().
- virtual void lshr (const Value &a, const Value &b)
  Implementation of Value::lshr().
- virtual void ashr (const Value &a, const Value &b)
  Implementation of Value::ashr().
- virtual void and_ (const Value &a, const Value &b)
  Implementation of Value::and_().
- virtual void or_ (const Value &a, const Value &b)
  Implementation of Value::or_().
- virtual void xor_ (const Value &a, const Value &b)
  Implementation of Value::xor_().
- virtual void icmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
  Implementation of Value::icmp().
- virtual void fcmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
  Implementation of Value::fcmp().
- virtual float accuracy () const
  Implementation of AccuracyValue::accuracy().
- virtual bool isBottom () const
  Implementation of AccuracyValue::isBottom().
- virtual void setBottom ()
  Implementation of AccuracyValue::setBottom().
- virtual bool isTop () const
  Implementation of AccuracyValue::isTop().
- virtual void setTop ()
  Implementation of AccuracyValue::setTop().

Public Attributes

- std::vector< Value ∗ > **mValues**

10.4.1 Constructor & Destructor Documentation

Canal::Integer::Container::Container (const llvm::APInt & *number*)

Creates a new container with an initial value. Signedness, number of bits is taken from the provided number.

10.4.2 Member Function Documentation

Container ∗ Canal::Integer::Container::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.
Implements Canal::Value.

Container ∗ Canal::Integer::Container::cloneCleaned () const `[virtual]`

Implementation of Value::cloneCleaned(). Covariant return type.
Implements Canal::Value.

bool Canal::Integer::Container::matchesString (const std::string & *text*, std::string & *rationale*) const `[virtual]`

Implementation of Value::matchesString(). Examples: integer enumeration -8 integer enumeration -10 2 4 6 8 range -10 8
Implements Canal::Value.

bool Canal::Integer::Container::signedMax (llvm::APInt & *result*) const

Highest signed number represented by this container. Uses the abstract domain (enum, range, bits) with highest precision.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Container::signedMin (llvm::APInt & *result*) const

Lowest signed number represented by this container. Uses the abstract domain (enum, range, bits) with highest precision.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Container::unsignedMax (llvm::APInt & *result*) const

Highest unsigned number represented by this container. Uses the abstract domain (enum, range, bits) with highest precision.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Container::unsignedMin (llvm::APInt & *result*) const

Lowest unsigned number represented by this container. Uses the abstract domain (enum, range, bits) with highest precision.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

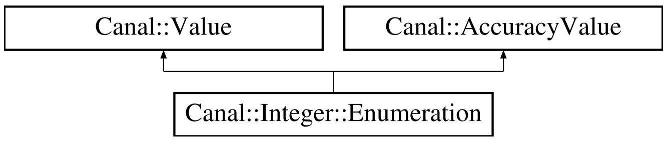
True if the result is known and the parameter was set to correct value.

The documentation for this class was generated from the following files:

- lib/IntegerContainer.h
- lib/IntegerContainer.cpp

10.5 Canal::Integer::Enumeration Class Reference

Inheritance diagram for Canal::Integer::Enumeration::



Public Member Functions

- Enumeration (unsigned numBits)

  Initializes to the lowest value.

- Enumeration (const llvm::APInt &number)

  Initializes to the given value.

- unsigned **getBitWidth** () const
- bool signedMin (llvm::APInt &result) const
- bool signedMax (llvm::APInt &result) const
- bool unsignedMin (llvm::APInt &result) const
- bool unsignedMax (llvm::APInt &result) const
- virtual Enumeration ∗ clone () const
- virtual Enumeration ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().

- virtual void merge (const Value &value)

  Implementation of Value::merge().

- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().

- virtual std::string toString () const

  Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const
- virtual void add (const Value &a, const Value &b)

  Implementation of Value::add().

- virtual void sub (const Value &a, const Value &b)

  Implementation of Value::sub().

- virtual void mul (const Value &a, const Value &b)

  Implementation of Value::mul().

- virtual void udiv (const Value &a, const Value &b)

  Implementation of Value::udiv().

- virtual void sdiv (const Value &a, const Value &b)

  Implementation of Value::sdiv().

- virtual void urem (const Value &a, const Value &b)

  Implementation of Value::urem().

- virtual void srem (const Value &a, const Value &b)

  Implementation of Value::srem().

- virtual void shl (const Value &a, const Value &b)

  Implementation of Value::shl().

- virtual void lshr (const Value &a, const Value &b)

  Implementation of Value::lshr().

- virtual void ashr (const Value &a, const Value &b)

  Implementation of Value::ashr().

- virtual void and_ (const Value &a, const Value &b)

  Implementation of Value::and_().

- virtual void or_ (const Value &a, const Value &b)

  Implementation of Value::or_().

- virtual void xor_ (const Value &a, const Value &b)

  Implementation of Value::xor_().

- virtual void icmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)

  Implementation of Value::icmp().

- virtual void fcmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)

  Implementation of Value::fcmp().

- virtual float accuracy () const

  Implementation of AccuracyValue::accuracy().

- virtual bool isBottom () const

  Implementation of AccuracyValue::isBottom().

- virtual void setBottom ()

  Implementation of AccuracyValue::setBottom().

- virtual bool isTop () const

  Implementation of AccuracyValue::isTop().

- virtual void setTop ()

  Implementation of AccuracyValue::setTop().

Public Attributes

- APIntUtils::USet **mValues**
- bool **mTop**
- unsigned **mNumBits**

Protected Member Functions

- void **applyOperation** (const Value &a, const Value &b, APIntUtils::Operation operation1, APIntUtils::OperationWithOverflo
  operation2)

10.5.1 Member Function Documentation

Enumeration ∗ Canal::Integer::Enumeration::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.
Implements Canal::Value.

Enumeration ∗ Canal::Integer::Enumeration::cloneCleaned () const `[virtual]`

Implementation of Value::cloneCleaned(). Covariant return type.
Implements Canal::Value.

bool Canal::Integer::Enumeration::matchesString (const std::string & *text*, std::string & *rationale*) const `[virtual]`

Implementation of Value::matchesString(). Examples of allowed input: enumeration 3 -4 5 enumeration 0xff 0xfa enumeration 0x1000000F 0xABABABAB enumeration top enumeration empty enumeration empty
Implements Canal::Value.

bool Canal::Integer::Enumeration::signedMax (llvm::APInt & *result*) const

Highest signed number represented by this abstract domain.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Enumeration::signedMin (llvm::APInt & *result*) const

Lowest signed number represented by this abstract domain.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Enumeration::unsignedMax (llvm::APInt & *result*) const

Highest unsigned number represented by this abstract domain.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Enumeration::unsignedMin (llvm::APInt & *result*) const

Lowest unsigned number represented by this abstract domain.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

The documentation for this class was generated from the following files:

- lib/IntegerEnumeration.h
- lib/IntegerEnumeration.cpp

10.6 Canal::Environment Class Reference

Public Member Functions

- **Environment** (const llvm::Module &module)

Public Attributes

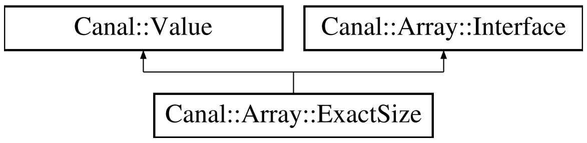
- const llvm::Module & **mModule**
- llvm::TargetData **mTargetData**
- SlotTracker **mSlotTracker**

The documentation for this class was generated from the following files:

- lib/Environment.h
- lib/Environment.cpp

10.7 Canal::Array::ExactSize Class Reference

`#include <ArrayExactSize.h>`Inheritance diagram for Canal::Array::ExactSize::



Public Member Functions

- **ExactSize** (const ExactSize &exactSize)
- size_t **size** () const
- virtual ExactSize ∗ clone () const
- virtual ExactSize ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().

