AUTOMATIC FUNCTION POINT COUNTING USING STATIC AND DYNAMIC CODE ANALYSIS

Keith Paton
350 Pine Avenue
Saint Lambert (Quebec)
Canada
J4P 2N8
Tel (450) 671-1969
fax (450) 465-9386
internet paton@total.net

SUMMARY

We define an intermediate representation of a program P as a data flow graph DF(P) and shown that this representation allows us to express the program as a quadruple Q={F,T,r,w} useful in function point analysis.

We show that we can derive DF(P) by program slicing, a form of static code analysis. Starting with one given output file, A say, we derive the smallest program A(P) that mimics P in its writing to A. When we repeat this process for all output files,(A,B,C,D say) we obtain a set of programs A(P), B(P), C(P) and D(P) in which any two are disjoint or identical. The number of unique such programs is the number of transactions and straightforward analysis yields DF(P).

We shown that we can also derive DF(P) by program tracing, a form of dynamic code analysis. In this case, we can modify the program P being studied into a second program P' such that P' has the same behavior as P and P' generates a trace showing what it is doing. From the trace, we can automatically derive the intermediate representation DF(P). The modification of P to P' can be carried out automatically by methods of static code analysis now under development.

1. INTRODUCTION

This paper discusses the question

Given a working program, how can we apply some simple methods of static and dynamic code analysis to generate files that will be valuable in automatic function point analysis?

I shall interpret the phrase *a working program* to mean an exe file derived from a set of C source files and headers. This means that we can investigate in three ways:

1	We can run the exe file as supplied and observe its behaviour.	This is the fly-on-the-wall technique, in which we use whatever tools we have available to examine the external behaviour of the program.
2	We can analyze the source files and header files and predict the behaviour of the exe file from that	This refers to static code analysis, in which we use tools to examine the code of the program for evidence that certain files exist, certain transactions exist, certain transactions read certain files, and so on. We discuss this in section 3.
3	We can modify the source files in such a way that when we regenerate the program the program writes a trace file describing its behaviour.	This refers to dynamic code analysis, in which we use tools to modify the program itself in such a way that the modified program not only does its original work but also tells us about the files, the transactions, the reading and so on. We discuss this in section 4.

What do we mean by the phrase

files that will be valuable in automatic function point analysis?

If we can extract from the working program a list of *files* and a list of *transactions* together with a statement of which transactions *read* and *write* which files, we shall have almost enough to count function points. We shall also need a list of *applications* and a statement of which applications *own* which files and which transactions. In short, we need a sextuple S={A,F,T,r,w,o} where

- F is a set of files
- T is a set of transactions
- r is a reads relation on (T,F)
- w is a writes relation on (T,F)
- A is a set of applications
- is an owns relation on (A,T∪F)

In this paper we show how to derive the quadruple Q={F,T,r,w}.

I should add that the files we talk of here are physical files. I think there is no way of seeing into the mind of the user to detect the logical files , defined as grouping of data as perceived by the user. Since our files are physical, our transactions (constructed programs that manipulate files) are also physical.

Section 1 introduces a family of test cases and explains why they it is relevant to function point analysis. Section 2 defines a representation of the program as a data flow graph from which we can derive the quadruple Q. Section 3 shows how we can use static code analysis to obtain the data flow graph. Section 4 shows how we can use dynamic code analysis to derive a program trace from which we can to obtain the data flow graph.

1. Test Cases

The programs P1 and P2 to be analyzed will be handed out at the talk. Each program reads one integer per record from files W and X and each program writes one integer per record to files A and B. The relation between inputs and outputs is as follows

P1	P2
a=w+x b=w-x	a=w b=-x

Att first sight both programs are two-input, two-output transactions, but this is to ignore their internal structure. We can break down P1 to two transactions, each involving one input and one output, whereas we can break down P2 in these two ways:

- one transactions each involving two inputs and two outputs
- two transactions each involving two inputs and one output

Since the inputs variables (w,x) come from the files (W,X) respectively and the output variables (a,b) go to the files (A,B) respectively, it follows that the output files depend on the input files like this:

	output	
input	A	B
W	T1	
X		T2

P1

	output	
input	A	B
W	T3	T3
X	T3	T3

P2 (first view)

	output	
input	A	B
W	T4	T5
X	T4	T5

P2 (second view)

Figure 1: Dependence of output files on input files

We have identified transactions in a way that will become clear in section 3. We shall show at the talk that we can derive these three views by static code analysis (section 3) or dynamic code analysis (section 4). We can now identify each unique index as a transaction and obtain these descriptions of the subprograms:

