Steering Clear of Triples: Deriving the Control Flow Graph Directly from the Abstract Syntax Tree in C Programs

Naftali Schwartz

May 24, 1998

Abstract

This article explores the extension of Morgenthaler's Virtual Control Flow technique[Mor97], which derives control flow[ASU86] semantics directly from the Abstract Syntax Tree, from the relatively coarse granularity of syntactic C expressions to the finer granularity of basic block expressions, that is, expressions without embedded control flow. We explain why this is a better level of abstraction for program analysis, and discuss the elements of an efficient and elegant solution, motivating the presentation by appealing to a more explicit intermediate form. We present our algorithm, and conclude with remarks about the suitability of Morgenthaler's version of Virtual Control Flow for customary exhaustive data-flow analysis.

1 Introduction

It is now well established that the abstract interpretation of programs pursued in static analysis (used, e.g., in program verification and optimization[AH87]) is best served by a high-level program representation. There are a variety of reasons for this:

- the analysis can more easily locate common programmatic idioms for which helpful transformations are already known
- the association between the input program and intermediate representation can be more closely maintained; this is helpful for diagnostics and debugging
- code transformations, such as those performed within an interactive program development or optimization environment, would not engender expensive reconstruction of elaborate intermediate forms
- avoiding complexity in program representation allows us to concentrate more resources on program analysis

As part of ongoing research within the MILAN project[Ked], we are developing a set of prototypes to enable reliable execution of parallel computations on distributed platforms. One research direction calls for the construction of a source-code parallelization tool which could benefit substantially from a high-level representation. Not only would initial construction of this tool become easier, but more elaborate analyses might be possible by keeping the system simple. However, we discovered that taking full advantage of that approach would not be possible unless we could extend the current state-of-the-art as represented by John Morgenthaler's recent work[Mor97].

1.1 Virtual Control Flow

Morgenthaler has developed a program analysis tool, Cstructure[Mor97], which uses the Abstract Syntax Tree (AST) as the program's sole intermediate representation. He details how the control flow[ASU86] predecessors and successors of any statement may be computed on demand, thus providing precisely enough information to drive the demand driven data-flow analysis he adopts. He terms this technique *Virtual Control Flow*, since no Control Flow Graph (CFG) links are ever constructed.

As defined by Morgenthaler, Virtual Control Flow describes only the execution ordering between syntactic expressions. Since embedded control flow gives rise to C expressions which are only partially executable, the semantics of an arbitrary C expression may depend on the values of its component expressions. This generates ambiguities akin to those associated with indirect function calls, where the actual code executed depends on the current value of a function pointer[CG94].

To cope with this uncertainty, Morgenthaler categorizes subcomponents of expressions containing embedded control flow as either *may-execute* or *must-execute*. A killing definition, for example, in a *may-execute* context, becomes a preserving definition. However, it is easy to see that this information provides little detail about the internal orderings within expression subcomponents, and Morgenthaler points out that standard data-flow analysis techniques, such as SSA construction[CFR+91] cannot be applied directly in this context.

1.2 Building a CFG with Virtual Control Flow

We explore the extension of Morgenthaler's technique to the domain of customary exhaustive data-flow analysis, where we would actually like to construct a CFG, while avoiding the syntax-directed translation into three-address code of [ASU86], which adds an additional layer of obfuscation to control flow which is, after all, explicit in the source code.

Thus, we seek to calculate (and store) the predecessors and successors of every statement. Since we are calculating the complete solution and these are two halves of a dual problem, we need only compute either predecessor or successor information. We choose the latter for simplicity and efficiency. (We shall have more to say about direct computation of predecessors in Section 6.)

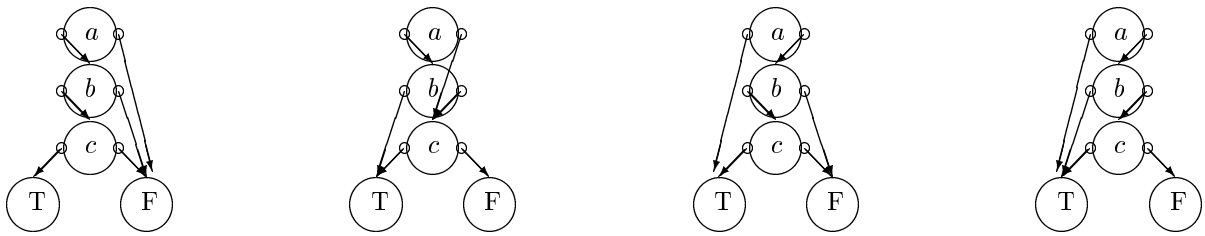
The direct computation of successor links among statements is not very difficult. As Morgenthaler points out, the only complications are locating the targets for unstructured and semi-structured jumps (the **goto**, **break** and **continue** statements), and an efficient implementation can easily compute successors of these and all other statements in amortized constant time from the AST.

