Flow Grammars – a Flow Analysis Methodology

James S. Uhl and R. Nigel Horspool

Department of Computer Science
University of Victoria[1]

Abstract: Flow grammars provide a new mechanism for modeling control flow in flow analyzers and code optimizers. Existing methods for representing control flow are inadequate in terms of both their generality and their accuracy. Flow grammars overcome these deficiencies and are well-suited to the specification and solution of data flow analysis problems.

1 Introduction

Source programs must be subjected to sophisticated and extensive optimization to approach the full potential of modern computer architectures. Many powerful optimizations rely on *static analysis* of the source code. Examples of such static analysis include: live variable analysis [8], various forms of constant propagation [10,20], and aliasing analysis [4]. All of these static analyses may be realized as instances of *flow analysis problems*. They share the following general framework [10]:

1. A *model* of the program defines points where information is to be determined, as well as specifying how control may flow between those points at run-time.
2. A set of *abstract states* representing the desired static information is created. In the case of constant propagation, for example, each abstract state is a partial mapping from variable names to values. Generally this *information space* is a structured as a lattice or semi-lattice [5,10]. For some analyses, this set of states may even depend upon the structure of the model used to represent the program [17].
3. A set of *flow equations* relating the abstract state of the program at one point with the points that may immediately precede or follow it during execution.

The goal is to find a least fix point solution to the set of equations. The solution is then used to guide the optimization process.

This paper describes a new formalism which we call *flow grammars*, developed for use in the *Flow Analysis Compiler Tool* (FACT), an on-going research project at

1. P.O. Box 3055, Victoria, BC, Canada V8W 3P6. {juhl,nigelh}@csr.uvic.ca

the University of Victoria [19]. Specifically, we demonstrate how context-free flow grammars may be used to model both intra- and inter-procedural control flow; we subsequently show how to interpret a context free grammar as a set of data flow equations; and we give an approach for solving these equations. Finally, we demonstrate that programming language control constructs form a hierarchy which corresponds in a natural manner to a hierarchy of grammar classes.

2 Program Executions

This section motivates the use of grammars to model control flow. The essential idea is to represent each possible execution of a program as a string of symbols, called an *execution path*, where each symbol corresponds to a fixed run-time action that may be made by the program during execution (e.g., an assignment). Following the framework of denotational semantics, each run-time action may be viewed as a function transforming one program state into another. In a *concrete semantics*, the program state consists of the values of all variables, as well as the input and output streams; such states are called *concrete states*. Static analyses typically define an *abstract semantics* where program states contain only the information needed to infer facts about the concrete state at various points of execution; these states are called *abstract states*. Since these facts must be true regardless of which path is followed on a given run, static analyses must consider all possible executions of a program. A mechanism for representing a (potentially unbounded) number of execution paths is needed. Recursive set equations, where each set contains zero or more execution paths, satisfy this requirement. These equations correspond to grammars in formal language theory.

2.1 Execution Paths

An *execution path* represents a single execution path through part of a program. A *complete execution path* represents a single execution of an entire program. Consider the program in Fig. 1, where program points are indicated by numbers to the left of the code. Program point 4, for example, occurs between the assignments "`f:=f*i`" and "`i:=i-1`". Each execution path may be viewed as a sequence of program point pairs. For this program there is one complete execution path (though a given static analysis may not be able to determine this fact):

(1,2) (2,3) (3,4) (4,5) (5,6) (6,3) (3,4) (4,5) (5,7) (7,8)

For better readability, we abbreviate the path description to:

$t_1\ t_2\ t_3\ t_4\ t_{5/6}\ t_6\ t_3\ t_4\ t_{5/7}\ t_7$

The decision at the **if** statement is embodied in the symbols $t_{5/6}$ and $t_{5/7}$: when control proceeds from point 5 to point 6, the true branch has been taken; when control proceeds from point 5 to point 7, the false branch was taken. In this formulation, an execution path is simply a string of symbols, each of which represents one or more run-time actions.

```
    program ex(output);
    label 10;
    var i, f:integer;
    begin
1       i := 3;
2       f := 1;
3   10:    f := f * i;
4          i := i - 1;
5       if i>1 then
6          goto 10;
7       writeln(f)
8   end.
```

Fig. 1. Example Pascal program

2.2 Representation

To perform optimization, a compiler needs to know the (abstract) state of the program at each point *p* irrespective of which execution path was followed to reach *p*. Thus, a compiler must consider the *entire* set of possible execution paths to compute the abstract state at each point in the program.