Subprogram	Description
P1	There are four files {A, B, W, X,} and two transactions {T1,T2}. The reads relation is {(T1, W), (T2, X)}. The writes relation is {(T1, A),(T2, B) }. Each of the two transactions has one input and one output.
P2 (first view)	There are four files {A, B, W, X,} and one transaction {T3}. The reads relation is {(T3, W), (T3, X)}. The writes relation is {(T3, A),(T3, B) }. The transaction has two inputs and two outputs.
P2 (second view)	There are four files {A, B, W, X,} and two transactions {T4,T5}. The reads relation is {(T4, W), (T4, X),(T5,W),(T5,X)}. The writes relation is {(T4, A),(T5, B) }. Each of the two transactions has two inpus and one output.

We now have to show how we can get from the programs to these descriptions automatically. In section 2 we define an intermediate representation of the program as a data flow graph.

2. DATA FLOW GRAPH

We now define an intermediate representation of the program as a form of data flow graph. We recall that a graph consists of nodes and edges joining pairs of nodes. In our case a node can represent a datum or a statement. A datum is one of

- A file name
- A file pointer
- A variable
- A statement is one of
- An fopen statement
- A write statement
- A read statement
- An assign statement

We first show the graph for P2 and then explain how it is built up from an analysis of the code.

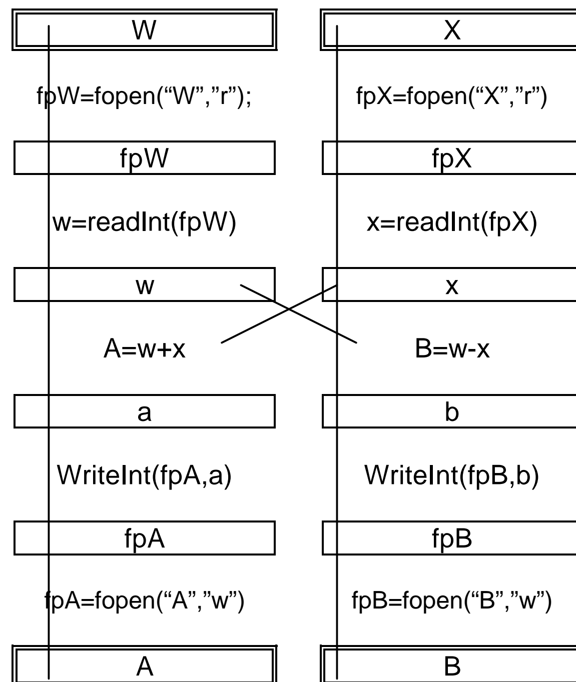

Figure 2: Program Graph for P2

Nodes in solid boxes denote data; nodes in dotted boxes denote statements. Nodes in double solid boxes denote files; nodes in single solid boxes denote variables. I shall explain the edges of this graph at the talk.

We show the first few steps of the analysis that generates this graph.

We start by choosing a statement such as fpA=fopen("A","w") that opens an output file.	We generate a node for this statement.
We examine the statement and find in it two data, the file name "A" and the file pointer fpA.	We generate nodes for these two and draw edges (fpA, fpA=fopen("A","w")) and (fpA=fopen("A","w"), A).
We now examine the datum fpA and find that it is involved in statement writeInt(fpA,a).	We generate the node for the statement and draw the edges (writeInt(fpA,a), fpA).
We now examine the statement writeInt(fpA,a) and find that it involves the variable a.	We generate a node for a and draw the edge (a, writeInt(fpA,a)).
We now examine the datum a and find that it is involved in statement a=w+x	We generate the node for the statement and draw the edges (a=w+x, a).
We now examine the statement a=w+x and find that a depends on w and x.	We generate nodes for w and x and draw the edges (w,a=w+x) and (x,a=w+x).
We now examine the datum w and find that it is involved in statements b=w-x and readInt(fpW).	We generate nodes for w and x and draw the edges (w,a=w+x) and (x,a=w+x).
… and so on …	

Thus the analysis proceeds by alternate examination of statements and data. When no more analysis can be done starting from the statement fpA=fopen("A","w") we choose another open statement such as fpB=fopen("B","w") and continue. When we have analyzed the two fopen statements that open files for writing, we are done.

The next step is to do a connectivity analysis on the graph G. We can do this in two ways.

