1 Introduction of IKOS

1.1 core

1.1.1 number.hpp

This program is a header file that contains definitions related to numbers in the IKOS static analyzer library. It includes various headers related to numbers like bounds, machine integers, signedness, etc. This header file provides essential definitions and inclusions for working with numbers in the IKOS library.

```
1      #include <ikos/core/number/f_number.hpp>  // By zoush99
2      #include <ikos/core/number/machine_int.hpp>
3      #include <ikos/core/number/q_number.hpp>
4      #include <ikos/core/number/z_number.hpp>
```

1.1.2 literal.hpp

This program introduces data structures for literals. The class to represent a literal is classfied to these types: `constant machine integer, constant floating point, constant memory location, null, undefined, machine integer variable, floating point variable, and pointer variable`. Lit is defined as follows,

```
1      using Lit = boost::variant< MachineIntLit,
2      FloatingPointLit,
3      MemoryLocationLit,
4      NullLit,
5      UndefinedLit,
6      MachineIntVarLit,
7      FloatingPointVarLit,
8      PointerVarLit >;
```

The `MachineIntLit` and `FloatingPointLit` are defined as follows. Every type of literal is defined with a value and an operator ==.

```
1      struct MachineIntLit {
2          MachineInt value;
3
4          bool operator==(const MachineIntLit& o) const
5          { return value == o.value; }
6        };
7
8        struct FloatingPointLit {
9          FNumber value;
10
11         bool operator==(const FloatingPointLit& o) const {
12           return value == o.value;
13         }
14       };
```

The program construct a literal as variable literal by some constructors.

```
1      /// \brief Create a machine integer variable literal
2      static Literal machine_int_var(VariableRef v) {
```

```
3       return Literal(Lit(MachineIntVarLit{v}));
4     }
5
6     /// \brief Create a floating point variable literal
7     static Literal floating_point_var(VariableRef v) {
8       return Literal(Lit(FloatingPointVarLit{v}));
9     }
10
11    /// \brief Create a pointer variable literal
12    static Literal pointer_var(VariableRef v) {
13      return Literal(Lit(PointerVarLit{v}));
14    }
```

There are some functions to check whether the literal is a specific type, such as constant machine integer, constant floating point and constant memory location et al..

```
1     /// \brief Return true if the literal is a constant machine integer
2     bool is_machine_int() const {
3       return boost::apply_visitor(IsType< MachineIntLit >(), this->_lit);
4     }
5
6     /// \brief Return true if the literal is a constant floating point
7     bool is_floating_point() const {
8       return boost::apply_visitor(IsType< FloatingPointLit >(), this->_lit);
9     }
10
11    /// \brief Return true if the literal is a constant memory location
12    bool is_memory_location() const {
13      return boost::apply_visitor(IsType< MemoryLocationLit >(), this->_lit);
14    }
```

In addition, a base class for visitors of literals is defined.

```
1     /// \brief Base class for visitors of literals
2     ///
3     /// Visitors should implement the following methods:
4     ///
5     /// R machine_int(const MachineInt&);
6     /// R numeric(const FNumber&);
7     /// R memory_location(MemoryLocationRef);
8     /// R null();
9     /// R undefined();
10    /// R machine_int_var(VariableRef);
11    /// R floating_point_var(VariableRef);
12    /// R pointer_var(VariableRef);
```

1.1.3 linear_expression.hpp

This program defines a data structure for the symbolic manipulation of linear expressions. It includes functionalities to represent and manipulate linear expressions involving variables, constants, addition, subtraction, multiplication, and unary minus. The program also provides an output operator to write a linear expression to a stream. Additionally, it includes specific functionalities like creating constant expressions, variable expressions, and expressions involving constants multiplied by variables. The program uses templates to handle

different number types and variable references. It also includes various helper functions and iterators to work with the terms of the linear expression.

