Expression Normalization for Reverse Computation

In the Rose, there is only one intermediate representation which is AST. To facilitate our transformations, we need another form of IR, which will be generated by this expression normalization.

In C/C++ source code, an expression can be very complicated, which is hard to transform and analyze. In compiler area, the source code is transformed into three-address code as IR. Since the Rose is a source-to-source compiler, transforming source to three-address code hurts readability very much. To aid our event reverse, we require that between sequence points, there is only one variable which is modified. SSA form analysis also benefits from this normalization.

There are three steps in expression normalization.

First, split the assignment (include prefix or postfix `++/--`) expressions into comma operator expression, except that whose value is not used at all. The left hand side operand of comma operator is the original expression, and the right hand side operand is the return value which does not modify any value. After this transformation, the return value is not changed. For example, `++i` is transformed as `(++i, i)`, `i = j` is transformed as `(i = j, i)`. The postfix increment/decrement operators should be treated carefully. We bring a temporary variable here to make a correct transformation. Then `i++` is transformed as `(t = i, ++i, t)`.

In this step, we also transform the logical and/or operators into conditional operators, so that we can make use of the transformations on conditional operator expression. The rules are

```
a && b    ==>    a ? b : false
a || b    ==>    a ? true : b
```

Second, hoist the comma operator from inside to outside. There are four rules for three different kinds of expressions (which have one, two and three operands separately) to complete this transformation.

```
a + (b, c)          ==>       (b, a + c)
(a, b) + c          ==>       (a, b + c)
!(a, b)             ==>       (a, !b)
(a, b) ? c : d      ==>       (a, b ? c : d)
```

The + operator above represents a binary operator without specified evaluation order of its two operands. Note that in C++, only comma operator, logical and/or operator and conditional operator have specified evaluation order. So + above cannot be comma or logical and/or operators. The ! operator represents a unary operator. Note that the `sizeof` operator is not a unary operator.

In this step, we also hoist the conditional operator. The rules are as following.

```
a + (b ? c : d)          ==>       b ? (a + c) : (a + d)
(a ? b : c) + d          ==>       a ? (b + d) : (c + d)
!(a ? b : c)             ==>       a ? !b : !c
(a ? b : c) ? d : e      ==>       a ? (b ? d : e) : (c ? d : e)
```

Note that we require that there is no unspecified behavior in any expression. The C++ standard requires that “between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression, and the prior value shall be accessed only to determine the value to be stored”. Otherwise the behavior is unspecified. For example, the following expression

```
j = ++i + ++i
```

will be transformed as

```
++i, ++i, j = i + i
```

Those two expressions may produce the different values. Therefore, we forbid this unspecified behavior and will detect it in our language restriction check.

The last step separate comma operator expressions into statements to improve readability. During our normalization, lots of comma operator expressions may be generated which makes code ugly. A simple case is a comma operator expression in an expression statement:

```
++i, j = i;
```

which is separated into two statements:

```
++i;
j = i;
```

A little more complicated cases are those comma operator expressions which are in other statements, like if condition, return statement, etc. We take care all kinds of such situations.

Note that we cannot avoid processing comma operator expression in our reverser since it may appear in a conditional expression, unless we can transform conditional expressions into if statements.