- virtual void merge (const Value &value)

  Implementation of Value::merge().

- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().

- virtual std::string toString () const

  Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const

  Implementation of Value::matchesString().

- virtual void add (const Value &a, const Value &b)

  Implementation of Value::add().

- virtual void fadd (const Value &a, const Value &b)

  Implementation of Value::fadd().

- virtual void sub (const Value &a, const Value &b)

  Implementation of Value::sub().

- virtual void fsub (const Value &a, const Value &b)

  Implementation of Value::fsub().

- virtual void mul (const Value &a, const Value &b)

  Implementation of Value::mul().

- virtual void fmul (const Value &a, const Value &b)

  Implementation of Value::fmul().

- virtual void udiv (const Value &a, const Value &b)

  Implementation of Value::udiv().

- virtual void sdiv (const Value &a, const Value &b)
  Implementation of Value::sdiv().
- virtual void fdiv (const Value &a, const Value &b)
  Implementation of Value::fdiv().
- virtual void urem (const Value &a, const Value &b)
  Implementation of Value::urem().
- virtual void srem (const Value &a, const Value &b)
  Implementation of Value::srem().
- virtual void frem (const Value &a, const Value &b)
  Implementation of Value::frem().
- virtual void shl (const Value &a, const Value &b)
  Implementation of Value::shl().
- virtual void lshr (const Value &a, const Value &b)
  Implementation of Value::lshr().
- virtual void ashr (const Value &a, const Value &b)
  Implementation of Value::ashr().
- virtual void and_ (const Value &a, const Value &b)
  Implementation of Value::and_().
- virtual void or_ (const Value &a, const Value &b)
  Implementation of Value::or_().
- virtual void xor_ (const Value &a, const Value &b)
  Implementation of Value::xor_().
- virtual void icmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
  Implementation of Value::icmp().
- virtual void fcmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
  Implementation of Value::fcmp().
- virtual std::vector< Value ∗ > getItem (const Value &offset) const
  Implementation of Array::Interface::getItem().
- virtual Value ∗ getItem (uint64_t offset) const
  Implementation of Array::Interface::getItem().
- virtual void setItem (const Value &offset, const Value &value)
  Implementation of Array::Interface::setItem().
- virtual void setItem (uint64_t offset, const Value &value)
  Implementation of Array::Interface::setItem().

Public Attributes

- std::vector< Value * > **mValues**

10.7.1 Detailed Description

Array with exact size and limited length. It keeps all array members separately, not losing precision at all.

10.7.2 Member Function Documentation

ExactSize * Canal::Array::ExactSize::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.
Implements Canal::Value.

ExactSize * Canal::Array::ExactSize::cloneCleaned () const `[virtual]`

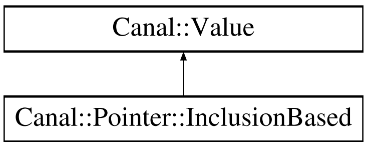
Implementation of Value::cloneCleaned(). Covariant return type.
Implements Canal::Value.
The documentation for this class was generated from the following files:

- lib/ArrayExactSize.h
- lib/ArrayExactSize.cpp

10.8 Canal::Pointer::InclusionBased Class Reference

Inclusion-based flow-insensitive abstract pointer.

`#include <Pointer.h>`Inheritance diagram for Canal::Pointer::InclusionBased::



Public Member Functions

- InclusionBased (const llvm::Module &module, const llvm::Type ∗type)

    Standard constructor.

- InclusionBased (const InclusionBased &second)

    Copy constructor.

- virtual ~InclusionBased ()

    Standard destructor.

- void addTarget (Target::Type type, const llvm::Value ∗instruction, const llvm::Value ∗target, const std::vector< Value ∗ > &offsets, Value ∗numericOffset)
- Value ∗ dereferenceAndMerge (const State &state) const
- InclusionBased ∗ bitcast (const llvm::Type ∗type) const

    Creates a copy of this object with a different pointer type.

- InclusionBased ∗ getElementPtr (const std::vector< Value ∗ > &offsets, const llvm::Type ∗type) const
- void **store** (const Value &value, State &state)
- virtual InclusionBased ∗ clone () const
- virtual InclusionBased ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

    Implementation of Value::operator==().

- virtual void merge (const Value &value)

    Implementation of Value::merge().

- virtual size_t memoryUsage () const

    Implementation of Value::memoryUsage().

- virtual std::string toString () const

    Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const

    Implementation of Value::matchesString().

10.8.1 Detailed Description

Inclusion-based flow-insensitive abstract pointer.

10.8.2 Member Function Documentation

void Canal::Pointer::InclusionBased::addTarget (Target::Type *type*, const llvm::Value ∗ *instruction*, const llvm::Value ∗ *target*, const std::vector< Value ∗ > & *offsets*, Value ∗ *numericOffset*)

Add a new target to the pointer.

Parameters:

type Type of the referenced memory.

instruction Place where the pointer target is added.

target Represents the target memory block. If type is Constant, it must be NULL. Otherwise, it must be a valid pointer to an instruction. This is a key to State::mFunctionBlocks, State::mFunctionVariables, State::mGlobalBlocks, or State::mGlobalVariables, depending on the type.

offsets Offsets in the getelementptr style. The provided vector might be empty. The newly created pointer target becomes the owner of the objects in the vector.

numericOffset Numerical offset that is used in addition to the getelementptr style offset and after they have been applied. It might be NULL, which indicates the offset 0. The newly created pointer target becomes the owner of the numerical offset when it's provided. This parameter is mandatory for pointers of Constant type, because it contains the constant.

InclusionBased ∗ Canal::Pointer::InclusionBased::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type -- it really overrides Value::clone().
Implements Canal::Value.

InclusionBased ∗ Canal::Pointer::InclusionBased::cloneCleaned () const `[virtual]`

Implementation of Value::cloneCleaned(). Covariant return type.
Implements Canal::Value.

Value ∗ Canal::Pointer::InclusionBased::dereferenceAndMerge (const State & *state*) const

Dereference all targets and merge the results into single abstract value. The returned value is owned by the caller.

Returns:

It might return NULL.

InclusionBased ∗ Canal::Pointer::InclusionBased::getElementPtr (const std::vector< Value ∗ > & *offsets*, const llvm::Type ∗ *type*) const

Creates a copy of this object pointing to subtargets.

Parameters:

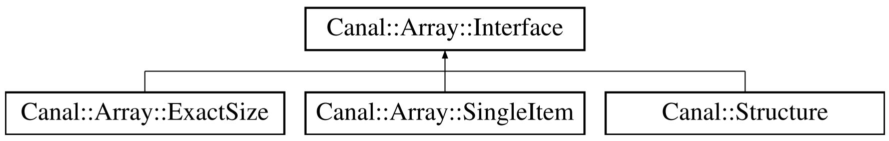
offsets Pointer takes ownership of the values inside the vector.

