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.

```
#include <ikos/core/number/f_number.hpp>  // By zoush99
#include <ikos/core/number/machine_int.hpp>
#include <ikos/core/number/q_number.hpp>
#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,

```
using Lit = boost::variant< MachineIntLit,
FloatingPointLit,
MemoryLocationLit,
NullLit,
UndefinedLit,
MachineIntVarLit,
FloatingPointVarLit,
PointerVarLit >;
```

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

```
struct MachineIntLit {
    MachineInt value;

    bool operator==(const MachineIntLit& o) const
    { return value == o.value; }
  };

  struct FloatingPointLit {
    FNumber value;

    bool operator==(const FloatingPointLit& o) const {
      return value == o.value;
    }
  };
```

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

```
/// \brief Create a machine integer variable literal
static Literal machine_int_var(VariableRef v) {
  return Literal(Lit(MachineIntVarLit{v}));
}

/// \brief Create a floating point variable literal
static Literal floating_point_var(VariableRef v) {
  return Literal(Lit(FloatingPointVarLit{v}));
}

/// \brief Create a pointer variable literal
static Literal pointer_var(VariableRef v) {
  return Literal(Lit(PointerVarLit{v}));
}
```

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..

```
  /// \brief Return true if the literal is a constant machine integer
  bool is_machine_int() const {
    return boost::apply_visitor(IsType< MachineIntLit >(), this->_lit);
  }

  /// \brief Return true if the literal is a constant floating point
  bool is_floating_point() const {
    return boost::apply_visitor(IsType< FloatingPointLit >(), this->_lit);
  }

  /// \brief Return true if the literal is a constant memory location
  bool is_memory_location() const {
    return boost::apply_visitor(IsType< MemoryLocationLit >(), this->_lit);
  }
```

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

```
  /// \brief Base class for visitors of literals
  ///
  /// Visitors should implement the following methods:
  ///
  /// R machine_int(const MachineInt&);
  /// R numeric(const FNumber&);
  /// R memory_location(MemoryLocationRef);
  /// R null();
  /// R undefined();
  /// R machine_int_var(VariableRef);
  /// R floating_point_var(VariableRef);
  /// 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.

```
    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,

```
    /// \brief Add a variable
    void add(VariableRef var) { this->add(1, var); }

    /// \brief Add a term cst * var
    void add(const Number& cst, VariableRef var) {
      auto it = this->_map.find(var);
      if (it != this->_map.end()) {
        Number r = it->second + cst;
        if (r == 0) {
          this->_map.erase(it);
        } else {
          it->second = r;
        }
      } else {
        if (cst != 0) {
          this->_map.emplace(var, cst);
        }
      }
    }
```

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

```
template < typename Number, typename VariableRef >
inline LinearExpression< Number, VariableRef > operator*(
    VariableExpression< Number, VariableRef > e, Number n) {
  return LinearExpression< Number, VariableRef >(std::move(n), e.var());
}

template < typename Number, typename VariableRef >
inline LinearExpression< Number, VariableRef > operator*(
    LinearExpression< Number, VariableRef > e, const Number& n) {
  e *= n;
  return e;
}
```

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`, $wecanuse$`VariableExpression`.

```
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;
```