Suppose S_i is the set of execution paths starting from point *i* in a program and ending at the end of the program. Then the control flow of the source program yields a set of relationships among these sets. For a backwards analysis of the example program, these relationships may be represented by the family of set equations, as shown in Fig. 2. The *least fixed point (lfp) solution* of these equations yields the set of execution paths from each point to the end of the program. Note that there is only one path from point 6 to the end of the program, "$t_6\ t_7$", and, correspondingly, the lfp solution of $S_6 = \{ t_6\ t_7 \}$ and $S_7 = \{t_7\}$. The loop in the program (points 3 through 6) implies that there are an infinite number of possible executions, and thus execution paths in the sets $S_1, S_2, ..., S_6$ each contain an infinite number of paths.

$$S_1 = \{t_1\} \bullet S_2 \qquad S_5 = \{t_{5/6}\} \bullet S_6 \cup \{t_{5/7}\} \bullet S_7$$
$$S_2 = \{t_2\} \bullet S_3 \qquad S_6 = \{t_6\} \bullet S_3$$
$$S_3 = \{t_3\} \bullet S_4 \qquad S_7 = \{t_7\}$$
$$S_4 = \{t_4\} \bullet S_5$$

where $A \bullet B = \{\alpha\,\beta \mid \alpha \in A, \beta \in B\}$

Fig. 2. Backward analysis execution path equations for the example program

In a forward analysis, information is known at the start of execution and must be propagated forward through the program. This is represented by equations that the "mirror" the backwards analysis equations, as shown in Fig. . Here, S_1 is the start of the program, and consequently its least fixed point solution contains only the empty execution path, ε. Similarly, S_2 contains only the execution path t_1, since there is only one path from the start of the program to point 2. The top of the loop, at point 3, introduces a union of paths, one entering the loop from outside (point 2) and the other from the bottom of the loop (point 6). Again the lfp solution to these equations has a structure defined by the equations that yields only execution paths representing possible executions.

$$S_1 = \{\varepsilon\} \qquad S_6 = \{t_{5/6}\} \bullet S_5$$
$$S_2 = \{t_1\} \bullet S_1 \qquad S_7 = \{t_{5/7}\} \bullet S_5$$
$$S_3 = \{t_2\} \bullet S_2 \cup \{t_6\} \bullet S_6 \qquad S_8 = \{t_7\} \bullet S_7$$
$$S_4 = \{t_3\} \bullet S_3$$
$$S_5 = \{t_4\} \bullet S_4$$

Fig. 3. Forward analysis execution path equations for the example program

2.3 Generating Execution Paths with a Grammar

The equations in Fig. 2 correspond exactly to the regular grammar in Fig. 4. What have previously been called "symbols" are now terminals in a grammar. Similarly, the variables in Fig. 2 are now non-terminals.

The set of strings generated by each of the non-terminals in this grammar is equal to the lfp solution of the corresponding equation in Fig. 2. Note also that non-terminals at the end of each production act as *continuations*, indicating where execution is to proceed.

$S_1 ::= t_1\ S_2$
$S_2 ::= t_2\ S_3$
$S_3 ::= t_3\ S_4$
$S_4 ::= t_4\ S_5$
$S_5 ::= t_{5/6}\ S_6$
$S_5 ::= t_{5/7}\ S_7$
$S_6 ::= t_6\ S_3$
$S_7 ::= t_7$

Fig. 4. Grammar generating execution paths for example program

2.4 Semantics

Given a set of equations describing execution paths, along with semantic functions for each symbol, it is possible to define a generalized program semantics that takes into account the behaviour of all possible executions. In this context, the run-time semantics of a program are known as *concrete semantics*, and the flow analysis semantics are called *abstract semantics*.