The documentation for this class was generated from the following files:

- lib/Pointer.h
- lib/Pointer.cpp

10.9 Canal::Array::Interface Class Reference

Inheritance diagram for Canal::Array::Interface::



Public Member Functions

- Value $*$ getValue (const Value &offset) const
- Value $*$ getValue (uint64_t offset) const
- virtual std::vector< Value $*$ > getItem (const Value &offset) const =0
- virtual Value $*$ getItem (uint64_t offset) const =0
- virtual void setItem (const Value &offset, const Value &value)=0
- virtual void setItem (uint64_t offset, const Value &value)=0

10.9.1 Member Function Documentation

virtual Value$*$ Canal::Array::Interface::getItem (uint64_t *offset*) const `[pure virtual]`

Get the array item pointed by the provided offset. Returns internal array item that is owned by the array. Caller must not delete the item.

Note:

The uint64_t offset variant exists because of the extractvalue instruction, which provides exact numeric offsets. For future array domains it might be necessary to extend this method to return a list of values.

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, and Canal::Structure.

virtual std::vector<Value$*$> Canal::Array::Interface::getItem (const Value & *offset*) const `[pure virtual]`

Get the array items pointed by the provided offset. Returns internal array items that are owned by the array. Caller must not delete the items.

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, and Canal::Structure.

Value$*$ Canal::Array::Interface::getValue (uint64_t *offset*) const

Gets the value representing the array item pointed by the provided offset. Caller is responsible for deleting the returned value.

Note:

The uint64_t offset variant exists because of the extractvalue instruction, which provides exact numeric offsets.

Value $*$ Canal::Array::Interface::getValue (const Value & *offset*) const

Gets the value representing the array item or items pointed by the provided offset. Caller is responsible for deleting the returned value.

virtual void Canal::Array::Interface::setItem (uint64_t *offset*, const Value & *value*) `[pure virtual]`

Parameters:

value The method does not take the ownership of this memory. It copies the contents of the value instead.

Note:

The uint64_t offset variant exists because of the insertvalue instruction, which provides exact numeric offsets.

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, and Canal::Structure.

virtual void Canal::Array::Interface::setItem (const Value & *offset*, const Value & *value*) `[pure virtual]`

Parameters:

value The method does not take the ownership of this memory. It copies the contents of the value instead.

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, and Canal::Structure.
The documentation for this class was generated from the following files:

- lib/ArrayInterface.h
- lib/ArrayInterface.cpp

10.10 Canal::Interpreter Class Reference

`#include <Interpreter.h>`

Public Member Functions

- void addGlobalVariables (State &state, const Environment &environment)
- virtual bool step (Stack &stack, const Environment &environment)
- void interpretInstruction (Stack &stack, const Environment &environment)

  Interprets current instruction.

Protected Member Functions

- virtual void ret (const llvm::ReturnInst &instruction, State &state, const Environment &environment)
- virtual void br (const llvm::BranchInst &instruction, State &state, const Environment &environment)
- virtual void switch_ (const llvm::SwitchInst &instruction, State &state, const Environment &environment)
- virtual void indirectbr (const llvm::IndirectBrInst &instruction, State &state, const Environment &environment)
- virtual void invoke (const llvm::InvokeInst &instruction, Stack &stack, const Environment &environment)
- virtual void unreachable (const llvm::UnreachableInst &instruction, State &state, const Environment &environment)
- virtual void add (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Sum of two operands. It's a binary operator.

- virtual void fadd (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)
- virtual void sub (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Difference of two operands. It's a binary operator.

- virtual void fsub (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)
- virtual void mul (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Product of two operands. It's a binary operator.

- virtual void fmul (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Product of two operands. It's a binary operator.

- virtual void udiv (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)
- virtual void sdiv (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)
- virtual void fdiv (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)
- virtual void urem (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Unsigned division remainder. It's a binary operator.

- virtual void srem (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Signed division remainder. It's a binary operator.

- virtual void frem (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  Floating point remainder. It's a binary operator.

- virtual void shl (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  It's a bitwise binary operator.

- virtual void lshr (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  It's a bitwise binary operator.

- virtual void ashr (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  It's a bitwise binary operator.

- virtual void and_ (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  It's a bitwise binary operator.

- virtual void or_ (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  It's a bitwise binary operator.

- virtual void xor_ (const llvm::BinaryOperator &instruction, State &state, const Environment &environment)

  It's a bitwise binary operator.

- virtual void extractelement (const llvm::ExtractElementInst &instruction, State &state, const Environment &environment)

  It's a vector operation.

- virtual void insertelement (const llvm::InsertElementInst &instruction, State &state, const Environment &environment)

  It's a vector operation.

- virtual void shufflevector (const llvm::ShuffleVectorInst &instruction, Stack &stack, const Environment &environment)

  It's a vector operation.

- virtual void extractvalue (const llvm::ExtractValueInst &instruction, State &state, const Environment &environment)

  It's an aggregate operation.

- virtual void insertvalue (const llvm::InsertValueInst &instruction, State &state, const Environment &environment)

  It's an aggregate operation.

- virtual void alloca_ (const llvm::AllocaInst &instruction, Stack &stack, const Environment &environment)

  It's a memory access operation.

- virtual void load (const llvm::LoadInst &instruction, State &state, const Environment &environment)

  It's a memory access operation.

- virtual void store (const llvm::StoreInst &instruction, State &state, const Environment &environment)

  It's a memory access operation.

- virtual void getelementptr (const llvm::GetElementPtrInst &instruction, Stack &stack, const Environment &environment)

  It's a memory addressing operation.

- virtual void trunc (const llvm::TruncInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void zext (const llvm::ZExtInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void sext (const llvm::SExtInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void fptrunc (const llvm::FPTruncInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void fpext (const llvm::FPExtInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void fptoui (const llvm::FPToUIInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void fptosi (const llvm::FPToSIInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void uitofp (const llvm::UIToFPInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void sitofp (const llvm::SIToFPInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void ptrtoint (const llvm::PtrToIntInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void inttoptr (const llvm::IntToPtrInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void bitcast (const llvm::BitCastInst &instruction, State &state, const Environment &environment)

  It's a conversion operation.

- virtual void **icmp** (const llvm::ICmpInst &instruction, State &state, const Environment &environment)
- virtual void **fcmp** (const llvm::FCmpInst &instruction, State &state, const Environment &environment)
- virtual void **phi** (const llvm::PHINode &instruction, State &state, const Environment &environment)
- virtual void **select** (const llvm::SelectInst &instruction, State &state, const Environment &environment)
- virtual void **call** (const llvm::CallInst &instruction, Stack &stack, const Environment &environment)
- virtual void **va_arg** (const llvm::VAArgInst &instruction, State &state, const Environment &environment)

10.10.1 Detailed Description

Context-sensitive flow-insensitive operational abstract interpreter. Interprets instructions in abstract domain.

This is an abstract class, which is used as a base class for actual abstract interpretation implementations.

