Flow Grammars: A Methodology for Automatically Constructing Static Analyzers

by

James S. Uhl
B.Sc., University of Calgary, 1987
M.Sc., University of Victoria, 1989

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science.

We accept this thesis as conforming
to the required standard

Dr. R. N. S. Horspool, Co-supervisor (Department of Computer Science)

Dr. H. A. Müller, Co-supervisor (Department of Computer Science)

Dr. W. W. Wadge (Department of Computer Science)

Dr. V. Bhargava (Department of Electrical and Computer Engineering)

Dr. G. V. Cormack (University of Waterloo)

Co-supervisors: Dr. R. N. S. Horspool, Dr. H. A. Müller

ABSTRACT

A new control flow model called flow grammars is introduced which unifies the treatment of intraprocedural and interprocedural control flow. This model provides excellent support for the rapid prototyping of flow analyzers. Flow grammars are an easily understood, easily constructed and flexible representation of control flow, forming an effective bridge between the usual control flow graph model of traditional compilers and the continuation passing style of denotational semantics. A flow grammar semantics is given which is shown to summarize the effects all possible executions generated by a flow grammar conservatively. Various interpretations of flow grammars for data flow analysis are explored, including a novel bidirectional interprocedural variant. Several algorithms, based on a similar technique called grammar flow analysis, for solving the equations arising from the interpretations are given. Flow grammars were developed as a basis for FACT (Flow Analysis Compiler Tool), a compiler construction tool for the automatic construction of flow analyzers. Several important analyses from the literature are cast in the flow grammar framework and their implementation in a FACT prototype is discussed.

Examiners:

Dr. R. N. S. Horspool, Co-supervisor (Department of Computer Science)

Dr. H. A. Müller, Co-supervisor (Department of Computer Science)

Dr. W. W. Wadge (Department of Computer Science)

Dr. V. Bhargava (Department of Electrical and Computer Engineering)

Dr. G. V. Cormack (University of Waterloo)

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, my thanks go to co-supervisors Nigel Horspool and Hausi Müller for all their help and encouragement. Thanks also to my other committee members, Vijay Bhargava and especially Bill Wadge for valuable constructive criticism. My family members have been a constant source of encouragement and inspiration throughout my academic career, for which I am very grateful. Major benefit has come from many students over the years; the following people, in particular, provided helpful insights into this research and the research process itself: Eric Davies, Carlos Escalante, Philipp Heuberger, Brian Koehler, Jan Vitek and Mike Zastre. Thanks go as well to the systems administrators, Gord Broom, Gary Duncan, Alan Idler, Will Kastelic and Mark McIntosh, for keeping the systems running and providing the software I needed for my work. Finally, I would like to thank Mike Miller for his rational and objective advice at several key times.

Financial support from the Natural Sciences and Engineering Research Council, the British Columbia Advanced System Institute, International Business Machines Corporation and the University of Victoria made this work possible.

I Introduction

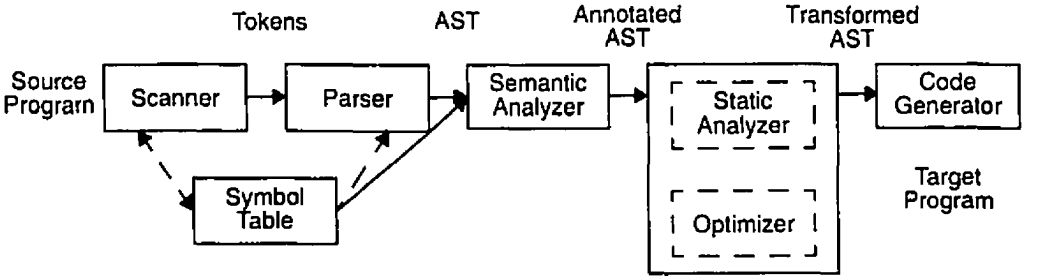
This dissertation introduces *flow grammars*, a uniform mechanism for describing intraprocedural and interprocedural control flow to aid in the prototyping of flow analyzers. Flow grammars address the problem of describing control flow in a straightforward manner, much as regular expressions may be used to describe the lexical elements of a programming language. Intended for use in a compiler generation tool, flow grammars have proven valuable in their own right as a structuring mechanism for prototyping flow analyzers in an existing compiler. The simple structure of a flow grammar captures the essence of a program being compiled; various interpretations of a flow grammar permit a broad range of flow analyses to be performed.