There are two types of class, `VariableExpression` and `LinearExpression`. `VariableExpression` is defined by a `VariableRef`. The coefficient of `VariableExpression` is `1`. `LinearExpression` is defined by `map` and `Number`. A `LinearExpression` contains two components, expression with variables and coefficients as well as constant. The variable expression are defined by maps as follows.

```
1      using VariableExpressionT = VariableExpression< Number, VariableRef >;
```

There are some functions to manipulate LinearExpressions, such as `+=, -=, *=, -` et al.. Programs can add a variable of term to the `LinearExpression`. For example,

```
1      /// \brief Add a variable
2      void add(VariableRef var) { this->add(1, var); }
3
4      /// \brief Add a term cst * var
5      void add(const Number& cst, VariableRef var) {
6        auto it = this->_map.find(var);
7        if (it != this->_map.end()) {
8          Number r = it->second + cst;
9          if (r == 0) {
10           this->_map.erase(it);
11         } else {
12           it->second = r;
13         }
14       } else {
15         if (cst != 0) {
16           this->_map.emplace(var, cst);
17         }
18       }
19     }
```

Besides, programs can generate a `LinearExpression` from multiplication of two components: `VariableExpression` and `Number`.

```
1  template < typename Number, typename VariableRef >
2  inline LinearExpression< Number, VariableRef > operator*(
3      VariableExpression< Number, VariableRef > e, Number n) {
4    return LinearExpression< Number, VariableRef >(std::move(n), e.var());
5  }
6
7  template < typename Number, typename VariableRef >
8  inline LinearExpression< Number, VariableRef > operator*(
9      LinearExpression< Number, VariableRef > e, const Number& n) {
10   e *= n;
11   return e;
12 }
```

To sum up, this program enable other programs to manipulate `LinearExpression` easily.

1.1.4 linear_constraint

This program defines a data structure for the symbolic manipulation of linear expressions. It includes functionalities to represent and manipulate linear expressions involving `variables,`

constants, addition, subtraction, multiplication, and unary minus. The program also provides an output operator to write a linear expression to a stream. Additionally, it includes specific functionalities like creating constant expressions, variable expressions, and expressions involving constants multiplied by variables. The program uses templates to handle different number types and variable references. It also includes various helper functions and iterators to work with the terms of the linear expression.

The program defines a class called LinearConstraint that represents a linear constraint. It has a template parameter for the number type and a template parameter for the variable reference type. The class has a member variable _expr of type LinearExpression that represents the linear expression of the constraint. It also has a member variable _kind of type Kind that represents the kind of linear constraint (Equality, Disequation, or Inequality).

The program also defines a class called LinearConstraintSystem that represents a set of linear constraints. It has a member variable _csts of type std::vector< LinearConstraintT > that stores the linear constraints. The class provides a method variables that returns a set of variables present in the system.

Overall, this program provides a data structure and functionalities for symbolic manipulation of linear expressions and linear constraints.

The internal expression of LinearConstraint is as follows. A LinearConstraint consists of two components, LinearExpression and Operator, the Operator express the relations of LinearExpression and zero.

```
public:
  using VariableExpressionT = VariableExpression< Number, VariableRef >;
  using LinearExpressionT = LinearExpression< Number, VariableRef >;

public:
  /// \brief Kind of linear constraint
  enum Kind {
    Equality,    // ==
    Disequation, // !=
    Inequality,  // <=
  };

private:
  LinearExpressionT _expr;
  Kind _kind;
```

There are some functions to construct simple constraints, such as tautology and contradiction.

```
  /// \brief Create the tautology 0 == 0
  static LinearConstraint tautology() {
    return LinearConstraint(LinearExpressionT(), Equality);
  }

  /// \brief Create the contradiction 0 != 0
  static LinearConstraint contradiction() {
    return LinearConstraint(LinearExpressionT(), Disequation);
  }
```

In addition, there are many auxiliary functions that can help linearexpression convert to linearConstraint.Such as $\leq, <, =, >, \geq$. Besides LinearExpression, we can use

`VariableExpression`. Let's take an example below.

```
template < typename Number, typename VariableRef >
inline LinearConstraint< Number, VariableRef > operator<=(
    LinearExpression< Number, VariableRef > e, const Number& n) {
  return LinearConstraint<
      Number,
      VariableRef >(std::move(e) - n,
                    LinearConstraint< Number, VariableRef >::Inequality);
}

template < typename Number, typename VariableRef >
inline LinearConstraint< Number, VariableRef > operator<=(
    VariableExpression< Number, VariableRef > x,
    VariableExpression< Number, VariableRef > y) {
  return LinearConstraint<
      Number,
      VariableRef >(std::move(x) - std::move(y),
                    LinearConstraint< Number, VariableRef >::Inequality);
}

template < typename Number, typename VariableRef >
inline LinearConstraint< Number, VariableRef > operator<=(
    LinearExpression< Number, VariableRef > x,
    const LinearExpression< Number, VariableRef >& y) {
  return LinearConstraint<
      Number,
      VariableRef >(std::move(x) - y,
                    LinearConstraint< Number, VariableRef >::Inequality);
}
```