In a concrete semantics, the semantics of one statement is a function which maps one program state into another program state. For a given statement *S*, the semantic function is written as $[\![S]\!]$. The semantics of an execution path may be defined inductively. In the most basic instance, the semantics of the null statement (represented as the empty string) is the identity function, $I \equiv \lambda S.S$. For a sequence of statements along *one* execution path, the semantic functions associated with each symbol are composed, yielding:

$$[\![S_1 ; S_2 ; \ldots S_n]\!] \equiv [\![S_n]\!] \circ \ldots \circ [\![S_2]\!] \circ [\![S_1]\!]$$

In a flow analysis, the goal is to compute an *abstract state* for each point *p* in the program that summarizes the possible *concrete states* that may occur at *p* during execution. That is, the semantic functions map one abstract state to another and the semantics of *sets* of execution paths are defined. An abstract state is generally a member of a lattice, say L. Given two paths between point *a* and point *b*, P_1 and P_2, the desired abstract semantics is:

$$[\![P_1 \text{ or } P_2]\!] \equiv [\![\{P_1, P_2\}]\!] \equiv [\![\{P_1\}]\!] \wedge [\![\{P_2\}]\!]$$

where $\wedge$ is the *meet* operator used to ensure that the abstract state represents all possible concrete states that may occur, regardless of which path is taken. In practice, the precision implied in this equation can become arbitrarily expensive to compute and approximation must be used to limit the computation time.

For example, for S_1 in Fig. , we have:

$$[\![S_1]\!] \equiv [\![\{\varepsilon\}]\!] \equiv \lambda S.S$$

and, thus, $[\![S_1]\!] \equiv \lambda S.S$. For S_2, the semantic function for S_1 must be composed with the meaning of the set $\{t_1\}$:

$$\begin{aligned} [\![S_2]\!] &= [\![\{t_1\}]\!] \circ [\![S_1]\!] \\ &= [\![\{t_1\}]\!] \circ (\lambda S.S) \\ &= [\![\{t_1\}]\!] \end{aligned}$$

Here the semantics of the singleton set $\{t_1\}$ is composed with the identity function, which is just the semantics of the singleton set itself.

When two sets are merged in a union, as in the equation for S_3 above, the *meet* of the corresponding functions must be computed. This is typically the *point-wise meet*, $\bigwedge:(L\rightarrow L)\rightarrow(L\rightarrow L)$, of the functions, defined as:

$$f_1 \bigwedge f_2 = \lambda S.f_1(S) \wedge f_2(S)$$

where $\wedge:L\rightarrow L$ is the meet operator of the underlying lattice. In the case of S_3, the equation is:

$$[\![S_3]\!] = ([\![\{t_2\}]\!] \circ [\![S_2]\!]) \bigwedge ([\![\{t_6\}]\!] \circ [\![S_6]\!])$$

Figure 5 shows the complete set of forward analysis equations for the example program. The goal of static analysis is to compute the portion of the least fixed point functionals for each of these equations needed to determine the abstract state at each point in the program.

$$\begin{aligned} [\![S_1]\!] &= [\![\{\varepsilon\}]\!] \\ [\![S_2]\!] &= [\![\{t_1\}]\!] \circ [\![S_1]\!] \\ [\![S_3]\!] &= ([\![\{t_2\}]\!] \circ [\![S_2]\!]) \bigwedge ([\![\{t_6\}]\!] \circ [\![S_6]\!]) \\ [\![S_4]\!] &= [\![\{t_3\}]\!] \circ [\![S_3]\!] \end{aligned}$$

$$\begin{aligned} [\![S_5]\!] &= [\![\{t_4\}]\!] \circ [\![S_4]\!] \\ [\![S_6]\!] &= [\![\{t_{5/6}\}]\!] \circ [\![S_5]\!] \\ [\![S_7]\!] &= [\![\{t_{5/7}\}]\!] \circ [\![S_5]\!] \\ [\![S_8]\!] &= [\![\{t_7\}]\!] \circ [\![S_7]\!] \end{aligned}$$

Fig. 5. Semantic equations for forward analysis

3 Definition: Flow Grammar

A *flow grammar* is a quadruple $G=(\Sigma_N, \Sigma_T, P, S)$ where: Σ_N is the set of *flow non-terminals*, Σ_N, corresponding to the program points; Σ_T is the set of *flow terminals* corresponding to run-time actions such as assignments; P is a set of *flow productions* of the form $\alpha ::= \beta$, where $\alpha \in \Sigma_N^+$ and $\beta \in (\Sigma_T \cup \Sigma_N)^*$; and S is the *flow start symbol* and corresponds to the beginning of the program.[1]

