*Copyright (c) Microsoft Open Technologies, Inc.  All Rights Reserved.  Licensed under the Apache License, Version 2.0.  See License.txt in the project root for license information.*

Flow Analysis in the Roslyn C# Compiler

Revisions:

Neal Gafter 2011-02-28 initial draft  
Neal Gafter 2011-05-18 update

*This document is a bit out of date. We separated control-flow analysis and data-flow analysis and moved most of the code into a common base class.*

# Overview

The flow analysis phase of the Roslyn compiler is responsible for implementing the C# language specification’s rules about *reachable statements* (section 8) and *definite assignment* (section 5.3.3). Rather than being a strict implementation of the specification (which itself has some nontrivial bugs), the implementation is intended to accept all programs accepted by Dev10 and all programs accepted by the intent of the language specification. This phase also computes and reports the warnings regarding unused local variables.

The code for the flow analysis phase is also shared (by inheritance) with the implementation of a number of Roslyn’s *region analysis* APIs, which are used to support such IDE tools as *extract method*. To make that work, there are a number of “hooks” within the code.

The flow analysis phase processes a method’s body by maintaining a *state* variable that represents the definite assignment and reachability state of the program at the current point being analyzed. The state is updated according to the rules of the specification as the analysis scans through the body of the method.

The top level of the flow analysis phase is FlowAnalysisPass.Rewrite, which performs flow analysis and inserts a return statement at the end of the body if necessary. It uses the class FlowAnalysisWalker to scan the body of the method. The current “state” of the analysis is stored in the field FlowAnalysisWalker.state of type FlowAnalysisLocalState.

# Reachable Statements

Reachable statement analysis is the simplest analysis performed in this phase. According to the C# specification:

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable. (8.1)

Like most things in the real world, it isn’t quite that simple. The Dev10 compiler will not produce a warning if an empty statement or a block statement is unreachable (though it will complain if some other kind of statement within the block is unreachable). The implementation of this phase aims to reproduce the Dev10 behavior.

# Definite Assignment State

(TODO) After a statement that doesn’t complete normally (return, goto, throw, etc), we set the current state to an *unreachable* state…

# Definitely Assigned Variables

The more complex work of this phase is computing which variables are *definitely assigned* at each point in the program, in order to implement the following sentences of the specification:

A local variable must be definitely assigned (§5.3) at each location where its value is obtained. (8.5.1)

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used. (10.6.1.3)

Every output parameter of a method must be definitely assigned before the method returns. (10.6.1.3)

If the struct instance constructor doesn’t specify a constructor initializer, the this variable corresponds to an out parameter of the struct type, and similar to an out parameter, this must be definitely assigned (§5.3) at every location where the constructor returns. (11.3.8)

The “precise” rules for computing definite assignment states appears in the C# specification in section 5.3.3. However the specification is missing some significant foundational rules (discussed later).

# The State Representation (DA, or DAT/DAF)

According to the C# specification (5.3.3), a variable’s definite assignment state falls into one of four categories:

* Definitely assigned
* Not definitely assigned, with the following possible (exclusive) sub-states
  + Definitely assigned after true expression
  + Definitely assigned after false expression

The latter sub-states are only meaningful *after* the analysis of a Boolean expression. In effect, there are four possible states.

Within the compiler, we have two distinct mutually exclusive representations for definite assignment states: one for after a Boolean condition, and one for everywhere else.

After a Boolean condition, we represent the state of a variable using two bits (which is enough to represent four states). One represents whether the variable is definitely assigned when the expression is true (DAT). The other represents whether the variable if definitely assigned when the expression is false (DAF). Together these two bits can represent all four states of the specification in a way that (as we will see) makes the specification easier to implement:

|  |  |  |
| --- | --- | --- |
| DAT=false | DAF=false | Not definitely assigned |
| DAT=true | DAF=false | Definitely assigned when true |
| DAT=false | DAF=true | Definitely assigned when false |
| DAT=true | DAF=true | Definitely assigned |

We process a boolean condition using the method FlowAnalysisWalker.VisitCondition(BoundExpression node). It always produces a result (in the state variable) in this two-bit (per variable) representation.

In all other places, we use a single “definitely assigned” (DA) bit to represent the definite assignment state of a variable: either it is definitely assigned (DA=true) or not (DA=false). We process an expression using the method FlowAnalysisWalker.VisitExpression(BoundExpression node). It always leaves the result (in the state variable) in the one-bit (per variable) representation.

# Abstract Flow Analysis

(TODO) Flow analysis has been abstracted to separate control-flow analysis (which is needed early in processing lambda bodies) from data-flow analysis (which is done later).

# Specification versus Implementation

This decomposition of the definite assignment state space, while different from what is written in the C# specification, allows us to write a much simpler implementation of the specification than what would be required if the representation corresponded more directly to the terms of the specification. Let’s look at an example

# Example: && and &

The binary & operator is one of the simplest cases. It follows the “general rules for expressions with embedded expressions” (5.3.3.21), which reads

#### 5.3.3.21 General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions (§7.6.3), element access expressions (§7.6.6), base access expressions with indexing (§7.6.8), increment and decrement expressions (§7.6.9, §7.7.5), cast expressions (§7.7.6), unary +, -, ~, \* expressions, binary +, -, \*, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ expressions (§7.8, §7.9, §7.10, §7.11), compound assignment expressions (§7.17.2), checked and unchecked expressions (§7.6.12), plus array and delegate creation expressions (§7.6.10).

Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed order. For example, the binary % operator evaluates the left hand side of the operator, then the right hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. For an expression expr, which has sub-expressions expr1, expr2, ..., exprn, evaluated in that order:

* The definite assignment state of v at the beginning of expr1 is the same as the definite assignment state at the beginning of expr.
* The definite assignment state of v at the beginning of expri (i greater than one) is the same as the definite assignment state at the end of expri-1.
* The definite assignment state of v at the end of expr is the same as the definite assignment state at the end of exprn.

The implementation (in FlowAnalysisWalker.VisitBinaryOperator) is much simpler:

VisitExpression(node.Left);  
VisitExpression(node.Right);

For the Boolean && operator, the C# specification says

#### 5.3.3.24 && expressions

For an expression expr of the form expr-first && expr-second:

* The definite assignment state of v before expr-first is the same as the definite assignment state of v before expr.
* The definite assignment state of v before expr-second is definitely assigned if the state of v after expr-first is either definitely assigned or “definitely assigned after true expression”. Otherwise, it is not definitely assigned.
* The definite assignment state of v after expr is determined by:
* If the state of v after expr-first is definitely assigned, then the state of v after expr is definitely assigned.
* Otherwise, if the state of v after expr-second is definitely assigned, and the state of v after expr-first is “definitely assigned after false expression”, then the state of v after expr is definitely assigned.
* Otherwise, if the state of v after expr-second is definitely assigned or “definitely assigned after true expression”, then the state of v after expr is “definitely assigned after true expression”.
* Otherwise, if the state of v after expr-first is “definitely assigned after false expression”, and the state of v after expr-second is “definitely assigned after false expression”, then the state of v after expr is “definitely assigned after false expression”.
* Otherwise, the state of v after expr is not definitely assigned.

That’s quite a lot to understand. The implementation does the same thing, but is much easier to follow:

VisitCondition(node.Left);  
var leftState = this.state;  
this.state = new FlowAnalysisLocalState(  
 this.state.Reachable, leftState.AssignedWhenTrue);  
VisitCondition(node.Right);  
this.state.AssignedWhenFalse.IntersectWith(leftState.AssignedWhenFalse);

Where “IntersectsWith” is a Boolean & operation.

This is typical code in the implementation, so it would be valuable to make sure you understand how this code implements the meaning of the specification in a way that takes advantage of the compact representation (two bits for conditions, and one bit for general expressions) of the definite assignment state.

# The treatment of “true” and “false”

There is a foundational rule missing from the C# language specification, without which a large number of distinct use cases will not work “properly”. For example, Dev10 accepts the following, in spite of the fact that x is read but never written:

    static void Main(string[] args)  
    {  
        int x;  
        if (true || **x** == 3) { }  
    }

There is a lot of of code in Dev10’s flow analysis to handle this and other “missing” cases from the specification. The comments in Dev10 explain this in terms of supporting “unreachable expressions”, which it suggests are a necessary but missing element in the specification. However, the set of special cases needed is large, and Dev10 gets many of them, but not all of them. For example, the following code is (incorrectly) rejected by Dev10:

    static void Main(string[] args)  
    {  
        int x;  
        if (false && x == 3)  
        {  
            int z = x; // Dev10 error: x unassigned  
        }  
    }

It turns out that there is a much simpler way of handling all of these special cases. They are all due to two foundations rules being missing from the specification. Those are:

5.3.3.N Constant Expressions

For a constant expression with value true:

·         If *v* is definitely assigned before the expression, then *v* is definitely assigned after the expression.

·         Otherwise v is “definitely assigned after false expression” after the expression.

For a constant expression with value false:

·         If *v* is definitely assigned before the expression, then *v* is definitely assigned after the expression.

·         Otherwise *v* is “definitely assigned after true expression” after the expression.

In the algebra of definite assignment states, these rules are the “zero”. Roslyn’s flow analysis phase implements these foundational rules directly (in FlowAnalysisWalker.VisitExpression). None of the code from Dev10 for handling “unreachable expressions” are necessary with these in place.

# Suppressing cascaded diagnostics

When an error has been produced that an unassigned variable is being used, we set a flag (associated with the variable). Once the flag has been set, we do not produce an error on subsequent uses of the variable.

Similarly, when a warning has been produced that a statement is unreachable, there is no point in reporting subsequent statements unreachable. To suppress these cascaded diagnostics, we change the state from “unreachable” to “reachable” after producing the warning. Only the first in a sequence of unreachable statements will trigger a warning. This reproduces Dev10’s behavior.

However, since we also use reachability to decide when we need to insert a final return statement into a method (or when a final return statement is missing), we also maintain data that gives a more precise way of determining whether code is reachable. We do that by simulating the existence of a user-declared variable that is never assigned. Since the flow analysis makes *every* variable definitely assigned in unreachable code, if we find that this variable is definitely assigned at some point in the program, then it must be the case that the point in the program is unreachable. We use this technique to determine when to report a missing return statement for non-void methods, or to insert a return statement at the end of a void method.

# Bit vector implementation of the state

The C# language specification talks about the definite assignment state of variable *v*, but we need to compute the definite assignment state for *every* variable. We do that by using a bit vector to represent the state, with one bit allocated to each variable that is tracked.

See FlowAnalysisLocalState for the state representation, which has either one or two bit vectors, and BitArray for the underlying implementation of the bit vector.

# Control-flow and pending jumps

(TODO) Explain try-finally, ResolveBreaks, RestorePending, VisitBlock.

# Loops

(TODO) Note that in the normal analysis, only one pass is sufficient to handle loops. The backward branch doesn’t change the state.

# GOTO

(TODO) The treatment of GOTO may require multiple passes over the whole tree. That’s because when we encounter a label statement, we may not have seen all branches to the label, and we therefore do not know the state at the label…

# Region Analysis

(TODO) Explain the implementation of each analysis, and show why the hooks are where they are.

# trackUnassignmentsInLoops

(TODO)