In order to better process the restrictions in the future, in this file, the `linearconstraintsystem` class can be constructed, which can be uniformly collected to be collected for efficiency.The constructor process is more delicate. Use `vector` to store constraints, and use `iterator` to traverse the constraints. Programs can use function `add` to add constraints or constraint system.

```
  public:
  using LinearConstraintT = LinearConstraint< Number, VariableRef >;

private:
  using Constraints = std::vector< LinearConstraintT >;

public:
  using Iterator = typename Constraints::iterator;
  using ConstIterator = typename Constraints::const_iterator;

private:
  Constraints _csts;
```

1.2 exception.hpp

This is a basic definition of IKOS abnormal processing. It can print expannatory messages and exit the program. In actual use, this program does not participate in the core function.

1.3 value

1.3.1 lifetime.hpp

This file defines a class `Lifetime` that represents a lifetime abstract value. The class is a subclass of `core::AbstractDomain< Lifetime >` and has an enum `Kind` with values `BottomKind, AllocatedKind, DeallocatedKind` and `TopKind`. The class has a private member `_kind` of type Kind initialized to `TopKind`. The class has a private constructor that takes a Kind parameter and sets the `_kind` member. The purpose of this class is to represent and manipulate lifetime abstract values in the IKOSanalysis framework. The class provides methods to check if a lifetime is bottom, allocated, or deallocated, and to join and meet two lifetime values. The class also provides a method to compute the widening of a lifetime value.

```
1    enum Kind : unsigned {
2      BottomKind = 0,
3      AllocatedKind = 1,
4      DeallocatedKind = 2,
5      TopKind = 3
6    };
```

This program can be used to check the survival cycle of variables, which can analyze the reliability of the program.This program is the definition of the abstraction value of the lifetime of variables.

1.3.2 nullity.hpp

This program describes the abstract value of variables. There are four possible situations: `bottomKind, nullKind, nonnullkind` and `Topkind`.This program is used to check the vacancy of the variable and analyze the reliability of the program.It contains various methods of abstract value operation.

```
1    enum Kind : unsigned {
2      BottomKind = 0,
3      NullKind = 1,
4      NonNullKind = 2,
5      TopKind = 3
6    };
```

1.3.3 uinitilized.hpp

This program describes the abstract value of the initialization of variables. There are four abstract values, it is either top, bottom, initialized or uninitialized. This program is used to check the initialization of the variable and analyze the reliability of the program. There contains various methods of abstract value operation.

```
enum Kind : unsigned {
  BottomKind = 0,
  InitializedKind = 1,
  UninitializedKind = 2,
  TopKind = 3
};
```

1.3.4 machine_int/interval.hpp

This file is a definition of Machine Integer Interval. Its Member Variables include `_LB` and `_ub`. Bounds of Machine Integer Interval.

The representation of `Topkind` and `BottomKind` in this program is different from expressed in the numerical abstract domain.

```
/// \brief Create the top interval for the given bit-width and signedness
static Interval top(uint64_t bit_width, Signedness sign) {
  return Interval(MachineInt::min(bit_width, sign),
                  MachineInt::max(bit_width, sign));
}

/// \brief Create the bottom interval for the given bit-width and signedness
static Interval bottom(uint64_t bit_width, Signedness sign) {
  return Interval(MachineInt::max(bit_width, sign),
                  MachineInt::min(bit_width, sign));
}

/// \brief Create the interval [n, n]
explicit Interval(const MachineInt& n) : _lb(n), _ub(n) {}
```