First, we can start at A and trace backwards along all the arrows, generating g(A), then start again at B and trace backwards along all the arrows, generating g(B). Note that g(A) and g(B) share the subgraphs rooted at w and at x. In fact g(A) = G1 + G3 and g(B) = G2 + G3 where G1, G2 and G3 are pair-wise disjoint and G1+G2+G3 gives us back the whole graph. This leads to the first view (two transactions each with one output)

Alternatively, we can start at A and trace backward and forward along all the arrows. This generates the whole graph G. This leads to the second view (one transaction with two outputs)

Section 3 shows how to derive this data flow graph using program slicing; section 4 shows how to derive it using program tracing.

3. PROGRAM SLICING

We use a minor variant of the program slicing technique explained to me by Ettore Merlo in his lectures at Ecole Polytechnique. We shall illustrate it at the talk on program P2. We select the single fopen statement involving A and construct the smallest subprogram of P containing this fopen statement. We identify this program slice as one transaction with two inputs (W and X) and one output (A). We then select the fopen statement involving B and repeat. This yields a program slice which we identify as one transaction with two inputs (W and X) and one output (B). This approach leads to the first view of P2.

In this slicing technique we have used this rule.

- If statement S is live and statement S requires statement T then statement T becomes live.

This approach generates the smallest program containing the initial statement.

Suppose now that we vary the rule to this pair of rules.

- If statement S is live and statement S requires statement T then statement T becomes live.

- If statement S is live and statement T requires statement S then statement T becomes live.

This corresponds to the forward tracing we saw in section 2. This approach leads to the second view of P2.

4. PROGRAM TRACING

We now turn to program tracing, a form of dynamic code analysis in which we make the program tell us what it is doing while it is doing it. To make the program speak like this we have to add statements to it. We have done this by hand for purposes of explanation but the transformation can be dome mechanically.

The transformations can be tabulated as follows:

Whenever we see …	We …
fp=fopen(…)	Replace by the statement Fp=myfopen(…)
fclose(fp)	Replace by the statement Myfclose(fp);
A=<expression involving two variables w,x>	Add the statement Depends2(&a,&w,&x)
w=readInt(fpW)	Add the statement WasRead(&w, fpW)
WriteInt(fpA,a)	Add the statement WasWritten(&a,fpA)

We create five functions myfopen, myfclose, depends2, wasRead and wasWritten. Each one writes into the trace file the appropriate information. I shall show at the talk the program trace and demonstrate that we can recover from the trace the graph shown in figure 2.

5. DISCUSSION

The techniques described here work well in the absence of aliasing and external tables.

5.1 Aliasing

Aliasing means that one quantity is known by different names at different places in the program. In a C function, like foo say, the programmer can add x to y and assign the result to z. The names x,y,z are local to the function foo.

Program slicing knows how to trace variables from one function to another; it does so by concentrating not on the name but on the address of the variable. C programmers like names but compiler writers like addresses. Aliasing is therefore not a problem for slicing.

Program tracing, as I have implemented it, finds aliasing rather tedious to deal with and I shall discuss at he talk how to overcome these difficulties.

5.2 Use of External Tables

We have assumed that the names of the files are hard-wired into the code but programmers frown on this in C. They would no doubt prefer one of two alternatives:

1. Pass the file names as arguments to the program
2. Store the files names in a table whose name is passed to the function.

I think program slicing can probably cope with 1 but not with problem 2.

Program tracing can cope with both versions above; whenever a file is opened the modified program will write a statement of the form

File N opened in mode m at pointer p

where N stands for the name of the file, m stands for the mode (read or write) and p stands for the pointer.

6. CONCLUSION

We have defined an intermediate representation of a program P as a data flow graph DF(P) and shown that this representation allows us to express the program as a quadruple Q={F,T,r,w} useful in function poiht analysis.

We have shown that we can derive DF(P) by program slicing, a form of static code analysis. Starting with one given output file, A say, we derive the smallest program A(P) that mimics P in its writing to A. When we repeat this process for all output files,(A,B,C,D say) we obtain programs A(P), B(P), C(P) and D(P) in which any two are disjoint or identical. The number of unique such programs is the number of transactions and straightforward analysis yields DF(P).

We have shown that we can also derive DF(P) by program tracing, a form of dynamic code analysis. In this case, we can modify the program P being studied into a second program P' such that P' has the same behaviour as P and P' generates a trace showing what it is doing. From the trace, we can derive the intermediate representation DF(P). The modification of P to P' can be carried out automatically by methods of static code analysis now under development.

7. REFERENCES

I will supply these at the talk.