1.3 Importance of Finer Expression Grain

This relative simplicity of control flow analysis at the syntactic expression level does not carry over to the basic block expression level, however. Since the C language provides a very rich way of connecting expression components into flow graphs, the proper links cannot be generated from knowledge of syntactic juxtaposition of expressions, but depends on the operators which join them. Therefore, an effective implementation of Virtual Control Flow for expressions requires substantially more sophistication than is necessary for statement control flow, for which the only connectors are containment and sequencing.

C provides three basic interconnective operators for embedded control flow, **&&**, || and **,**. (The choice operator, **?:**, will be treated in Section 2.1.) These are further complicated by operator precedence and explicit parenthesezation.

Figure 1, for example, details the graphs corresponding to some simple expressions containing control flow,



(a) $(a \;\&\&\; b) \;\&\&\; c$ (b) $(a \;\&\&\; b) \;||\; c$ (c) $a \;||\; (b \;\&\&\; c)$ (d) $(a \;||\; b) \;||\; c$

(e) $a \;\&\&\; (b \;\&\&\; c)$ (f) $a \;\&\&\; (b \;||\; c)$ (g) $(a \;||\; b) \;\&\&\; c$ (h) $a \;||\; (b \;||\; c)$

Figure 1: Correspondence between C expressions and flow graphs.

using only the operators **&&** and ||. Each node in this figure is drawn with two exits. The right exit represents the direction taken if the expression evaluates to true, and the left to the one taken if false. Note that these two exits coalesce into one if the expression is used anywhere other than within a logical context, i.e., an **if** or other loop conditional, or as an argument to one of the logical operators **&&** or ||, or at the beginning of a **?:** construct. Figures 1(a-d) represent the default parenthesezation, while 1(e-h) represent the alternative parenthesezation.

Notice that in two of the graphs, (b) and (g), the flow of control is richer than can be represented by Morgenthaler's *may-execute* and *must-execute* predicates. Each of the graphs shows that while both of nodes a and b may be executed, exactly one of them *must* be executed. Therefore, if they both contain killing definitions, for example, the expression as a whole must generate a killing definition, something that Morgenthaler's formalism will miss. On the other hand, the SSA-based scalar dependence analysis presented in [SW95] easily handles this alternative control flow, when properly expressed in the CFG.

The above considerations, then, motivate our search as to how we may integrate the embedded control flow of expressions into the final CFG to avoid the extra complications and imprecision Morgenthaler finds are the consequences of the use of coarser granularity.

2 Reading Successor Links From the AST

The structure of the AST does not necessarily correspond in any obvious way to the shape of the successor graph that must be constructed.

Figures 2 and 3, for example, demonstrates this basic disparity. Figure 2 shows the AST that represents the expression $a \;||\; b \;||\; c \;||\; (d \;||\; e) \;\&\&\; f \;\&\&\; g,\; h \;\&\&\; i \;\&\&\; j \;\&\&\; k$, while Figure 3 details the successor links that need to be constructed. Note, for example, that the fact that f and h are successors of d is not even

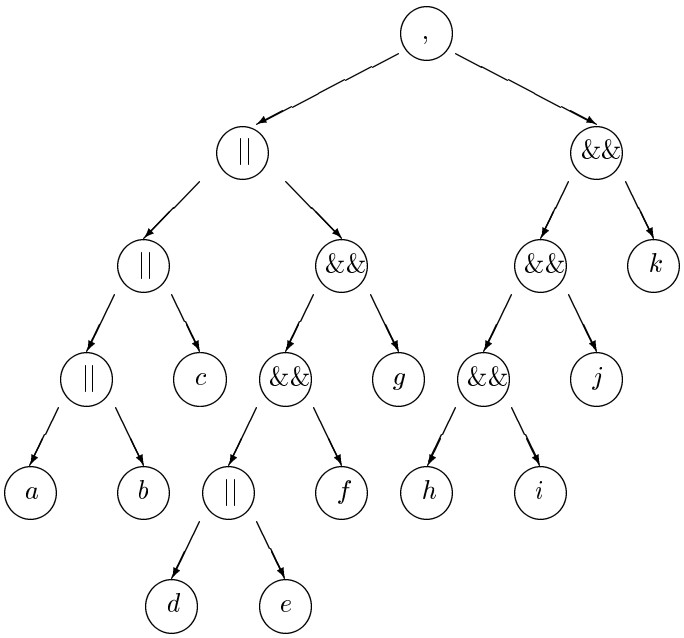


Figure 2: Abstract Syntax Tree.