There are many functions to make this file on the interval abstract valueOperations, such as `widening`, `norrowing`, `join`, and `meet`. In addition, this file defines how the abstraction value of unary and binary operations is operated. For example,

```
/// \brief Truncate the machine integer interval to the given bit width
Interval trunc(uint64_t bit_width) const {
  ikos_assert(this->bit_width() > bit_width);
  if (this->is_bottom()) {
    return bottom(bit_width, this->sign());
  } else if (this->_lb == this->_ub) {
    return Interval(this->_lb.trunc(bit_width));
  } else {
    // For unsigned integers, check that the first (m-n) bits match.
    // For signed integers, check the first (m-n+1) bits: if the result sign
    // bit is different, return top.
    MachineInt n(this->sign() == Unsigned ? bit_width : (bit_width - 1),
                 this->bit_width(),
                 this->sign());
    if (lshr(this->_lb, n) == lshr(this->_ub, n)) {
      // Identical upper bits
      return Interval(this->_lb.trunc(bit_width), this->_ub.trunc(bit_width));
    } else {
      return Interval::top(bit_width, this->sign());
    }
```

```
    }
  }

  /// \brief Extend the machine integer interval to the given bit width
  Interval ext(uint64_t bit_width) const {
    ikos_assert(this->bit_width() < bit_width);
    if (this->is_bottom()) {
      return bottom(bit_width, this->sign());
    } else {
      return Interval(this->_lb.ext(bit_width), this->_ub.ext(bit_width));
    }
  }

  /// \brief Addition with wrapping
inline Interval add(const Interval& lhs, const Interval& rhs) {
  assert_compatible(lhs, rhs);
  if (lhs.is_bottom()) {
    return lhs;
  } else if (rhs.is_bottom()) {
    return rhs;
  } else {
    return Interval::from_z_interval(lhs.to_z_interval() + rhs.to_z_interval(),
                                     lhs.bit_width(),
                                     lhs.sign(),
                                     Interval::WrapTag{});
  }
}
```

There are also methods for converting Machine Integer Interval to Zinterval, such as

```
  /// \brief Return the machine integer interval as a ZInterval
  ZInterval to_z_interval() const {
    if (this->is_bottom()) {
      return ZInterval::bottom();
    } else {
      return ZInterval(ZBound(this->_lb.to_z_number()),
                       ZBound(this->_ub.to_z_number()));
    }
  }

  struct WrapTag {};
  struct TruncTag {};

  /// \brief Convert a ZInterval to a machine integer interval
  static Interval from_z_interval(const ZInterval& i,
                                  uint64_t bit_width,
                                  Signedness sign,
                                  WrapTag) {
    if (i.is_bottom()) {
      return bottom(bit_width, sign);
    }

    const ZBound& lb = i.lb();
    const ZBound& ub = i.ub();

```

```
26      if (lb.is_infinite() || ub.is_infinite()) {
27        return top(bit_width, sign);
28      }
29
30      ZNumber z_lb = *lb.number();
31      ZNumber z_ub = *ub.number();
32
33      MachineInt i_lb(z_lb, bit_width, sign);
34      MachineInt i_ub(z_ub, bit_width, sign);
35
36      if (i_ub.to_z_number() - i_lb.to_z_number() == z_ub - z_lb) {
37        return Interval(i_lb, i_ub);
38      }
39
40      return top(bit_width, sign);
41    }
```

1.3.5 machine_int/congruence.hpp

This file describes the operation of the abstract value of the Congruence abstract domain. There are simple operations and `join, meet, widening` and `narrowing`.

1.3.6 machine_int/interval_congruence.hpp

This file design the intervalcongruence class of the Machine Integer. This is Implement as a Pair of interval and connected value. This abstract value type combines the `Interval` and `Congruence` abstraction value can represent the interval information of the variable, and it can represent TongyuRelationship.

1.3.7 machine_int/constant.hpp

This program describes the abstract value of the constant. This program is used to check the constant value and analyze the reliability of the program. There contains various methods of abstract value operation.

```
1  private:
2    enum Kind { BottomKind, TopKind, IntegerKind };
3
4  private:
5    Kind _kind;
6    MachineInt _n;
7
8  private:
9    struct TopTag {};
10   struct BottomTag {};
```

This represents the type of the constant abstraction of the machine integer. There are three cases: texttt bottomkind, texttt Topkind and texttt integerKind, respectively, respectively, respectively `bottomKind`, `Topkind` And `IntegerKind` type of abstract value.

We can construct top constant, bottom constant and machine integer constant for the given bit-width and signedness.

```
  /// \brief Create the top constant for the given bit-width and signedness
  static Constant top(uint64_t bit_width, Signedness sign) {
    return Constant(TopTag{}, bit_width, sign);
  }

  /// \brief Create the bottom constant for the given bit-width and signedness
  static Constant bottom(uint64_t bit_width, Signedness sign) {
    return Constant(BottomTag{}, bit_width, sign);
  }

  /// \brief Create the constant n
  explicit Constant(MachineInt n) : _kind(IntegerKind), _n(std::move(n)) {}
```

There are many other functions to describe different operations of constant abstract values, such as `add, sub, mul` and `div`. When performing abstract value operations, we judge whether it overflows, so the final analysis results are obtained.