10.10.2 Member Function Documentation

void Canal::Interpreter::addGlobalVariables (State & *state*, const Environment & *environment*)

Adds all global variables and constants from a module to the state.

void Canal::Interpreter::br (const llvm::BranchInst & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Transfer to a different basic block in the current function. It's a terminator instruction.

void Canal::Interpreter::fadd (const llvm::BinaryOperator & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Sum of two operands. It's a binary operator. The operands are floating point or vector of floating point values.

void Canal::Interpreter::fdiv (const llvm::BinaryOperator & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Quotient of two operands. It's a binary operator. The operands are floating point or vector of floating point values.

void Canal::Interpreter::fsub (const llvm::BinaryOperator & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Difference of two operands. It's a binary operator. The operands are floating point or vector of floating point values.

void Canal::Interpreter::indirectbr (const llvm::IndirectBrInst & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

An indirect branch to a label within the current function, whose address is specified by ”address”. It's a terminator instruction.

void Canal::Interpreter::invoke (const llvm::InvokeInst & *instruction*, Stack & *stack*, const Environment & *environment*) `[protected, virtual]`

Transfer to a specified function, with the possibility of control flow transfer to either the 'normal' label or the 'exception' label. It's a terminator instruction.

void Canal::Interpreter::ret (const llvm::ReturnInst & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Return control flow (and optionally a value) from a function back to the caller. It's a terminator instruction.

void Canal::Interpreter::sdiv (const llvm::BinaryOperator & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Quotient of two operands. It's a binary operator. The operands are integer or vector of integer values.

bool Canal::Interpreter::step (Stack & *stack*, const Environment & *environment*) `[virtual]`

One step of the interpreter. Interprets current instruction and moves to the next one.

Returns:

True if next step is possible. False on the end of the program.

void Canal::Interpreter::switch_ (const llvm::SwitchInst & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Transfer control flow to one of several different places. It is a generalization of the 'br' instruction, allowing a branch to occur to one of many possible destinations. It's a terminator instruction.

void Canal::Interpreter::udiv (const llvm::BinaryOperator & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

Quotient of two operands. It's a binary operator. The operands are integer or vector of integer values.

void Canal::Interpreter::unreachable (const llvm::UnreachableInst & *instruction*, State & *state*, const Environment & *environment*) `[protected, virtual]`

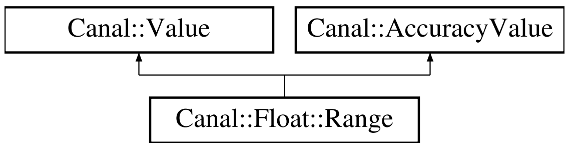
No defined semantics. This instruction is used to inform the optimizer that a particular portion of the code is not reachable. It's a terminator instruction.

The documentation for this class was generated from the following files:

- lib/Interpreter.h
- lib/Interpreter.cpp

10.11 Canal::Float::Range Class Reference

Inheritance diagram for Canal::Float::Range::



Public Member Functions

- **Range** (const llvm::fltSemantics &semantics)
- int **compare** (const Range &value, llvm::CmpInst::Predicate predicate) const
- const llvm::fltSemantics & **getSemantics** () const
- virtual Range ∗ clone () const

  Create a copy of this value.

- virtual Range ∗ cloneCleaned () const
- virtual bool **operator==** (const Value &value) const
- virtual void **merge** (const Value &value)
- virtual size_t memoryUsage () const

  Get memory usage (used byte count) of this abstract value.

- virtual std::string toString () const

  Create a string representation of the abstract value.

- virtual bool matchesString (const std::string &text, std::string &rationale) const
- virtual float accuracy () const
- virtual bool isBottom () const

  Is it the lowest value.

- virtual void setBottom ()

  Set to the lowest value.

- virtual bool isTop () const

  Is it the highest value.

- virtual void setTop ()

  Set it to the top value of lattice.

Public Attributes

- bool **mEmpty**
- bool **mTop**
- llvm::APFloat **mFrom**
- llvm::APFloat **mTo**

10.11.1 Member Function Documentation

float Canal::Float::Range::accuracy () const `[virtual]`

Get accuracy of the abstract value (0 - 1). In finite-height lattices, it is determined by the position of the value in the lattice.

Accuracy 0 means that the value represents all possible values (top). Accuracy 1 means that the value represents the most precise and exact value (bottom).

Reimplemented from Canal::AccuracyValue.

Range ∗ Canal::Float::Range::cloneCleaned () const `[virtual]`

This is used to obtain instance of the value type and to get an empty value at the same time.

Implements Canal::Value.

bool Canal::Float::Range::matchesString (const std::string & *text*, std::string & *rationale*) const `[virtual]`

Checks if the abstract value internal state matches the text description. Full coverage of the state is not expected, the text can contain just partial information.

Implements Canal::Value.

std::string Canal::Float::Range::toString () const `[virtual]`

Create a string representation of the abstract value. An idea for different memory interpretation. virtual Value ∗castTo(const llvm::Type ∗itemType, int offset) const = 0;

Implements Canal::Value.

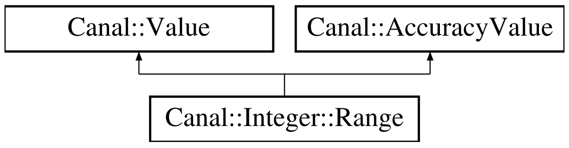
The documentation for this class was generated from the following files:

- lib/FloatRange.h
- lib/FloatRange.cpp

10.12 Canal::Integer::Range Class Reference

Abstracts integer values as a range min - max.

`#include <IntegerRange.h>`Inheritance diagram for Canal::Integer::Range::



Public Member Functions

- Range (unsigned numBits)

  Initializes to the lowest value.

- **Range** (const llvm::APInt &constant)
- unsigned **getBitWidth** () const
- bool signedMin (llvm::APInt &result) const
- bool signedMax (llvm::APInt &result) const
- bool unsignedMin (llvm::APInt &result) const
- bool unsignedMax (llvm::APInt &result) const
- bool isSingleValue () const
- virtual Range ∗ clone () const
- virtual Range ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().

- virtual void merge (const Value &value)

  Implementation of Value::merge().

- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().

- virtual std::string toString () const

  Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const

  Implementation of Value::matchesString().

- virtual void add (const Value &a, const Value &b)

  Implementation of Value::add().

- virtual void sub (const Value &a, const Value &b)

  Implementation of Value::sub().

- virtual void mul (const Value &a, const Value &b)

  Implementation of Value::mul().

- virtual void udiv (const Value &a, const Value &b)

  Implementation of Value::udiv().

- virtual void sdiv (const Value &a, const Value &b)

  Implementation of Value::sdiv().

- virtual void urem (const Value &a, const Value &b)

  Implementation of Value::urem().

- virtual void srem (const Value &a, const Value &b)

  Implementation of Value::srem().

- virtual void shl (const Value &a, const Value &b)

  Implementation of Value::shl().

- virtual void lshr (const Value &a, const Value &b)

  Implementation of Value::lshr().

- virtual void ashr (const Value &a, const Value &b)

  Implementation of Value::ashr().

- virtual void and_ (const Value &a, const Value &b)

  Implementation of Value::and_().

- virtual void or_ (const Value &a, const Value &b)

  Implementation of Value::or_().

- virtual void xor_ (const Value &a, const Value &b)

  Implementation of Value::xor_().

- virtual void icmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)

  Implementation of Value::icmp().