3.1 Example: Interprocedural Control Flow

Interprocedural analysis requires a context-free flow grammar to model the matching of calls and returns. Figure 6 shows a small Pascal program which is used to demonstrate interprocedural data flow analysis in the next section. Flow productions modeling the example program are shown in Fig. 7. Of particular importance are the productions corresponding to the procedure calls, "$S_4 ::= t_{4/1}\ S_1\ t_{6/5}\ S_5$" and "$S_8 ::= t_{8/1}\ S_1\ t_{6/9}\ S_9$". Both of these productions have an embedded non-terminal, S_1, representing entry to the procedure being called. Terminals $t_{4/1}$ and $t_{8/1}$ represent the actions that occur on procedure entry from points 4 and 8, respectively. Similarly, $t_{6/5}$ and $t_{6/9}$ represent the actions that occur upon return to the respective call sites.

```
   program example(output);
   var f,i:integer;
   procedure nfact;
   begin
1      if i<=1 then
2         f := 1
       else begin
3         i := i-1;
4         nfact;
5         f := f*i
       end
6  end;
```

```
      (* this is an incorrect
         implementation of
         factorial *)


   begin
 7     i:=5;
 8     nfact;
 9     write(f)
10 end.
```

Fig. 6. Example Pascal program

$S_1 ::= t_{1/2}\ S_2$
$S_1 ::= t_{1/3}\ S_3$
$S_2 ::= t_2\ S_6$
$S_3 ::= t_3\ S_4$
$S_4 ::= t_{4/1}\ S_1\ t_{6/5}\ S_5$
$S_5 ::= t_5\ S_6$
$S_6 ::= t_6$

$S_7 ::= t_7\ S_8$
$S_8 ::= t_{8/1}\ S_1\ t_{6/9}\ S_9$
$S_9 ::= t_9\ S_{10}$
$S_{10} ::= t_{10}$

Fig. 7. Flow grammar for program of Fig. 6

---

1. Σ^* is the set of all strings over Σ, including the empty string ε; Σ^+ is the set of all non-empty strings over Σ.

Note that procedure calls are handled naturally in the semantic equations, in the case of the call from the main program, we have:

$$[\![S_8]\!] = [\![\{t_{8/1}\}]\!] \circ [\![S_1]\!] \circ [\![\{t_{6/9}\}]\!] \circ [\![S_9]\!]$$

3.2 Discussion

It is interesting to consider the relationship between the Chomsky hierarchy [7] and various programming language constructs. The boundary between the context-free (type 2 in the hierarchy) and context-sensitive (type 1) flow grammars is important because the former admit the straightforward translation to flow equations shown above, but the latter do not.

A regular flow grammar (type 3) corresponds directly to a flow graph, and is therefore capable of representing the same intraprocedural constructs, including if/then/else, loops, and gotos to (constant) labels. Label variables are somewhat anomalous. At the expense of increased size, a regular flow grammar can precisely (up to the usual assumption that all control flow choices are possible at any branch, and that any one branch is independent of any previous branch) model intraprocedural control flow containing only simple label variables. The idea is to encode the current state of all label variables into each non-terminal of the flow grammar. This results in a finite number of non-terminals, because there must be a finite number of simple label variables, each of which can assume a finite number of label values. When an assignment to a label variable occurs, the productions ensure the continuation non-terminal encodes the correct state. Note, however, that label variables in arrays and other dynamic structures cannot be precisely tracked in this manner using a regular flow grammar (although conservative approximate tracking that takes account of aliasing is possible).

Context-free flow grammars add the key capability of modeling procedure calls and returns, making them suitable for many interprocedural flow analysis problems. A finite number of simple procedure variables may be directly encoded into the non-terminals similar to the encoding of label variables above. A goto statement whose (constant) target is not local causes premature termination of one or more activation records, including their suspended continuations. Surprisingly, this can be modeled with a context-free flow grammar by creating productions that generate prefixes of execution paths that eventually end with a production representing the non-local goto. Ginsburg and Rose show that the language of all proper prefixes of a context-free language is itself context-free [6], validating the assertion that such control flow is still

context-free; details relating this result to flow grammar construction may be found in [19].