1.3.8 numeric/interval.hpp

Similar to the interval abstract value of the integer type of the machine, the member variables of the abstract value in the abstract domain of the numerical interval are shown below.

```
public:
  using NumberT = Number;
  using BoundT = Bound< Number >;

private:
  // Lower bound
  BoundT _lb;

  // Upper bound
  BoundT _ub;

  // Invariant: is_bottom() <=> _lb = 1 && _ub = 0

private:
  struct TopTag {};
  struct BottomTag {};
```

This uses the `Bound` class as the upper and lower bounds of the interval. The specific details can view the definition of the `Bound` class. Besides, there are many functions to construct an interval, for example,

```
  /// \brief Create the top interval [-oo, +oo]
  explicit Interval(TopTag)
      : _lb(BoundT::minus_infinity()), _ub(BoundT::plus_infinity()) {}

  /// \brief Create the bottom interval
  explicit Interval(BottomTag) : _lb(1), _ub(0) {}

public:
  /// \brief Create the interval [-oo, +oo]
  static Interval top() { return Interval(TopTag{}); }

```

```
12    /// \brief Create the bottom interval
13    static Interval bottom() { return Interval(BottomTag{}); }
14
15    /// \brief Create the interval [n, n]
16    template <
17        typename T,
18        class = std::enable_if_t< IsSupportedIntegralOrFloat< T >::value > >
19    explicit Interval(T n) : _lb(n), _ub(n) {}
20
21    /// \brief Create the interval [n, n]
22    explicit Interval(const Number& n) : _lb(n), _ub(n) {}
23
24    /// \brief Create the interval [b, b]
25    explicit Interval(const BoundT& b) : _lb(b), _ub(b) {
26      ikos_assert(!b.is_infinite());
27    }
```

It is worth noting that here only support the original machine integer type and the converted `ZNumber` integer. If you need to support the analysis of floating-point types, you need to modify the part of this file. As follows.

```
1    inline Interval< FNumber > operator/(const Interval< FNumber >& lhs,
2                                         const Interval< FNumber >& rhs)
```

1.3.9 numeric/congruence.hpp

This file represents the abstract value operation of the `Congruence` abstract domain in the numerical abstract domain. This function mainly processs the analysis of the program in the program of `FNumber` type. Its definition is as follows:

```
1  class Congruence< ZNumber > final
2      : public core::AbstractDomain< Congruence< ZNumber > > {
3  public:
4    using NumberT = ZNumber;
5
6  private:
7    bool _is_bottom;
8    ZNumber _a;
9    ZNumber _b;
10
11   // Invariant: !_is_bottom => _a >= 0
12   // Invariant: !_is_bottom && _a != 0 => 0 <= _b < _a
13   // Invariant: _is_bottom => _a == _b == 0
14
15 private:
16   struct TopTag {};
17   struct BottomTag {};
18   };
```

1.3.10 numeric/interval_congruence.hpp

This file is the `intervalcongruence` abstract domain, which is similar to the intervalcongruence in the machine integer introduced earlier.

1.3.11 numeric/constant.hpp

This file is similar to the `constant` in the previously introduced machine integer type, but the difference is that the definition of generic `Number` is used here.This allows it to support qualified data types, such as `ZNumber, QNumber` and `FNumber`.

```
template < typename Number >
class Constant final : public core::AbstractDomain< Constant< Number > > {
public:
  using NumberT = Number;

private:
  enum Kind { BottomKind, TopKind, NumberKind };

private:
  Kind _kind;
  Number _n;

private:
  struct TopTag {};
  struct BottomTag {};

  /// \brief Create the top constant
  explicit Constant(TopTag) : _kind(TopKind) {}

  /// \brief Create the bottom constant
  explicit Constant(BottomTag) : _kind(BottomKind) {}

public:
  /// \brief Create the top constant
  static Constant top() { return Constant(TopTag{}); }

  /// \brief Create the bottom constant
  static Constant bottom() { return Constant(BottomTag{}); }

  /// \brief Create the constant n
  template <
      typename T,
      class = std::enable_if_t< IsSupportedIntegralOrFloat< T >::value > >
  explicit Constant(T n) : _kind(NumberKind), _n(n) {}

  /// \brief Create the constant n
  explicit Constant(Number n) : _kind(NumberKind), _n(std::move(n)) {}
};
```

What confuse me is that unary minus is operator -. But in machine int type, the unary operator is "trunc, ext, ...".What is going on, how should I modify the `unary operations` I have ignored before?

```
/// \brief Unary minus
Constant operator-() const {
    if (this->is_number()) {
        return Constant(-this->_n);
      } else {
        return *this;
```

```
7       }
8   }
```