- virtual float accuracy () const

  Implementation of AccuracyValue::accuracy().

- virtual bool isBottom () const

  Implementation of AccuracyValue::isBottom().

- virtual void setBottom ()

  Implementation of AccuracyValue::setBottom().

- virtual bool isTop () const

  Implementation of AccuracyValue::isTop().

- virtual void setTop ()

  Implementation of AccuracyValue::setTop().

Public Attributes

- bool **mEmpty**
- bool **mSignedTop**
- llvm::APInt mSignedFrom

  The number is included in the interval.

- llvm::APInt mSignedTo

    The number is included in the interval.

- bool **mUnsignedTop**
- llvm::APInt mUnsignedFrom

    The number is included in the interval.

- llvm::APInt mUnsignedTo

    The number is included in the interval.

10.12.1 Detailed Description

Abstracts integer values as a range min - max.

10.12.2 Member Function Documentation

Range ∗ Canal::Integer::Range::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.
Implements Canal::Value.

Range ∗ Canal::Integer::Range::cloneCleaned () const `[virtual]`

Implementation of Value::cloneCleaned(). Covariant return type.
Implements Canal::Value.

bool Canal::Integer::Range::isSingleValue () const

Returns true if the range represents a single number. Signed and unsigned representations might differ, though.

bool Canal::Integer::Range::signedMax (llvm::APInt & *result*) const

Highest signed number represented by this abstract domain.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Range::signedMin (llvm::APInt & *result*) const

Lowest signed number represented by this abstract domain.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Range::unsignedMax (llvm::APInt & *result*) const

Highest unsigned number represented by this abstract domain.

Parameters:

result Filled by the maximum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

bool Canal::Integer::Range::unsignedMin (llvm::APInt & *result*) const

Lowest unsigned number represented by this abstract domain.

Parameters:

result Filled by the minimum value if it is known. Otherwise, the value is undefined.

Returns:

True if the result is known and the parameter was set to correct value.

The documentation for this class was generated from the following files:

- lib/IntegerRange.h
- lib/IntegerRange.cpp

10.13 Canal::APIntUtils::SCompare Struct Reference

Public Member Functions

- bool **operator()** (const llvm::APInt &a, const llvm::APInt &b) const

The documentation for this struct was generated from the following file:

- lib/APIntUtils.h

10.14 Canal::SELinuxModulePass Class Reference

Public Member Functions

- virtual bool **runOnModule** (llvm::Module &module)
- void **interpretFunction** (const llvm::Function &F, const std::vector< Value > &Arguments)
- virtual void **getAnalysisUsage** (llvm::AnalysisUsage &AU) const

Static Public Attributes

- static char **ID** = 0

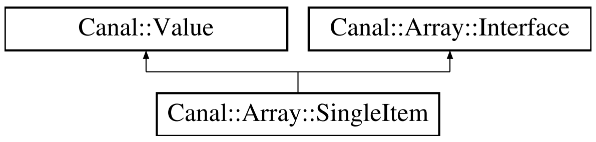
The documentation for this class was generated from the following file:

- lib/SELinuxModulePass.cpp

10.15 Canal::Array::SingleItem Class Reference

This array type is very imprecise.

`#include <ArraySingleItem.h>`Inheritance diagram for Canal::Array::SingleItem::



Public Member Functions

- **SingleItem** (const SingleItem &singleItem)
- virtual SingleItem ∗ clone () const
- virtual SingleItem ∗ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().
- virtual void merge (const Value &value)

  Implementation of Value::merge().
- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().
- virtual std::string toString () const

  Implementation of Value::toString().
- virtual bool matchesString (const std::string &text, std::string &rationale) const

  Implementation of Value::matchesString().
- virtual void add (const Value &a, const Value &b)

  Implementation of Value::add().
- virtual void fadd (const Value &a, const Value &b)

  Implementation of Value::fadd().
- virtual void sub (const Value &a, const Value &b)

  Implementation of Value::sub().
- virtual void fsub (const Value &a, const Value &b)

  Implementation of Value::fsub().
- virtual void mul (const Value &a, const Value &b)

  Implementation of Value::mul().
- virtual void fmul (const Value &a, const Value &b)

  Implementation of Value::fmul().
- virtual void udiv (const Value &a, const Value &b)

  Implementation of Value::udiv().

- virtual void sdiv (const Value &a, const Value &b)

  Implementation of Value::sdiv().

- virtual void fdiv (const Value &a, const Value &b)

  Implementation of Value::fdiv().

- virtual void urem (const Value &a, const Value &b)

  Implementation of Value::urem().

- virtual void srem (const Value &a, const Value &b)

  Implementation of Value::srem().

- virtual void frem (const Value &a, const Value &b)

  Implementation of Value::frem().

- virtual void shl (const Value &a, const Value &b)

  Implementation of Value::shl().

- virtual void lshr (const Value &a, const Value &b)

  Implementation of Value::lshr().

- virtual void ashr (const Value &a, const Value &b)

  Implementation of Value::ashr().

- virtual void and_ (const Value &a, const Value &b)

  Implementation of Value::and_().

- virtual void or_ (const Value &a, const Value &b)

  Implementation of Value::or_().

- virtual void xor_ (const Value &a, const Value &b)

  Implementation of Value::xor_().

- virtual void icmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)

  Implementation of Value::icmp().

- virtual void fcmp (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)

  Implementation of Value::fcmp().

- virtual std::vector< Value * > getItem (const Value &offset) const

  Implementation of Array::Interface::getItem().

- virtual Value * getItem (uint64_t offset) const

  Implementation of Array::Interface::getItem().

- virtual void setItem (const Value &offset, const Value &value)

  Implementation of Array::Interface::setItem().

- virtual void setItem (uint64_t offset, const Value &value)

  Implementation of Array::Interface::setItem().

Public Attributes

- Value ∗ **mValue**
- Value ∗ mSize

10.15.1 Detailed Description

This array type is very imprecise. The most trivial array type. It treats all array members as a single value. This means all the operations on the array are merged and used to move the single value up in its lattice.

10.15.2 Member Function Documentation

SingleItem ∗ Canal::Array::SingleItem::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.
Implements Canal::Value.

SingleItem ∗ Canal::Array::SingleItem::cloneCleaned () const `[virtual]`

Implementation of Value::cloneCleaned(). Covariant return type.
Implements Canal::Value.

10.15.3 Member Data Documentation

Value∗ Canal::Array::SingleItem::mSize

Number of elements in the array. It is either a Constant or Integer::Container.
The documentation for this class was generated from the following files:

- lib/ArraySingleItem.h
- lib/ArraySingleItem.cpp

10.16 Canal::SlotTracker Class Reference

`#include <SlotTracker.h>`

Public Types

- typedef std::map< const llvm::MDNode *, unsigned >::iterator mdn_iterator

  MDNode map iterators.

Public Member Functions

- SlotTracker (const llvm::Module &module)

  Construct from a module.

- void setActiveFunction (const llvm::Function &function)
- int getLocalSlot (const llvm::Value &value)
- const llvm::Value * **getLocalSlot** (int num)
- int getGlobalSlot (const llvm::Value &value)

  Get the slot number of a global value.

- const llvm::Value * **getGlobalSlot** (int num)
- int getMetadataSlot (const llvm::MDNode &node)

  Get the slot number of a MDNode.