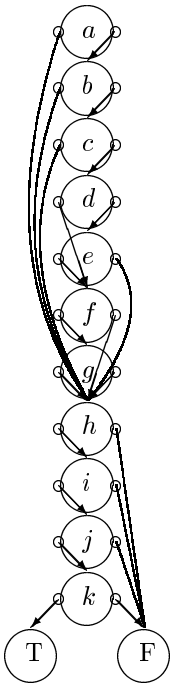


Figure 3: Successor Graph.

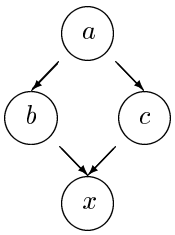


Figure 4: Flow graph for $a?b : c$.

remotely reflected in the AST topology.

Furthermore, linking to other syntactic expressions in other statements involves linking to the specific sub-expression that is to be executed first. For an expression containing an arbitrary mix of control-flow and non-control-flow operators, it's not at all obvious which sub-expression is the actual entry point of the entire expression.

2.1 The Choice Operator

The choice operator, although introducing embedded control flow, does not introduce complexity in determining successor links. This is because the operator's return value does not define a boolean value directly, but rather an arbitrary value[KR88]. Although this value may then be used to direct control flow, this is only an indirect effect. Consider, for example, Figure 4. This graph shows that while control does indeed flow through either b or c, control must ultimately come to internal node x, which is where the actual value of the entire choice expression is recorded. Therefore, the choice operator does not contribute to the overall complexity of Virtual Control Flow for expressions.

2.2 Mixing Operators

Simple expressions which contain control-flow expressions as subexpressions require special handling. In this case, as with the choice operator, it is a value which is the result of the control flow rather than a transfer of control. Therefore, we create dummy true and false targets for each of these control-flow expressions which simply record the result into a temporary. Figure 5 details this construction for the expression $(a \mid\mid b)$ && c + d&& e.

In this figure, there is a separate subgraph for the control-flow subexpression $(a \mid\mid b)$ && c, a separate subgraph for the control-flow subexpression d&& e, and extra support nodes and edges to propagate the result value of each of these subgraphs back to the original expression through compiler temporaries. Notice that all exits of the first subgraph lead to the start of second subgraph, and all exits of second subgraph lead to the single expression node. Furthermore, the arbitrary order of evaluation of C expressions[KR88] tells us that the general order of the first two subgraphs could also have been reversed.

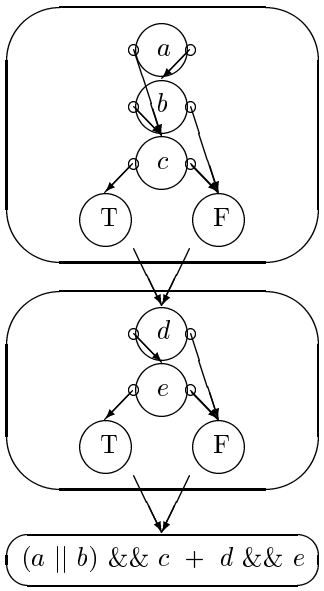


Figure 5: Flow graph for $(a \mid\mid b)$ && $c + d$ && e.

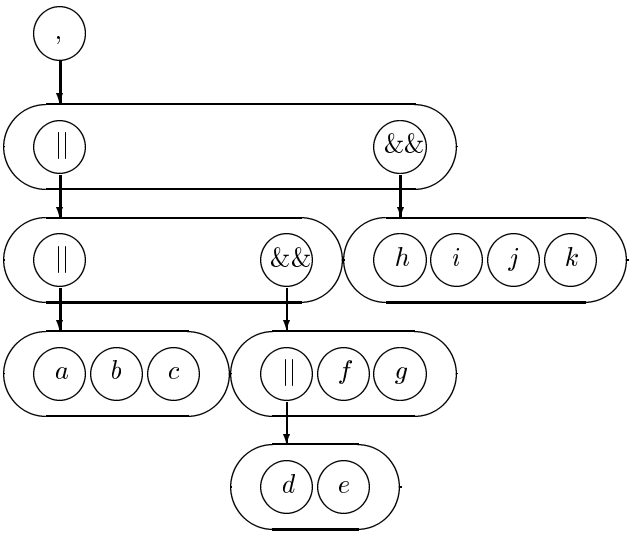


Figure 6: Equivalent parse tree, with k-ary operators.