In addition, you should also pay attention to the processing of divisions-by-zero errors, as shown below, if the divisor is zero, the abstract value of analysis results are set to `bottomKind`.

```
1  /// \brief Divide constants
2  template < typename Number >
3  inline Constant< Number > operator/(const Constant< Number >& lhs,
4                                      const Constant< Number >& rhs) {
5    using ConstantT = Constant< Number >;
6
7    if (lhs.is_bottom() || rhs.is_bottom()) {
8      return ConstantT::bottom();
9    } else if (rhs.is_zero()) {
10     return ConstantT::bottom();
11   } else if (lhs.is_zero()) {
12     return ConstantT(0);
13   } else if (lhs.is_top() || rhs.is_top()) {
14     return ConstantT::top();
15   } else {
16     return ConstantT(lhs._n / rhs._n);
17   }
18 }
```

1.3.12 numeric/gauge.hpp

This abstract domain is designed by IKOS itself and analyzes the nested loop of the integer type in programs. If you want to know more, you can study their paper: **The Gauge Domain: Scalable Analysis of Linear Inequality Invariants**.

1.3.13 pointer/pointer.hpp

The `PointerAbsValue` class contains the following members:

- `_uninitialized`: An instance of the Uninitialized class representing the uninitialized state of the pointer.
- `_nullity`: An instance of the Nullity class representing the nullity of the pointer.
- `_points_to`: An instance of the PointsToSetT class representing the set of memory locations that the pointer points to.
- `_offset`: An instance of the MachineIntInterval class representing the offset of the pointer.

The class provides various constructors to create instances of PointerAbsValue with different initial values. It also provides copy and move constructors and assignment operators.

The class overrides several virtual functions from the AbstractValue base class, such as `name(), is_bottom(), is_top(), equals(), leq()`, and others. These functions are used for comparison and manipulation of the pointer abstract values.

The class also provides member functions for reducing the abstract value, such as `reduce()`, `set_to_bottom()`, `set_to_null()`, `set_to_uninitialized()`, and others. These functions are used to simplify the abstract value based on certain conditions.

Overall, this program represents a part of the implementation of the abstract interpretation of pointers.

1.3.14 pointer/pointer_set.hpp

You need to take a closer look at these two programs. They are files for the abstract value operation.

1.3.15 pointer/pointer_to_set.hpp

1.4 semantic

1.4.1 variable.hpp

This file describes the requirements of the type of `VariableRef`:

```
1  /// \brief Check if the given type meets the requirements for Variable types
2  ///
3  /// Requirements:
4  ///
5  /// VariableRef has a noexcept copy constructor
6  /// VariableRef has a noexcept move constructor
7  /// VariableRef has a noexcept copy assignment operator
8  /// VariableRef has a noexcept move assignment operator
9  ///
10 /// bool operator==(VariableRef x, VariableRef y)
11 ///   Return true if x and y refers to the same variable
12 ///
13 /// bool operator<(VariableRef x, VariableRef y)
14 ///   Return true if index(x) < index(y)
15 ///
16 /// VariableRef implements DumpableTraits
17 ///
18 /// VariableRef implements IndexableTraits
19 ///
20 /// The VariableRef type should be cheap to copy.
```

1.4.2 memory_location.hpp

This file describes the requirements of the type of `MemoryLocationRef`:

```
1    /// \brief Check if the given type meets the requirements for memory location
2    /// types
3    ///
4    /// Requirements:
5    ///
6    /// MemoryLocationRef has a noexcept copy constructor
7    /// MemoryLocationRef has a noexcept move constructor
8    /// MemoryLocationRef has a noexcept copy assignment operator
```

```
 9    /// MemoryLocationRef has a noexcept move assignment operator
10    ///
11    /// bool operator==(MemoryLocationRef x, MemoryLocationRef y)
12    ///   Return true if x and y refers to the same memory location
13    ///
14    /// bool operator<(MemoryLocationref x, MemoryLocationref y)
15    ///   Return true if index(x) < index(y)
16    ///
17    /// MemoryLocationRef implements DumpableTraits
18    ///
19    /// MemoryLocationRef implements IndexableTraits
20    ///
21    /// The MemoryLocationRef type should be cheap to copy.
```

1.4.3 indexable.hpp

1.4.4 graph.hpp

1.4.5 dumpable.hpp