We note several important aspects of the flow grammar methodology:

1. Interprocedural and intraprocedural control flow are unified into a single all-encompassing model.
2. Results from formal language theory are useful when projected into patterns of control flow. For example, in-line expansion may be effected by the elimination of a production.
3. The structure of regular and context-free flow grammars naturally reflects a set of flow equations; a data flow analysis simply interprets the terminals and non-terminals in appropriate domains.

3.3 Interprocedural Data Flow Analysis: an Example

Within a flow grammar, each non-terminal represents an execution point in the program. As shown above, each non-terminal may be interpreted as representing the *state* of the program at that point. The state should, of course, capture just the information that is relevant to the data flow problem that we are interested in. For the *live variables* problem, the state is usually described by the set of variables that are *live* at a program point (i.e., there exists a path from that point to a use of that variable without an intervening assignment to the variable).

Translating our example context-free flow grammar to a set of backwards flow equations (as would be needed for solving the live variables problem) yields the family of equations in Fig. 8.

As a concrete example of the general technique, we now consider the specific problem of determining variable liveness at a given point in the program. For *intraprocedural* live variables, the lattice $L = 2^V$, where $V = \{i, f\}$, suffices, so that each $S_i \in 2^{\{i, f\}}$. The meet operator $\wedge$ is set union. With this interpretation and for the particular problem of live variables, the effects of the terminals may be described by set equations corresponding to "gen" and "kill" sets [1]; for example, t_5 represents the execution of the statement "`f:=f*i`" which kills `f` and then generates `f` and `i`, thus:

$$[\![\{t_5\}]\!] = \lambda x . ((x - \{f\}) \cup \{f, i\}) = \lambda x . (x \cup \{f, i\})$$

Figure 9 shows the abstract semantic functions for all the terminals in the example program.

As described above, a richer domain is required for a precise *interprocedural* analysis. In general, the effect of a statement inside a function depends on the environment

$$
\begin{aligned}
[\![S_1]\!] &= [\![\{t_{1/2}\}]\!] \circ [\![S_2]\!] \wedge [\![\{t_{1/3}\}]\!] \flat [\![S_3]\!] \\
[\![S_2]\!] &= [\![\{t_2\}]\!] \circ [\![S_6]\!] \\
[\![S_3]\!] &= [\![\{t_3\}]\!] \circ [\![S_4]\!] \\
[\![S_4]\!] &= [\![\{t_{4/1}\}]\!] \circ [\![S_1]\!] \circ [\![\{t_{6/5}\}]\!] \circ [\![S_5]\!] \\
[\![S_5]\!] &= [\![\{t_5\}]\!] \circ [\![S_6]\!] \\
[\![S_6]\!] &= [\![\{t_6\}]\!] \\
[\![S_7]\!] &= [\![\{t_7\}]\!] \circ [\![S_8]\!] \\
[\![S_8]\!] &= [\![\{t_{8/1}\}]\!] \circ [\![S_1]\!] \circ [\![\{t_{6/9}\}]\!] \circ [\![S_9]\!] \\
[\![S_9]\!] &= [\![\{t_9\}]\!] \circ [\![S_{10}]\!] \\
[\![S_{10}]\!] &= [\![\{t_{10}\}]\!]
\end{aligned}
$$

Fig. 8. Backwards flow equations for program in Fig. 6

$$
\begin{aligned}
[\![\{t_{1/2}\}]\!] &= \lambda x \,.\, (x \cup \{i\}) & [\![\{t_{6/5}\}]\!] &= \lambda x \,.\, x \\
[\![\{t_{1/3}\}]\!] &= \lambda x \,.\, (x \cup \{i\}) & [\![\{t_{6/9}\}]\!] &= \lambda x \,.\, x \\
[\![\{t_2\}]\!] &= \lambda x \,.\, (x - \{f\}) & [\![\{t_7\}]\!] &= \lambda x \,.\, (x - \{i\}) \\
[\![\{t_3\}]\!] &= \lambda x \,.\, (x \cup \{i\}) & [\![\{t_{8/1}\}]\!] &= \lambda x \,.\, x \\
[\![\{t_{4/1}\}]\!] &= \lambda x \,.\, x & [\![\{t_9\}]\!] &= \lambda x \,.\, (x \cup \{f\}) \\
[\![\{t_5\}]\!] &= \lambda x \,.\, (x \cup \{f, i\}) & [\![\{t_{10}\}]\!] &= \lambda x \,.\, x
\end{aligned}
$$

