Construction of a PAG control-flow graph from a ROSE abstract syntax tree

Gergő Bárány, e0026139@student.tuwien.ac.at

October 16, 2007

1 Introduction

The analyzers generated by PAG require the program under analysis to be represented as an explicit control-flow graph (CFG). The frontend used by ROSE represents whole programs as abstract syntax trees (ASTs). For programs represented in ROSE's intermediate representation, a CFG must therefore explicitly be computed. This document describes a concrete implementation of this computation. The information given is partly generally applicable to PAG, but mostly specific to our code.

As the ROSE ASTs closely match the original source code, they contain semantic ambiguities; for instance, the C++ standard does not prescribe the order of evaluation of function arguments, thus the control flow inside a function call expression is not completely determined. Because of the constraints posed on the CFG by PAG, the transforming code must in such cases choose some fixed control flow. That is, the transformation chooses one of possibly several different semantics, which might be different from the semantics chosen by a given compiler.

2 Structure of the CFG

2.1 General structure

The CFG consists of *procedures* (which we might also call functions), which in turn consist of *basic blocks*, each of which may contain one or more *statements*. In our implementation, however, each basic block contains exactly

one statement. Therefore this document might sloppily use the terms 'block', 'node' and 'statement' almost interchangeably. The statements in the CFG are partly the statements that occur in the original source code, partly transformed versions of these statements, and partly special statements that do not have an explicit representation in the source code.

Blocks are connected by directed *edges*, each of which has a certain *edge type* (which can be used for pattern matching in the PAG analysis specification). The type of most edges is `normal_edge`. Blocks may in general have several successors and several predecessors (but non-branching statements will not have more than one successor). There is never more than one edge from one block to another.

2.2 Procedures and variable scope

Procedures correspond to the functions (also member functions, including constructors, destructors and overloaded operators) in the C++ source code. Each procedure has an *entry* or *start* node marked by the statement

```
FunctionEntry ( funcname:c_str )
```

giving access to the name of the procedure, and an *exit* or *end* node which is marked by

```
FunctionExit ( funcname:c_str, vars:VariableSymbolNT* )
```

containing also the name of the procedure and a list of variables local to this function. (The intention of the latter being that these local variables are irrelevant outside of this function, thus the corresponding analysis information can be killed when the analysis reaches the function's exit node.)

There is no explicit representation of compound statements ('blocks' of C++ code), variable scopes are represented instead. Variable declarations occur in the CFG as:

```
DeclareStmt ( var:VariableSymbolNT, type:Type )
```

Initialization of a variable is represented as an assignment to that variable after the `DeclareStmt`. Where local variables go out of scope at the end of a compound statement, this is marked by

```
UndeclareStmt ( vars:VariableSymbolNT* )
```

2.3 Control-flow statements

Branching constructs are in general represented in the natural way. An exception are **for** loops, which are always transformed from the general form

```
for (initializations; condition; increment)
{
    body
}
```

into the equivalent of

```
initializations;
while (condition)
{
    body
    increment;
}
```

If the body contains **continue** statements, their outgoing edges are connected to the block representing the (beginning of) the increment expression statement. Loop heads and **if** statements use the edge types `true_edge` and `false_edge` to represent the two paths that can be taken.

2.4 Short-circuit operators

The logical operators && and || as well as the ternary operator ?: are special in that their operands must be evaluated in a certain order, and not necessarily all operands are evaluated. This must be reflected in the control-flow graph. A special statement

```
LogicalIf ( condition:Expression )
```

is used for this purpose, which has the same semantics as a normal **if** statement. It is introduced into the CFG in conjunction with temporary variables; the names of these always start with a $ sign. The transformation is designed such that each of these temporaries is only read at one point in the program; it is irrelevant afterwards, the corresponding analysis information can be killed if a temporary variable is evaluated.

Consider a statement S containing the subexpression A && B somewhere; denote this by S[A && B] (abusing array subscript syntax for want of a better representation). The code

```
S[A && B];
```

is transformed to the equivalent of

```
LogicalIf (A)
{
    $logical_42 = B;
}
else
{
    $logical_42 = 0;
}
S[$logical_42];
```

This first evaluates A; if the result is true, B is evaluated and the temporary variable set to its value. Otherwise, since A was false, the overall result is false. Thus the temporary variable is nonzero iff A && B evaluates to true. (The CFG should enforce that the logical variable only takes one of the values **true** or **false**. This is not implemented yet.) The statement S[$logical_42] is meant to represent that inside the statement the occurrence of the logical expression is replaced by a reference to the temporary variable.

