Program Slicing

Program slicing is a technique for aiding debugging and program comprehension by reducing complexity. The essence of program slicing is to remove statements from a program that do not affect the values of variables at a point of interest. There are many ways to do this (both variants of program slicing and different algorithms for achieving the same result).

In this essay I will give an overview of slicing as well as briefly covering some of the different approaches and algorithms. I will look in detail at one particular algorithm and survey the program slicing tools available.

Motivation

Debugging programs has always been regarded as a difficult activity. Often the hardest part of debugging can be finding the bug in the first place, once the bug is found it can be easy to fix. One of the reasons that a bug can be hard to find is that there can be a lot going on in a piece of code and not all of it will have an effect on the statements of interest. Weiser noted in [5] that programmers were mentally filtering out statements that could not affect a given point of interest in order to find bugs. If this process could be automated it could be faster and more accurate (due to less scope for human error) and a useful aid for debugging.

Program slicing has many applications apart from the original motivation of debugging. It is of great use in program comprehension, for example when software has to be maintained or evolved. It can be used to generate many automated software metrics such as a measure of cohesion in a program fragment. It can also aid testing, model checking (see Bandera below) and compiler tuning amongst others [4].

Overview

The first step in slicing a program is to specify a point of interest (a statement in the program to be sliced) and a set (often a singleton set) of variables. This is called the slicing criterion and is expressed as (n, V) e.g. (11, {x}). A program slice is then computed by removing statements that can not affect (or can not be affected by, depending on the type of slicing) the values of the specified variables at the given point of interest.

The produced program slice is a reduced (although not strictly so, the whole program is a valid program slice) program. Weiser stated in [5] that this reduced program should be executable, this constraint has been relaxed in some research since then. In the case that the program slice is executable then when the original program and the slice are executed the variables specified in the criterion should have the same values at the point of interest (for all input if we are slicing statically, see below for static vs. dynamic slicing).

Some program features are more difficult to slice than others. Unstructured control flow (such as 'goto' statements) are very difficult, any indirection in a program such as pointer or array use or object orientation also makes slicing more difficult. In fact in the general case program slicing is an undecidable problem [5]. In order to carry out program slicing we have to define slicing such that a slice is only equivalent to the original program *when the original program terminates*. Furthermore, a strictly minimal slice can not be found and only an approximation can be computed, however, this is usually good enough and program slicing is still a useful technique.

Variations

There are several variations on the theme of program slicing, many of these are orthogonal variations and a wide variety of program slicing can be performed.

Backward slicing is perhaps the most intuitive form of program slicing. Here the slice is computed by working backwards from the point of interest finding all statements that can affect the specified variables at the point of interest and discarding the other statements. Forward slicing is the opposite of this (and in fact many of the algorithms for backward slicing can be simply reversed to work for forward slicing). Here we work forward from the point of interest finding those statements that can be affected by changes to the specified variables at the point of interest.

Chopping is another form of program slicing, it combines both forward and backward slicing. Two points of interest are chosen and the slice consists of those statements that can transmit a change from the source to the target. There are other ways of restricting the slice in other fashions such as dicing and barrier slicing.

There are also different ways to compute the slice. The main division is between static and dynamic slicing: static slicing works by statically analysing the code, this means examining some representation of the source code without actually executing the program in question. Dynamic slicing dynamically analyses the code by executing the program. When we slice dynamically we must do so with a given input (included in the criterion), thus a dynamic slice is only correct for a specific input. By contrast a static slice is correct for all input. Quasi-static and conditional slicing are types of program slicing that combine static and dynamic slicing.

Most forms of program slicing are syntax preserving. That is they leave the syntax of the original program largely untouched and simply remove statements to create a program slice. The exception to this rule is that if removing statements may cause a compilation error then statements may be altered (for example in Java, removing the ‘then’ part of an if statement where the if statement does not use a block statement), this is still considered syntax preserving. If this constraint is relaxed and the slicer is allowed to make syntactic changes as long as the relevant semantics are preserved then this is known as amorphous slicing.

Example

The following pseudo code program will be considered:

```
1:     f(int a)
2:     {
3:             int x := a;
4:             int y := 25;
5:             String z := “”;
6:             for (int i:=0; i<x; ++i)
7:          {
8:                  z := z ++ “ “ ++ y;
9:                  y := y + 2 * i;
10:         }
11:
12:         print(x ++ “: “ ++ z ++ “ “ ++ y);
13:     }
```

A backward slice on the criterion (12, {y}) will be found. Statement 12 is a special case (it is obviously useful to include, but in this example y is not modified so is not strictly part of the slice). Working backwards: statement 9 modifies y but 8 does not, therefore we add 9 to the slice but delete 8. The for statement on line 6 must be included since its range of influence includes statement 8, this statement (6) includes the variable x and so statement 3 must be included (as well as statement 4), again statement 5 can be removed since z does not affect the value of y at line 12. the resulting program slice is thus:

```
1:      f(int a)
2:      {
3:          int x := a;
4:          int y := 25;
6:          for (int i:=0; i<x; ++i)
7:          {
9:                  y := y + 2 * i;
10:         }
12:         print(x ++ “: “ ++ z ++ “ “ ++ y);
13:     }
```

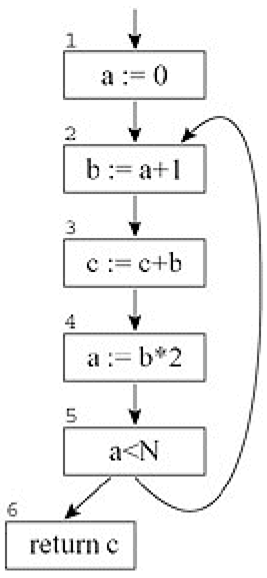
Implementation

Each type of slicing described above requires a different algorithm or the tailoring of an existing algorithm. In addition there are several different algorithms for some of the variants. Each algorithm is language independent, however they will need tweaking for the specific language they must slice due to the different constructs present in the language. In particular language features that rely on indirection (e.g. pointers) or unstructured control flow (e.g. goto) must be addressed.