I.1 Compiler construction environments and static analysis

Unlike most software development, compiler construction is more science than art. Indeed, the overall structure of most compilers (Figure 1) is essentially the same, although many of the low-level details may differ. Compiler generation tools exist for many of the

phases depicted in Figure 1, including scanner generators [33, 35, 58, 67], parser generators [36, 40, 67], code generator generators [69], and optimizer generators [23, 86, 94, 96]. Effective optimization, however, requires considerable static analysis of the program being compiled [3] and there has even been some success with automating the generation of static analyzers [28, 73, 81, 86, 97, 98]. Unfortunately, these latter tools are frequently designed to fit into a particular compiler or compiler generation environment, and make assumptions that prevent their straightforward use in other contexts. One key problem is the lack of an appropriate abstraction for specifying *control flow*. This dissertation introduces such an abstraction and describes the design and prototype implementation of a *Flow Analysis Compiler Tool* (*FACT*), directed at generating analyzers for imperative languages.

Figure 1 Structure of a typical compiler



1.2 Static analysis and its role in compilation

Figure 1 shows the structure of a typical compiler as assumed by a compiler generation system such as Eli [27]. The first phase *scans* the text of a *source language* program, combining characters into the tokens of the language. These tokens are then passed to the *parser* where their syntactic structure is determined. During these processes, a *symbol table* is built containing information about the user defined symbols in the program. In addition, the parser constructs an *abstract syntax tree* (*AST*) representing the program. The

tree and symbol table are then passed to the semantic analyzer which *annotates* or *decorates* the nodes in the AST with information about the program fragment represented by each node. If these initial phases are successful, various transformations, possibly including a set of *optimizations*, are applied to the AST. The final of the transformations is *code generation* which yields an equivalent program in the *target language*.

Optimizing transformations are important for a number of reasons. Perhaps the most compelling is that effective use of current hardware requires considerably more effort than in the past. Programs running on *reduced instruction set computer* (*RISC*) and *vector* architectures, for example, can benefit greatly by matching patterns of control and data flow inherent in the source program with specific features of the hardware. In order to perform this mapping effectively, close examination of the source program is needed to compute the required patterns. Such analysis is called *static analysis*. In the prototyping environment provided by a compiler generation system, it would be very useful to be able to generate such analyses automatically from a high level specification and then determine the utility of the analysis in performing optimizing transformations empirically.

1.3 Problem

The goal of this research is the development of a tool for automating the creation of static analyzers in a compiler generation environment. To be successful, such a tool should meet at least the following requirements, addressing the needs of imperative language compilation:

1. It must be based on an abstraction appropriate to specifying control flow in a programming language. Ideally, this abstraction would: a) allow the seamless integration of intraprocedural control flow (that is, within a single procedure) and interprocedural control flow (that is, procedure calls and returns) in a sin-

gle model; and b) be easily described in terms of objects present in a compiler generation environment.

2. The range of possible data flow analyses should not be unnecessarily constrained. It should be possible to specify several analyses that may, in general, be interdependent.
3. Results of an analysis should be incorporated back into the other internal representations of the program (generally the AST) for use in the optimizer. If properly designed, the model of 1 above would be useful as an intermediate representation in its own right.

