

# 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 classified 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.

```
1 public:
2     using VariableExpressionT = VariableExpression< Number, VariableRef >;
3     using LinearExpressionT = LinearExpression< Number, VariableRef >;
4
5 public:
6     /// \brief Kind of linear constraint
7     enum Kind {
8         Equality,      // ==
9         Disequation,   // !=
10        Inequality,    // <=
11    };
12
13 private:
14     LinearExpressionT _expr;
15     Kind _kind;
```