3 A More Explicit Intermediate Form

In fact, the successor links are hard to see because they are obfuscated by the construction of customary parse tree. There are other ways to construct the parse tree of Figure 2 that make the relationships we seek more explicit.

```
1: Eflow( e, t, f )
2:         if f is a flow operator expression then
3:             Eflow( e, f == t ? leftoperand-of( f ) : t, leftoperand-of( f ) )
4:         else if t is a flow operator expression then
5:             Eflow( e, leftoperand-of( t ), f )
6:         else if e is a flow operator expression then
7:             if operator-of( e ) == , then
8:                 Eflow( leftoperand-of( e ), rightoperand-of( e ), rightoperand-of( e ) );
9:             else if operator-of( e ) == && then
10:                Eflow( leftoperand-of( e ), rightoperand-of( e ), f );
11:            else if operator-of( e ) == || then
12:                Eflow( leftoperand-of( e ), t, rightoperand-of( e ) );
13:            Eflow( rightoperand-of( e ), t, f )
14:        else if t == f then
15:            set sole successor of e to t
16:        else
17:            set true/false successors of e to t and f
18: End Eflow
```

Figure 7: The expression flow algorithm.

Figure 6 shows the construction of a semantically equivalent parse tree. In this tree, operators are not binary, but k-ary, reducing ordered lists of operands. (The semantics of this reduction is obvious in the case of the three operators with which we are concerned: **&&**, || and **,**.)

We now examine how a single pass over this tree may construct the requisite links. The first observation is that all of the nodes a - g must ultimately lead to node h. Within this group, nodes a - c must all have true exits pointing to d. Of these, each false exit must point to the next right sibling, except for c, whose false exit must point to d due to the absence of a right sibling. In general, for all children of an || node, true exits point to the parent's true exit, and false exits point to either a right sibling, or the leftmost leaf of the next tree branch.

In the case of the following **&&** node, the true exit of all it's children point to their next sibling on the right. Applying the logic in the preceding paragraph, then, we have *exits*(d) = (f, e), *exits*(e) = (f, h), *exits*(f) = (g, h), and *exits*(g) = h.

Now that we understand how to construct the requisite links at each node in the AST knowing only the links pending at the parent node and the operation of the current node, we show how this may be done without explicit construction of this intermediate form.

4 The Algorithm

Figure 7 shows the pseudocode for our algorithm, **Eflow**. **Eflow** takes three arguments:

- the expression we are finding successors for
- the true target expression
- the false target expression

For expressions that appear in a non-logical context, the true and false targets identical.

The initial calls of the algorithm serve to find the leftmost leaves of c and b, because it is to these leaves that we need to link a.

5 An Example

We list in Figure 8 all the steps the algorithm goes through to generate all the links shown in Figure 3.

6 Complexity of Virtual Control Flow

6.1 Complexity of Eflow

It is easy to see that **Eflow** visits each node of the subject expression AST exactly once, plus, in the worst case, most of the AST nodes of each of target1 and target2. This gives an overall complexity of

Θ(average number of nodes in an expression AST)

Thus the cost is linear in the size of the program.

We will now contrast this with a method which would be more in keeping with that originally presented by Morgenthaler.

6.2 Directly Extending Morgenthaler's Method

With the goal of keeping as little state as possible, Morgenthaler proposed that even control flow information be computed on a demand driven basis. Thus, to compute reaching definitions[ASU86] on demand, for example, Morgenthaler starts with a particular program point, and recursively computes predecessors at each node until a killing definition is reached along each execution path leading to the program point.

It is interesting to note that if implementation suggested in [Mor97] were directly extended to incorporate the finer granularity promoted in this article, efficiency would suffer greatly. Morgenthaler insists that storage of predecessor and successor links would slow down the system and not make it responsive in interactive editing sessions. In fact, with this extension, the truth may be the reverse.

To see this, consider once again Figure 2. It is easy to see that in the worst case, a single predecessor computation within a complex expression with embedded control flow may have to consider *every component of the expression*. Since this is a recursive computation that continues until stopped by killing definitions, it may have to examine most components of complex expressions *several times*. This can easily produce quadratic running time in a single data-flow query.

Contrast this with a CFG construction. We've seen that fine-grain expression flow successors (and therefore predecessors) can be computed in linear time. This computation, then, will save a great deal of time over the simplistic "blind search" for predecessors.