I.4 Approach

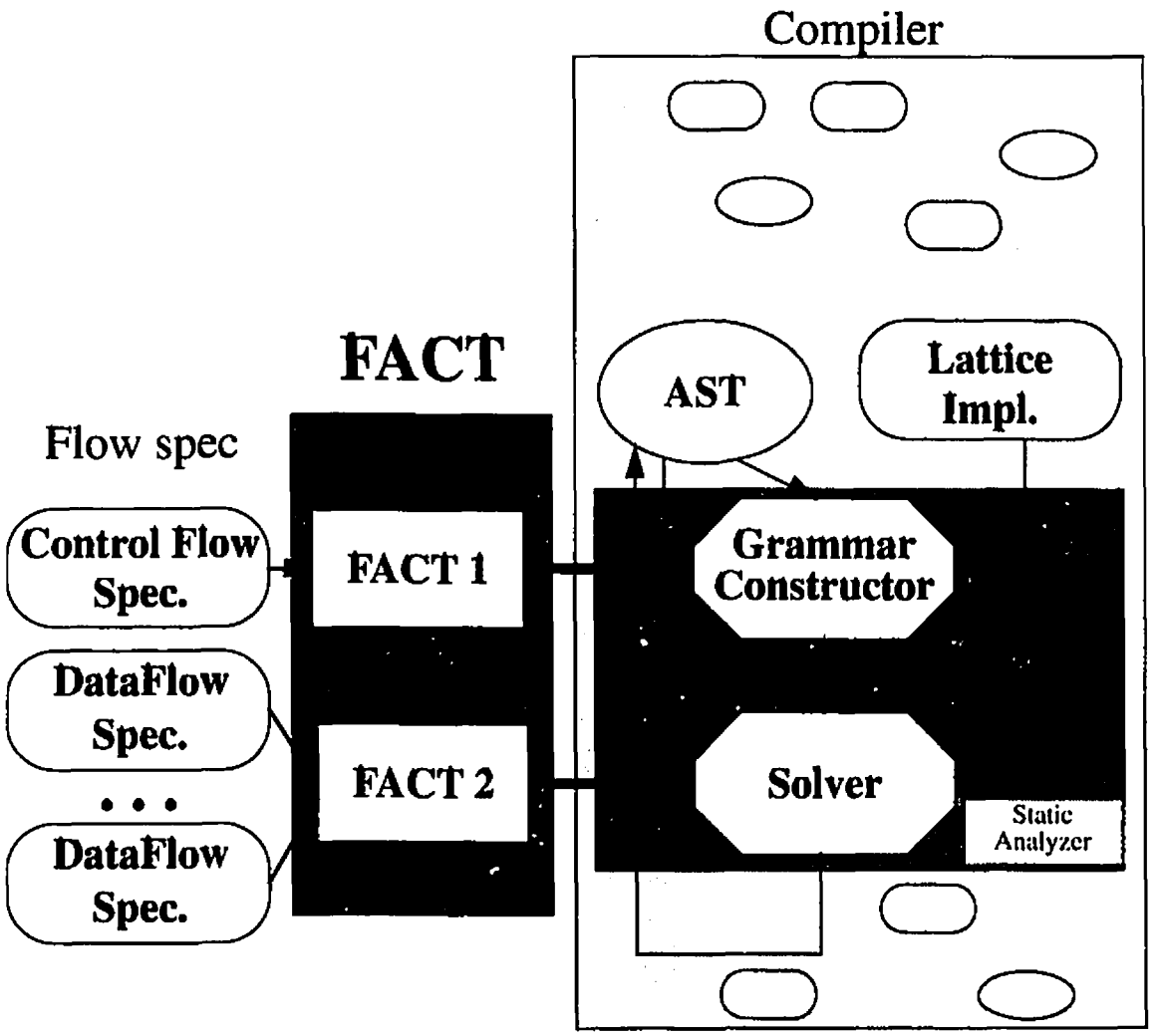
The approach taken in this dissertation is to represent control flow in a program as a *flow grammar*. *Trace semantics* [34], used initially to describe communicating sequential processes, provide the intuition for flow grammars. Because a trace semantics is easily specified using schema, it meets the first requirement above. In addition, when represented as a context free grammar, a trace semantics may be manipulated in various ways to reduce the machine resources needed to perform a static analysis. Indeed, a flow grammar may even be used to representation triples or quads [3], and is thus suited to performing low level optimizations such as *common subexpression elimination* and *code motion* (movement of invariant computations out of loops).