Weiser’s original algorithm [5] will be covered below. This algorithm computes static backward slices using dataflow analysis. Other algorithms may use information flow relations or dependence graphs [4].

Weiser’s algorithm

Weiser’s algorithm is based on dataflow analysis: the static analysis of a control flow graph. A control flow graph is a representation of a program that contains information about branching and other control flow in a program, it contains little additional information (for program slicing we require the sets of variables modified and those used by each statement). Control flow graphs (and dataflow analysis) are often used by compilers, especially for optimisation.



An example control flow graph taken from [1]

For each statement in the program the algorithm is interested in the edges of the control flow graph (denoted $i \rightarrow_{cfg} j$ where there is an edge between the statements i and j), the set of variables used (<u>ref</u>erenced) by the statement (denoted Ref(i) for a statement i) and the set of variables modified (<u>def</u>ined) by the statement (Def(i)).

Weiser's algorithm is given by the following equations from [4]:

- $R^0_C(i) = V$ when $i = n$.
- For every $i \rightarrow_{\text{CFG}} j$, $R^0_C(i)$ contains all variables v such that either (i) $v \in R^0_C(j)$ and $v \notin \text{DEF}(i)$, or (ii) $v \in \text{REF}(i)$, and $\text{DEF}(i) \cap R^0_C(j) \neq \emptyset$.

$$S^0_C \equiv \{i \mid \text{DEF}(i) \cap R^0_C(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

$$B^k_C \equiv \{b \mid i \in S^k_C,\ i \in \text{INFL}(b)\}$$

$$R^{k+1}_C(i) \equiv R^k_C(i) \cup \bigcup_{b \in B^k_C} R^0_{(b,\text{REF}(b))}(i)$$

$$S^{k+1}_C \equiv B^k_C \cup \{i \mid \text{DEF}(i) \cap R^{k+1}_C(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

R(i) is the set of relevant variables for a statement i, S is the set of relevant statements for the program and B is the set of relevant branching statements for the program. C is the criterion, C = (n, V), as mentioned above. Infl(b) is the set of all statements in the range of influence of a branching statement b, that is, all statements whose execution depends on the result of the branching statement.

The algorithm starts by finding the set of directly relevant variables, $R^0(i)$, for each statement (this set may be empty if there are no directly relevant variables for a statement). From these sets a set of directly relevant statements, S^0, is found by taking all statements that define a variable that is relevant to the following statement in the control flow graph. Next a set of (indirectly) relevant branching statements is found from the set of relevant statements. The algorithm then progresses iteratively, R being found from the last iteration's R and those variables relevant to the branching statements and S and B being found in the same way as above (but S includes the last iteration's B). S will be a non-decreasing subset of the statements in the program being sliced. When S reaches a fixed point between iterations (i.e. S does not increase from one iteration to the next) then this set S is the desired program slice.

Tools

As a debugging aid (and for some of its other applications) program slicing is only really useful when implemented as a tool for the programmer, indeed there are many program slicing tools available. Most tools do not offer comprehensive coverage of the targeted program language and generally implement only the simplest types of program slicing (typically static, syntax preserving slicing). Other variations of program slicing have typically only been implemented over toy languages and is still very much a research area [2]. However, some tools (for example the Wisconsin Program Slicer and Unravel) cover a large subset of the targeted language and are very powerful and useful tools despite these restrictions.

The most widely known program slicing tool is the Wisconsin Program Slicer [10]. It can perform forwards and backwards slicing and chopping of C programs, it can only slice statically. The project is no longer supported but the code base has been evolved into CodeSurfer [7], this is a commercial tool that is multi-platform and (relatively) comprehensive. It can slice C++ (at an alpha stage) and C, again performing forward and backward slicing and chopping.

Unravel [9] is another widely known tool. It can only perform static, backward slicing of C programs, but is fairly comprehensive.

The Bandera project [6] is a project at Kansas State University. Bandera is a model checker that uses program slicing. The Indus project [8] is a spin off from Bandera that contains some of the primitive tools used by Bandera including the Java Program Slicer, this is a library and has been presented as an Eclipse program called Kaveri. This program slicer again does static forward and backward slicing and chopping, but of Java programs. It can handle object orientation and concurrency.

Conclusion

Program slicing is a technique to reduce the complexity of a program in order to aid debugging or comprehension. It has a multitude of uses, variations and implementation methods. In this essay I have covered one particular algorithm for implementing static, backwards slicing that uses dataflow analysis. There are many powerful tools available to perform program slicing and there is also vast scope for improvement in this area as the more modern (and complex) forms of program slicing have not been implemented as usable tools for complete languages.

Bibliography

[1] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, Second Edition*. Cambridge University Press. 2002.

[2] M Harman and R Hierons. An Overview of Program Slicing. *Software Focus*. 2001.

[3] Jeff Russel. Program Slicing Literature Survey. *http://www.ece.utexas.edu/~jrussell/seminar/slicing_survey.pdf*. 2001.

[4] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*. 1995.

[5] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*. 1984.

[6] Bandera Project. *http://bandera.projects.cis.ksu.edu/*.

[7] CodeSurfer. *http://www.grammatech.com/products/codesurfer/*.

[8] Indus Project. *http://indus.projects.cis.ksu.edu/*.

[9] Unravel. *http://hissa.nist.gov/unravel/*.

[10] Wisconsin Program Slicing Tool. *http://www.cs.wisc.edu/wpis/slicing_tool/*.