This seems to be a good example, then, of how solving a more difficult problem, finding successors and predecessors of all nodes, can sometimes be easier than solving an apparently easier problem— finding the predecessors of a single node.

```
Eflow( ‘a || b || c || (d || e) && f && g , h && i && j && k’, ‘T’, ‘F’, ‘||’ )
Eflow( ‘a || b || c || (d || e) && f && g’, ‘h && i && j && k’, ‘h && i && j && k’, ‘,’ )
Eflow( ‘a || b || c || (d || e) && f && g’, ‘h && i && j’, ‘h && i && j’, ‘,’ )
Eflow( ‘a || b || c || (d || e) && f && g’, ‘h && i’, ‘h && i’, ‘,’ )
Eflow( ‘a || b || c || (d || e) && f && g’, ‘h’, ‘h’, ‘,’ )
Eflow( ‘a || b || c’, ‘h’, ‘(d || e) && f && g’, ‘||’ )
Eflow( ‘a || b || c’, ‘h’, ‘(d || e) && f’, ‘||’ )
Eflow( ‘a || b || c’, ‘h’, ‘(d || e)’, ‘||’ )
Eflow( ‘a || b || c’, ‘h’, ‘d’, ‘||’ )
Eflow( ‘a || b’, ‘h’, ‘c’, ‘||’ )
Eflow( ‘a’, ‘h’, ‘b’, ‘||’ )
        a --> ( h, b )
Eflow( ‘b’, ‘h’, ‘c’, ‘||’ )
        b --> ( h, c )
Eflow( ‘c’, ‘h’, ‘d’, ‘||’ )
        c --> ( h, d )
Eflow( ‘(d || e) && f && g’, ‘h’, ‘h’, ‘,’ )
Eflow( ‘(d || e) && f’, ‘g’, ‘h’, ‘&&’ )
Eflow( ‘(d || e)’, ‘f’, ‘h’, ‘&&’ )
Eflow( ‘d’, ‘f’, ‘e’, ‘||’ )
        d --> ( f, e )
Eflow( ‘e’, ‘f’, ‘h’, ‘&&’ )
        e --> ( f, h )
Eflow( ‘f’, ‘g’, ‘h’, ‘&&’ )
        f --> ( g, h )
Eflow( ‘g’, ‘h’, ‘h’, ‘,’ )
        g --> h
Eflow( ‘h && i && j && k’, ‘T’, ‘F’, ‘||’ )
Eflow( ‘h && i && j’, ‘k’, ‘F’, ‘&&’ )
Eflow( ‘h && i’, ‘j’, ‘F’, ‘&&’ )
Eflow( ‘h’, ‘i’, ‘F’, ‘&&’ )
        h --> ( i, F )
Eflow( ‘i’, ‘j’, ‘F’, ‘&&’ )
        i --> ( j, F )
Eflow( ‘j’, ‘k’, ‘F’, ‘&&’ )
        j --> ( k, F )
Eflow( ‘k’, ‘T’, ‘F’, ‘||’ )
        k --> ( T, F )
```

Figure 8: Execution trace.

7 Related Work

As a prototypical example of other efforts at maintaining a high-level intermediate code representation, we mention only the SIMPLE language used in the McCat compiler[HDE+93]. In this work, all complexities of the C language, including embedded expression control flow and unstructured control flow at the statement level, are rewritten using a basic subset of the C language.

While it's certainly true that such a representation can be analyzed more easily than the original program, there is also a cost in system complexity associated with the introduction of this intermediate format, and

perhaps an obfuscation of the proper correspondence of the original input program to the (equivalent) program which is being analyzed. It is this complexity that the present author has labored to avoid.

8 Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

The author gratefully acknowledges perceptive insights into the nature of the problem and possible solutions by Miriam Schwartz and Daniel Schwartz. Thanks also to Zvi Kedem, Davi Geiger and Eric Freudenthal for directing the application of MILAN techniques to MSTAR. It was this work that ultimately uncovered the problem and inspired the search for a general solution.

References

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* John Wiley and Sons, 1987.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994.

[HDE+93] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. *Lecture Notes in Computer Science*, 757:406–??, 1993.

[Ked] Zvi M. Kedem. Metacomputing in Large Asynchromous Networks. See http://www.cs.nyu.edu/milan/milan/index.html.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Eaglewood Cliffs, NJ, second edition, 1988.

[Mor97] J. Morgenthaler. *Static Analysis for a Software Transformation Tool.* PhD thesis, University of California, San Diego, 1997.

[SW95] E. Stoltz and M. Wolfe. Detecting value-based scalar dependence. *Lecture Notes in Computer Science*, 892:186–??, 1995.