- mdn_iterator **mdn_begin** ()
- mdn_iterator **mdn_end** ()
- unsigned **mdn_size** () const
- bool **mdn_empty** () const

Protected Types

- typedef std::map< const llvm::Value *, unsigned > ValueMap

  A mapping of Values to slot numbers.

- typedef std::vector< const llvm::Value * > **ValueList**

Protected Member Functions

- void initialize ()

  This function does the actual initialization.

- void createFunctionSlot (const llvm::Value &value)

  Insert the specified Value into the slot table.*

- void createModuleSlot (const llvm::GlobalValue &value)

  Insert the specified GlobalValue into the slot table.*

- void createMetadataSlot (const llvm::MDNode &node)

  Insert the specified MDNode into the slot table.*

- void processModule ()
- void processFunction ()

Protected Attributes

- const llvm::Module & mModule

  The module for which we are holding slot numbers.
- bool **mModuleProcessed**
- const llvm::Function ∗ mFunction

  The function for which we are holding slot numbers.
- bool **mFunctionProcessed**
- ValueMap mModuleMap

  The slot map for the module level data.
- ValueList **mModuleList**
- unsigned **mModuleNext**
- ValueMap mFunctionMap

  The slot map for the function level data.
- ValueList **mFunctionList**
- unsigned **mFunctionNext**
- std::map< const llvm::MDNode ∗, unsigned > mMetadataMap

  The slot map for MDNodes.
- unsigned **mMetadataNext**

10.16.1 Detailed Description

This class provides computation of slot numbers. Initial version was taken from LLVM source code (lib/VMCore/AsmWriter.cpp).

10.16.2 Member Function Documentation

int Canal::SlotTracker::getLocalSlot (const llvm::Value & *value*)

Get the slot number for a value that is local to a function. Return the slot number of the specified value in it's type plane. If something is not in the SlotTracker, return -1.

void Canal::SlotTracker::processFunction () `[protected]`

Add all of the functions arguments, basic blocks, and instructions.

void Canal::SlotTracker::processModule () `[protected]`

Add all of the module level global variables (and their initializers) and function declarations, but not the contents of those functions.

void Canal::SlotTracker::setActiveFunction (const llvm::Function & *function*)

If you'd like to deal with a function instead of just a module, use this method to get its data into the SlotTracker.
The documentation for this class was generated from the following files:

- lib/SlotTracker.h
- lib/SlotTracker.cpp

10.17 Canal::Stack Class Reference

Public Member Functions

- bool **nextInstruction** ()
- bool **hasEnteredNewFrame** () const
- bool **hasReturnedFromFrame** () const
- std::vector< StackFrame > & **getFrames** ()
- const std::vector< StackFrame > & **getFrames** () const
- const llvm::Instruction & **getCurrentInstruction** () const
- State & **getCurrentState** ()
- const llvm::Function & **getCurrentFunction** () const
- void addFrame (const llvm::Function &function, const State &initialState)

Protected Attributes

- std::vector< StackFrame > **mFrames**
- bool **mHasEnteredNewFrame**
- bool **mHasReturnedFromFrame**

10.17.1 Member Function Documentation

void Canal::Stack::addFrame (const llvm::Function & ***function*****, const State &** ***initialState*****)**

Parameters:

function Function to be interpreted. Its instructions will be applied in abstract domain on the provided input state.

initialState Initial state when entering the function. It includes global variables and function arguments.

The documentation for this class was generated from the following files:

- lib/Stack.h
- lib/Stack.cpp

10.18 Canal::StackFrame Class Reference

Public Member Functions

- **StackFrame** (const llvm::Function ∗function, const State &initialState)
- bool nextInstruction ()
- Value ∗ **getReturnedValue** () const
- void **mergeGlobalVariables** (State &target) const

Public Attributes

- const llvm::Function ∗ mFunction
- std::map< const llvm::BasicBlock ∗, State > **mBlockInputState**
- std::map< const llvm::BasicBlock ∗, State > **mBlockOutputState**
- llvm::Function::const_iterator **mCurrentBlock**
- State **mCurrentState**
- llvm::BasicBlock::const_iterator **mCurrentInstruction**
- bool **mChanged**

10.18.1 Member Function Documentation

bool Canal::StackFrame::nextInstruction ()

Returns true if next instruction should be interpreted for this frame, and false when fixpoint has been reached.

10.18.2 Member Data Documentation

const llvm::Function∗ Canal::StackFrame::mFunction

Never is NULL. It is a pointer just to allow storing instances of this class in std::vector.
The documentation for this class was generated from the following files:

- lib/Stack.h
- lib/Stack.cpp

10.19 Canal::State Class Reference

```
#include <State.h>
```

Public Member Functions

- **State** (const State &state)
- State & **operator=** (const State &state)
- bool **operator==** (const State &state) const
- bool **operator!=** (const State &state) const
- void clear ()

  Clears everything. Releases all memory.

- void clearFunctionLevel ()

  Clears function variables, blocks and returned value.

- void **merge** (const State &state)
- void mergeGlobalLevel (const State &state)
- void addGlobalVariable (const llvm::Value &place, Value ∗value)
- void addFunctionVariable (const llvm::Value &place, Value ∗value)
- void **addGlobalBlock** (const llvm::Value &place, Value ∗value)
- void addFunctionBlock (const llvm::Value &place, Value ∗value)

  Adds a value created by alloca to the stack.

- const PlaceValueMap & **getGlobalVariables** () const
- const PlaceValueMap & **getGlobalBlocks** () const
- const PlaceValueMap & **getFunctionVariables** () const
- const PlaceValueMap & **getFunctionBlocks** () const
- Value ∗ findVariable (const llvm::Value &place) const
- Value ∗ findBlock (const llvm::Value &place) const

Public Attributes

- Value ∗ mReturnedValue

  Value returned from function.

Protected Attributes

- PlaceValueMap mGlobalVariables
- PlaceValueMap mGlobalBlocks
- PlaceValueMap mFunctionVariables
- PlaceValueMap mFunctionBlocks

10.19.1 Detailed Description

Includes global variables and heap. Includes function-level memory and variables (=stack).

10.19.2 Member Function Documentation

void Canal::State::addFunctionVariable (const llvm::Value & *place*, Value ∗ *value*)

Adds a register-type value to the stack.

Parameters:

place Represents a place in the program where the function variable is assigned. Usually it is an instance of llvm::Instruction for a result of the instruction. It might also be an instance of llvm::Argument, which represents a function call parameter.

See also:

To add a value created by alloca to the stack, use the method addFunctionBlock.

void Canal::State::addGlobalVariable (const llvm::Value & *place*, Value ∗ *value*)

Parameters:

place Represents a place in the program where the global variable is defined and assigned.

Value ∗ Canal::State::findBlock (const llvm::Value & *place*) const

Search both global and function blocks for a place. If the place is found, the block is returned. Otherwise NULL is returned.

Value ∗ Canal::State::findVariable (const llvm::Value & *place*) const

Search both global and function variables for a place. If the place is found, the variable is returned. Otherwise NULL is returned.

void Canal::State::mergeGlobalLevel (const State & *state*)

Merge only global variables and global memory blocks of the provided state. This is used after a function call, where the modifications of the global state need to be merged to the state of the caller, but its function level state is not relevant.