Fig. 9. Abstract semantic functions for terminal symbols

of the function call. Therefore, we use a domain whose elements have the form *environment* $\rightarrow$ *state* to provide the values associated with terminals and non-terminals of the flow grammar. For the live variables problem, both the state and the environment may be represented by a set of live variables. I.e., the environment of a function call is the state at the point of call. In this case, the set of variables that are live on return from the function. Thus all values, for both terminals and non-terminals, belong to the domain of functions, $2^V \rightarrow 2^V$. However, it is unnecessary to compute the function values fully. We only need to know the effects of statements inside a function for those invocation environments that actually occur. Thus, our iterative approach for finding a fixpoint is demand-driven and, in general, only partially computes the functions.

Our analysis uses the fact that no variable in the program is live at the point where the program terminates. (If a program sets a status variable that could be inspected by

the operating system on return, such a variable would be deemed to be live at program point S_{10}). The iteration to compute the functions proceeds as follows. Each value shows what is known about the various functions at each iteration. Suppose that, in the course of an iteration, $[\![S_2]\!]$currently has the value { {f}→{}, {f, i}→{i} }. This would indicate that the function containing point S_2 is currently known to have two calling environments, {f} and {f, i}. That is, in one set of calls to the enclosing function, f is the only live variable on exit and in another set of calls, both f and i are live on exit. When that function is invoked in the {f} environment, the set of live variables at point S_2 is {}; similarly the calling environment {f, i} gives {i}. If the value of $[\![S_2]\!]$ is shown as the empty set Φ, this corresponds to the bottom element of the enriched lattice and means that no invocations of the function containing point S_2 have been processed yet.

Initially, we want to know what is live at the beginning of the program, and this is represented by $[\![S_7]\!](\{\})$. That is, S_7 is contained in the main program and the environment for the main program is an empty set – no variables are live on exit. The demand for $[\![S_7]\!](\{\})$ triggers the addition of {}→{} to $[\![S_{10}]\!]$, and initiates the first iteration, which proceeds as follows:

1. $[\![S_9]\!](\{\})$ is computed from $[\![S_{10}]\!](\{\})$ yielding {f}, which is added to $[\![S_9]\!]$.
2. The computation of $[\![S_8]\!](\{\})$ requires the value of $[\![S_1]\!](\{f\})$. Since $[\![S_1]\!]$ is currently Φ, this triggers the addition of {f}→{} to $[\![S_6]\!]$, and the (optimistic) value {} is used for $[\![S_1]\!](\{f\})$.
3. The iteration concludes with the empty set as the current value of $[\![S_7]\!](\{\})$.

Figure 10 shows the entire computation.

Within a given iteration, partial function values are computed in the order shown in the table. Note that this particular order leads to rapid convergence; other orders will yield the same solution but will usually require more iterations.

Reading from the final column of the table, we can deduce that there are no live variables at the beginning of the program, since $[\![S_7]\!](\{\}) = \{\}$. (That is, if there are no live variables at the end of execution, then there are no live variables at the start of execution.) This result is a simple application of live variable analysis which proves that all variables are initialized before being used. The final column also shows which variables are live at each program point. Given a function value F for some program point p, then the set of variables that are live at p is computed as $\bigcup_{X \to Y \in F} Y$. For example, the set of live variables at point S_2 is {i}; in one calling environment, the set is empty

State	Iteration Number			
	1	2	3	4
$[\![S_6]\!]$	Φ	{f}→{f} }	{f}→{f}, {f,i}→{f,i} }	{f}→{f}, {f,i}→{f,i} }
$[\![S_5]\!]$	Φ	{f}→{f,i} }	{f}→{f,i}, {f,i}→{f,i} }	{f}→{f,i}, {f,i}→{f,i} }
$[\![S_4]\!]$	Φ	{f}→{} }[b]	{f}→{}, {f,i}→{} }	{f}→{i}, {f,i}→{i} }
$[\![S_3]\!]$	Φ	{f}→{i} }	{f}→{i}, {f,i}→{i} }	{f}→{i}, {f,i}→{i} }
$[\![S_2]\!]$	Φ	{f}→{} }	{f}→{}, {f,i}→{i} }	{f}→{}, {f,i}→{i} }
$[\![S_1]\!]$	Φ	{f}→{i} }	{f}→{i}, {f,i}→{i} }	{f}→{i}, {f,i}→{i} }
$[\![S_{10}]\!]$	{}→{} }	{}→{} }	{}→{} }	{}→{} }
$[\![S_9]\!]$	{}→{f} }	{}→{f} }	{}→{f} }	{}→{f} }
$[\![S_8]\!]$	{}→{} }[a]	{}→{i} }	{}→{i} }	{}→{i} }
$[\![S_7]\!]$	{}→{} }	{}→{} }	{}→{} }	{}→{} }