Expressions using the || operator are transformed in an analogous way, and

```
S[(A ? B : C)];
```

as if it had been written

```
LogicalIf (A)
{
    $logical_37 = B;
}
else
{
    $logical_37 = C;
}
S[$logical_37];
```

These transformations apply recursively for nested logical expressions; note that the resulting code does not contain the original operator at all.

The number in the name of the temporary variable varies, of course, and you shouldn't rely on the fact that the name of the variable is of this exact

form. You may, however, safely assume that it will always start with the dollar sign.

Finally, while the comma operator forces order of evaluation, it does not short-circuit. Therefore it is not treated specially in the CFG, the correct order of evaluation of its arguments must be considered in the analysis specification.

2.5 Function calls

Function calls require somewhat complicated code because while PAG has support for the concept of procedures and calls between them, it does not provide for any way to perform passing of argument and return values. The approach taken to model these is therefore to pass arguments by assigning the values of a function call expression's argument expressions to (conceptually) global temporary variables, and similarly to pass the return value back via such a temporary variable.

That is, the statement

```
S[func(A, B)];
```

is treated as if it were written roughly like

```
$func$arg_0 = A;
$func$arg_1 = B;
func();
$func$return_84 = $func$return;
S[$func$return_84];
```

There are many things to note here. The variables associated with a function call contain the function's name between dollar signs, but there are three different numbering schemes: The variables for the argument expressions are always numbered from 0, these same names are used at every site where this function is called. There are 'return' variables without numbers and there are return variables with unique numbers for each call. As always with temporaries, the exact name should not matter for the analysis, and the temporaries can be killed at the point they are read.

In reality, the assignments shown above are not really normal assignments but special statements. The assignments of argument expressions to argument variables, and the assignment of the general return variable to the the return variable specific to this call site are denoted, respectively, by:

```
ArgumentAssignment ( lhs:Expression, rhs:Expression )
ReturnAssignment ( lhs:VariableSymbolNT,
                   rhs:VariableSymbolNT )
```

The special statement

```
ParamAssignment ( lhs:VariableSymbolNT,
                  rhs:VariableSymbolNT )
```

is inserted at the beginning of each procedure for each parameter. This assigns the argument variables to the formal parameters. All three of these special assignment statements are semantically simple assignments which just have special names.

The actual call to the function is modelled by a pair of special statements:

```
FunctionCall ( funcname:c_str )
FunctionReturn ( funcname:c_str )
```

The `ArgumentAssignment` nodes are placed before the call node, while the `ReturnAssignment` is after the return node. An edge of type `local_edge` connects the call to the return node; additionally, there is an edge of type `call_edge` to the entry node of the called function, and an edge of type `return_edge` from the exit node of the called function to the return node. These edges make it possible to propagate analysis information to the called function and back.

Every **return** statement in a function is represented by assigning the expression in the return statement (if any) to the function's return variable and an immediate jump to the function's exit node. This bypasses the undeclare statements in the enclosing compound statements, which is not good and will be fixed some time.

Calls to overloaded functions are resolved statically. Default function arguments are inserted as `ArgumentAssignment`s if not explicitly present in the call. Functions without known implementations, either because only a declaration but not a definition is known or because they go through function pointers, are represented by a single node of type:

```
ExternalCall ( type:Type )
```

Note that such calls can at the very least arbitrarily change all global variables, and potentially any local variable whose address was ever taken. Thus parts of the analysis information have to be eliminated when such nodes are encountered.

2.6 Member function calls

Member functions are treated as normal function calls, but with a special implicit argument for the **this** pointer. The address of the object on which the member function is invoked is assigned to this variable inside the called function using the `ArgumentAssignment/ParamAssignment` mechanism.

If a function call is virtual, there are `call_edge`s from the call node to the entry node of every potential implementation of the called member function. Virtual calls to overloaded functions are not yet handled correctly (too many potential implementations for the function are identified; thus analysis will be safe, but less exact).

The use of an overloaded operator is treated as a member function call to an appropriately named function.

2.7 Constructors and destructors

Constructor calls are handled like member function calls, the **this** pointer being initialized either with the **new** expression or the address of the object being constructed. The constructors of superclasses are called automatically, if they are not explicitly called in the source code.

There is no support for copy constructors yet. Overloaded constructors are not yet handled correctly.

Destructors are also called similarly to normal member functions, virtual destructors are also supported. If a destructor was invoked because of a **delete** statement, that statement appears in the CFG after the return from the destructor. Destructors are called automatically for objects of class type that go out of scope.

The two special statements

```
ConstructorCall ( name:c_str, type:Type )
DestructorCall ( name:c_str, type:Type )
```

are used to denote calls to constructors and destructors whose implementation is not known. The type referred to is the class type to which the called constructor or destructor belongs.