10.19.3 Member Data Documentation

PlaceValueMap Canal::State::mFunctionBlocks `[protected]`

Nameless memory/values allocated on the stack. The values are referenced either by a pointer in mFunctionVariables or mGlobalVariables, or by another item in mFunctionBlocks or mGlobalBlocks.

The members of the list are owned by this class, so they are deleted in the state destructor.

PlaceValueMap Canal::State::mFunctionVariables `[protected]`

The value pointer does _not_ point to mFunctionBlocks! To connect with a mFunctionBlocks item, create a Pointer object that contains a pointer to a StackBlocks item.

The key (llvm::Value∗) is not owned by this class. It is not deleted. The value (Value∗) memory is owned by this class, so it is deleted in state destructor.

PlaceValueMap Canal::State::mGlobalBlocks `[protected]`

Nameless memory/values allocated on the heap. It's referenced either by a pointer somewhere on a stack, by a global variable, or by another Block or stack Block.

The keys are not owned by this class. They represent the place where the block has been allocated. The values are owned by this class, so they are deleted in the state destructor.

PlaceValueMap Canal::State::mGlobalVariables `[protected]`

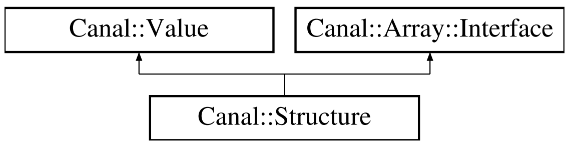
The key (llvm::Value*) is not owned by this class. It is not deleted. The value (Value*) memory is owned by this class, so it is deleted in state destructor.

The documentation for this class was generated from the following files:

- lib/State.h
- lib/State.cpp

10.20 Canal::Structure Class Reference

Inheritance diagram for Canal::Structure::



Public Member Functions

- **Structure** (const Structure &structure)
- virtual Structure $*$ clone () const
- virtual Structure $*$ cloneCleaned () const
- virtual bool operator== (const Value &value) const

  Implementation of Value::operator==().

- virtual void merge (const Value &value)

  Implementation of Value::merge().

- virtual size_t memoryUsage () const

  Implementation of Value::memoryUsage().

- virtual std::string toString () const

  Implementation of Value::toString().

- virtual bool matchesString (const std::string &text, std::string &rationale) const

  Implementation of Value::matchesString().

- virtual std::vector< Value $*$ > getItem (const Value &offset) const

  Implementation of Array::Interface::getItem().

- virtual Value $*$ getItem (uint64_t offset) const

  Implementation of Array::Interface::getItem().

- virtual void setItem (const Value &offset, const Value &value)

  Implementation of Array::Interface::set().

- virtual void setItem (uint64_t offset, const Value &value)

  Implementation of Array::Interface::set().

Public Attributes

- std::vector< Value $*$ > **mMembers**

10.20.1 Member Function Documentation

Structure $*$ Canal::Structure::clone () const `[virtual]`

Implementation of Value::clone(). Covariant return type.
Implements Canal::Value.

Structure ∗ Canal::Structure::cloneCleaned () const [virtual]

Implementation of Value::cloneCleaned(). Covariant return type.

Implements Canal::Value.

The documentation for this class was generated from the following files:

- lib/Structure.h
- lib/Structure.cpp

10.21 Canal::Pointer::Target Class Reference

TODO: Pointers to functions.

`#include <PointerTarget.h>`

Public Types

- enum **Type** {
  Constant, **FunctionBlock**, **FunctionVariable**, **GlobalBlock**,
  GlobalVariable }

Public Member Functions

- Target (Type type, const llvm::Value ∗target, const std::vector< Value ∗ > &offsets, Value ∗numericOffset)
- Target (const Target &target)

  Copy constructor.

- bool **operator==** (const Target &target) const
- bool **operator!=** (const Target &target) const
- void merge (const Target &target)

  Merge another target into this one.

- size_t memoryUsage () const

  Get memory usage (used byte count) of this value.

- std::string toString (SlotTracker &slotTracker) const

  Get a string representation of the target.

- std::vector< Value ∗ > dereference (const State &state) const

Public Attributes

- Type mType

  Type of the target.

- const llvm::Value ∗ mInstruction
- std::vector< Value ∗ > mOffsets
- Value ∗ mNumericOffset

10.21.1 Detailed Description

TODO: Pointers to functions. Pointer target -- where the pointer points to. Pointer can:

- be set to a fixed constant (memory area), such as 0xbaadfood

- point to a heap object (global block)

- point to a stack object (alloca, function block) Pointer can point to some offset in an array.

10.21.2 Constructor & Destructor Documentation

Canal::Pointer::Target::Target (Type *type*, const llvm::Value ∗ *target*, const std::vector< Value ∗ > & *offsets*, Value ∗ *numericOffset*)

Standard constructor.

Parameters:

type Type of the referenced memory.

target Represents the target memory block. If type is Constant, it must be NULL. Otherwise, it must be a valid pointer to an instruction. This is a key to State::mFunctionBlocks, State::mFunctionVariables, State::mGlobalBlocks, or State::mGlobalVariables, depending on the type.

offsets Offsets in the getelementptr style. The provided vector might be empty. The newly created pointer target becomes the owner of the objects in the vector.

numericOffset Numerical offset that is used in addition to the getelementptr style offset and after they have been applied. It might be NULL, which indicates the offset 0. The target becomes the owner of the numerical offset when it's provided. This parameter is mandatory for pointers of Constant type, because it contains the constant.

10.21.3 Member Function Documentation

std::vector< Value ∗ > Canal::Pointer::Target::dereference (const State & *state*) const

Dereference the target in a certain state. Dereferencing might result in multiple Values being returned due to the nature of mOffsets (offsets might include integer ranges). The returned pointers point to the memory owned by State and its abstract domains -- caller must not release the memory.

10.21.4 Member Data Documentation

const llvm::Value∗ Canal::Pointer::Target::mInstruction

Valid when the target represents an anonymous memory block. This is a key to either State::mGlobalBlocks or State::mFunctionBlocks. The referenced llvm::Value instance is owned by the LLVM framework and not by this class.

Value∗ Canal::Pointer::Target::mNumericOffset

An additional numeric offset on the top of mOffsets. The value represents a number of bytes. This class owns the memory. It might be NULL instead of 0.

std::vector<Value∗> Canal::Pointer::Target::mOffsets

Array or struct offsets in the GetElementPtr style. This class owns the memory.