[a] And add element {f}→{} to set $[\![S_6]\!]$.

[b] And add element {f, i}→{} to set $[\![S_6]\!]$.

Fig. 10. Example live variable computation

and in the only other environment, the set is {i}, taking their union yields the desired answer.

3.4 An Iteration Strategy

An obvious method for speeding convergence of the iteration is to ensure that whenever a computation is performed, as many as possible of its abstract inputs are already computed. This is precisely what the techniques of Jourdan and Parigot to solve "grammar flow analysis" problems [9] yield. Combining these methods with the algorithm of Sharir and Pnueli [17, pp. 207-209] results in an effective solution procedure. In essence, the flow grammar is partitioned into a set of sub-components that encapsulate recursion, resulting in a directed acyclic graph. Iteration is then performed by visiting the sub-components in reverse topological order.

3.5 Handling Arguments to Procedures

Arguments to procedures are, in general, handled by defining an appropriate lattice and mappings for the call/return/exit terminals. The bit-vector technique of Knoop and Steffen [13], for example, may be applied directly. As more than one flow analysis

specification may be incorporated into the compiler, determination of aliasing may be performed before subsequent analysis to ensure conservative solutions.

4 Previous Work

Previous work on control flow analysis is limited; most effort has been devoted to various aspects of data flow analysis. As mentioned above, graphs are the most frequently discussed mechanism for representing control flow [5,10,12,14,17] and *graph grammars* [11] were considered for use in FACT. Graph grammars are effective for representing hierarchical control structures, but cannot handle the arbitrary control flow made possible by the **goto** statement, and also cannot effectively match calls with returns.

Languages with various flavours of procedure variables provide many challenges to effective flow analysis. Weihl's approach ignores local control flow effects to derive a conservative approximation of the possible values each procedure variable may have [21]. Shivers addresses the difficult task of determining a control flow model for a Scheme program where all functions are bound dynamically [18].

The task of specifying control flow in terms of syntax is addressed by Sethi's *plumb* project [16]. In essence, plumb allows a continuation passing style semantics to be specified for a programming language using a special function composition operator. Flow grammars can also be considered as representing control flow using continuations, but in a more direct manner.

Work on static analysis in the form of data flow analysis and abstract interpretation is extensive. Performing flow analysis at the source level ("high-level data flow analysis") for specific data flow problems has been considered by Rosen [15] and Babich and Jazayeri [2,3]. Generalization of various related flow analysis techniques into uniform frameworks includes the work of the Cousots [5], Kam and Ullman [10] and Kildall [12]. Marlowe and Ryder provide an excellent survey of data flow analysis problems and the computational cost of their solutions in [14].

Yi and Harrison describe a tool called Z1 [22] whose goals are similar to those of FACT. Static analyses in Z1 are specified by defining an (abstract) interpreter along with appropriate lattices.The abstract interpreter operates on an intermediate representation of the source program in which gotos have been eliminated. Z1's novel lattice definition mechanism is of particular interest: though somewhat restrictive, it allows the precision of the analysis to be controlled by projecting the analysis lattices. FACT differs from Z1 by not requiring the explicit writing of an abstract interpreter. Instead,

the user specifies the construction of a flow grammar in terms of the abstract syntax of the source language and an analysis on the resulting grammar.

5 Discussion and Future Work

Our main achievement has been to integrate intraprocedural and interprocedural flow analysis in a seamless manner. Flow grammars not only represent control flow effectively, but are directly amenable to specifying data flow analysis problems as well. We argue that, in a general purpose tool such as FACT, it is appropriate to begin with an accurate control flow model and lose precision at the data flow analysis stage; rather than lose precision even prior to data flow analysis by constructing an inaccurate control flow model.