The documentation for this class was generated from the following files:

- lib/PointerTarget.h
- lib/PointerTarget.cpp

10.22 Canal::APIntUtils::UCompare Struct Reference

Public Member Functions

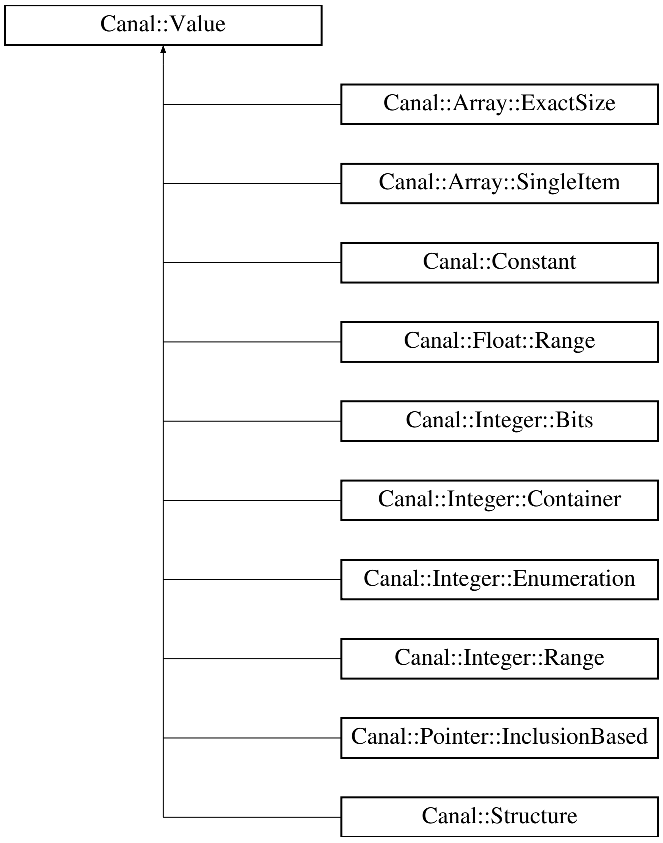
- bool **operator()** (const llvm::APInt &a, const llvm::APInt &b) const

The documentation for this struct was generated from the following file:

- lib/APIntUtils.h

10.23 Canal::Value Class Reference

Inheritance diagram for Canal::Value::



Public Types

- typedef void(Value::∗ **CastOperation**)(const Value &)
- typedef void(Value::∗ **BinaryOperation**)(const Value &, const Value &)
- typedef void(Value::∗ **CmpOperation**)(const Value &, const Value &, llvm::CmpInst::Predicate predicate)

Public Member Functions

- virtual Value ∗ clone () const =0

  Create a copy of this value.

- virtual Value ∗ cloneCleaned () const =0
- virtual bool operator== (const Value &value) const =0
- virtual bool operator!= (const Value &value) const

  Inequality is implemented by calling the equality operator.

- virtual void merge (const Value &value)

  Merge another value into this one.

- virtual size_t memoryUsage () const =0

  Get memory usage (used byte count) of this abstract value.

- virtual std::string toString () const =0

    Create a string representation of the abstract value.

- virtual bool matchesString (const std::string &text, std::string &rationale) const =0
- virtual void add (const Value &a, const Value &b)

    Implementation of instructions operating on values.

- virtual void **fadd** (const Value &a, const Value &b)
- virtual void **sub** (const Value &a, const Value &b)
- virtual void **fsub** (const Value &a, const Value &b)
- virtual void **mul** (const Value &a, const Value &b)
- virtual void **fmul** (const Value &a, const Value &b)
- virtual void udiv (const Value &a, const Value &b)

    Unsigned division.

- virtual void sdiv (const Value &a, const Value &b)

    Signed division.

- virtual void fdiv (const Value &a, const Value &b)

    Floating point division.

- virtual void **urem** (const Value &a, const Value &b)
- virtual void **srem** (const Value &a, const Value &b)
- virtual void **frem** (const Value &a, const Value &b)
- virtual void **shl** (const Value &a, const Value &b)
- virtual void **lshr** (const Value &a, const Value &b)
- virtual void **ashr** (const Value &a, const Value &b)
- virtual void **and_** (const Value &a, const Value &b)
- virtual void **or_** (const Value &a, const Value &b)
- virtual void **xor_** (const Value &a, const Value &b)
- virtual void **icmp** (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
- virtual void **fcmp** (const Value &a, const Value &b, llvm::CmpInst::Predicate predicate)
- virtual void **trunc** (const Value &value)
- virtual void **zext** (const Value &value)
- virtual void **sext** (const Value &value)
- virtual void **fptrunc** (const Value &value)
- virtual void **fpext** (const Value &value)
- virtual void **fptoui** (const Value &value)
- virtual void **fptosi** (const Value &value)
- virtual void **uitofp** (const Value &value)
- virtual void **sitofp** (const Value &value)

10.23.1 Member Function Documentation

void Canal::Value::add (const Value & *a*, const Value & *b*) `[virtual]`

Implementation of instructions operating on values. Load the abstract value state from a string representation.

Parameters:

text The textual representation. It must not contain any text that does not belong to this abstract value state.

Returns:

True if the text has been successfully parsed and the state has been set from the text. False otherwise.

virtual Value∗ Canal::Value::cloneCleaned () const `[pure virtual]`

This is used to obtain instance of the value type and to get an empty value at the same time.

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, Canal::Constant, Canal::Float::Range, Canal::Integer::Bits, Canal::Integer::Container, Canal::Integer::Enumeration, Canal::Integer::Range, Canal::Pointer::InclusionBased and Canal::Structure.

virtual bool Canal::Value::matchesString (const std::string & *text*, std::string & *rationale*) const `[pure virtual]`

Checks if the abstract value internal state matches the text description. Full coverage of the state is not expected, the text can contain just partial information.

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, Canal::Constant, Canal::Float::Range, Canal::Integer::Bits, Canal::Integer::Container, Canal::Integer::Enumeration, Canal::Integer::Range, Canal::Pointer::InclusionBased and Canal::Structure.

virtual bool Canal::Value::operator== (const Value & *value*) const `[pure virtual]`

Implementing this is mandatory. Values are compared while computing the fixed point.

virtual std::string Canal::Value::toString () const `[pure virtual]`

Create a string representation of the abstract value. An idea for different memory interpretation. virtual Value ∗castTo(const llvm::Type ∗itemType, int offset) const = 0;

Implemented in Canal::Array::ExactSize, Canal::Array::SingleItem, Canal::Constant, Canal::Float::Range, Canal::Integer::Bits, Canal::Integer::Container, Canal::Integer::Enumeration, Canal::Integer::Range, Canal::Pointer::InclusionBased and Canal::Structure.

The documentation for this class was generated from the following files:

- lib/Value.h
- lib/Value.cpp

10.24 Canal::VariablePrecisionValue Class Reference

`#include <Value.h>`

Public Member Functions

- virtual bool limitMemoryUsage (size_t size)

10.24.1 Detailed Description

Base class for abstract values that can lower the precision and memory requirements on demand.

10.24.2 Member Function Documentation

bool Canal::VariablePrecisionValue::limitMemoryUsage (size_t *size*) `[virtual]`

Decrease memory usage of this value below the provided size in bytes. Returns true if the memory usage was limited, false when it was not possible.

The documentation for this class was generated from the following files:

- lib/Value.h
- lib/Value.cpp

Bibliography

[1] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[2] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, 1979.

[3] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[4] Brian Albert Davey and Hilary Ann Priestley. Introduction to Lattices and Order. 2nd ed. Cambridge University Press, 2002.

[5] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[6] Antoine Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, 2006.

[7] Antoine Miné. Static Analysis of Run-time Errors in Embedded Critical Parallel C Programs. In *ESOP '11: Proceedings of The 20th European Symposium on Programming*, 2011.

[8] Antoine Miné. Abstract Domains for Bit-Level Machine Integer and Floating-point Operations. In *WING '12: Proceedings of The 4th International Workshop on Invariant Generation*, 2012.

[9] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

Index