Flow grammars open up a variety of avenues for future research. Preliminary work on modeling programs containing more diverse language constructs, such as exception handlers and bounded numbers of procedure variables, is encouraging. Aside from unrestricted flow grammars, we are also considering the use of two-level grammars to model the dynamic nature of procedure variables. While not discussed in this paper, we have found examples of programs for which control flow is naturally modeled by Type 0 grammars.

Work is proceeding on the algorithm to solve the flow problems generated from flow grammars. Because FACT is intended to be general purpose, minimal assumptions are made about the data flow analysis framework: that the lattice is of finite height and that all functions are monotonic. Currently under investigation is an algorithm which computes the effect of a function call using iteration for up to k different elements in the input domain, and then uses conservative approximations when necessary for subsequent inputs. The use of the restricted lattices as found in Z1 is also under investigation.

References

[1] Aho, A., R. Sethi and J. Ullman. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing, 1986.

[2] Babich, W. and M. Jazayeri. "The Method of Attributes for Data Flow Analysis Part I:Exhaustive Analysis," *Acta Informatica* 10, 1978, pp. 245-264.

[3] Babich, W. and M. Jazayeri. "The Method of Attributes for Data Flow Analysis Part II: Demand Analysis," *Acta Informatica* 10, 1978, pp. 265-272.

[4] Cooper, K., K. Kennedy and L. Torczon. "The Impact of Interprocedural Analysis and Optimization in the $\mathbf{R}^n$ Programming Environment," *ACM TOPLAS* 8, 4, October 1986, pp. 491-523.

[5] Cousot, P. and R. Cousot. "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *4th POPL*, January 1977, pp. 238-252.

[6] Ginsburg, S. and G. F. Rose. "Operations which preserve definability in languages," *JACM* 10(2), April 1963, pp. 175-195.

[7] Hopcroft, J. E. and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 1979, pp. 217-228.

[8] Hecht, M. *Flow Analysis of Computer Programs*, Elsevier, 1977.

[9] Jourdan, M. and D. Parigot. "Techniques for Improving Grammar Flow Analysis," *ESOP'90, LNCS 432*, pp. 240-255.

[10] Kam, J. and J. Ullman. "Monotone Data Flow Analysis Frameworks," *Acta Informatica* 7, 1977, pp. 305-317.

[11] Kennedy, K. and L. Zucconi. "Applications of a Graph Grammar for Program Control Flow Analysis," *4th POPL*, January 1977, pp. 72-85.

[12] Kildall, G. "A Unified Approach to Global Program Optimization," *(1st) POPL*, October 1973, pp. 194-206.

[13] Knoop, J. and B. Steffen. "The Interprocedural Coincidence Theorem," *4th Intl. Conf., CC'92*, October 1992, pp. 125-140.

[14] Marlowe, T. and B. Ryder. "Properties of Data Flow Frameworks," *Acta Informatica* 28, 1990, pp. 121-163.

[15] Rosen, B. "High-Level Data Flow Analysis," *CACM* 20, 10, October 1977, pp. 712-724.

[16] Sethi, R. "Control Flow Aspects of Semantics-Directed Compiling," *ACM TOPLAS* 5, 4, October 1983, pp. 554-595.

[17] Sharir, M. and A. Pnueli. "Two Approaches to Interprocedural Data Flow Analysis," in *Program Flow Analysis: Theory and Applications*, Muchnick S. and Jones N. (eds.), 1981, pp. 189-233.

[18] Shivers, O. "Control Flow Analysis in Scheme," *PLDI'88*, June 1988, pp. 164-174.

[19] Uhl, J. S. *FACT: A Flow Analysis Compiler Tool.* Ph.D. Dissertation, in preparation.

[20] Wegman, M. and F. Zadeck. "Constant Propagation with Conditional Branches," *ACM TOPLAS* 13, 2, April, 1991, pp. 181-210.

[21] Weihl, W. "Interprocedural Data Flow Analysis in the Presence of Pointer, Procedure Variables, and Label Variables," *7th POPL*, January 1980, pp. 83-94.

[22] Yi, K. and Harrison, W. L. "Automatic generation and management of interprocedural program analyses," *20th POPL*, January 1993, pp. 246-259.