In general, a data flow analysis specifies an *abstract semantics* and the goal is to compute a value representing possible program states for each point in the program. Applying this technique to a flow grammar yields what is essentially a *continuation passing style semantics* [82] defined over an abstract domain.

Figure 2 shows the relation between FACT and the generated compiler. An analysis is specified in two parts: a *control flow specification* and one or more *data flow specifications*. FACT itself also consists of two components: the first builds a grammar

construction module and the second builds a module which constructs and solves the desired data flow analyses. Implicit in the construction is a lattice implementation for each analysis. Common lattices are provided, but user-implemented lattices are also permitted.

Figure 2 The FACT compiler tool



I.5 Contributions

The main contributions of this dissertation are:

1. A *uniform* control flow model integrating intraprocedural an interprocedural control flow called *flow grammars.*
2. *Interpretations* of flow grammars permitting a *broad range* of *flow analyses* to be performed.
3. *Algorithms* for solving the equation systems resulting from flow grammar interpretations.
4. *Validation* of the flow grammar model by demonstrating the use of the model in solving a variety of flow analysis from the literature.
5. A prototype *implementation* of FACT, based on flow grammars, a tool for the automatic construction of flow analyzers, including implementations of several analyses from the literature.

I.6 Overview

The remainder of this dissertation begins with a survey of related work and relevant notation in Chapter II. Following this, Chapter III illustrates how control flow may be specified in terms abstract syntax using an intuitive graphical notation that is remarkably similar to context free grammars. Chapter IV elaborates on this notion and shows how the semantics of an imperative programming language may be specified in terms of a flow grammar, and how this relates to continuation passing style semantics. A detailed investigation of performing interprocedural data flow analysis in the flow grammar realm appears in Chapter V. Adaptation of a variety of existing analyses in a prototype implementation of the FACT tool is described in Chapter VI. Finally, contributions and avenues of future research are described in Chapter VII.

II Background

The fields of compiler generation and flow analysis both have a long history. Efforts to construct tools for automating the construction of compilers go back to the mid-1970s and beyond [30]. Flow analysis as a systematic technique dates from Kildall's seminal paper in 1971 [51]. After an initial section on notation and terminology, this chapter presents an overview compiler generation and flow analysis, paying particular attention to the intersection: tools for building flow analyzers.

II.1 Notation and terminology

This section introduces the terminology and notation used throughout the remainder of the dissertation. Areas include graph theory, formal language theory, lattice theory, attribute grammars and denotational semantics. Throughout, angle brackets, ‹ and ›, are used to enclose tuples, such as the pair consisting of the numbers 1 and 2: ‹1,2›.

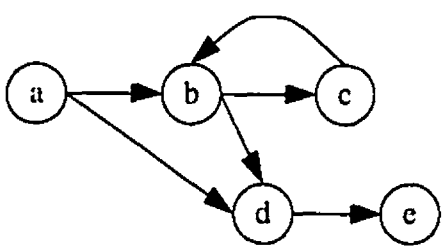
II.1.1 Discrete mathematics

First we introduce a few important definitions from discrete mathematics [8], along with brief descriptions of their use in this work.

Definition 1: A *graph* is a pair ‹N,E› where N is a set of *nodes* and $E \subseteq N \times N$ is a set of *(directed) arcs* or *edges*.

Graphs are frequently represented visually. Figure 3 show a graph with five nodes (circles) and six arcs (arrows). As shown, nodes are frequently labelled; occasionally arcs are labelled with adjacent text.

Figure 3 Example of a graph



N = {a,b,c,d,e}
E = {‹a,b›,‹a,d›,
‹b,c›,‹b,d›,
‹c,b›,‹d,e›}

Definition 2: A *path* in a graph ‹N,E› is set of edges $\langle n_1,n_2\rangle,\langle n_2,n_3\rangle,\ldots,\langle n_{m-1},n_m\rangle$.

Definition 3: A *multi-graph* is a pair ‹N,E› along with two functions, *head* : $E \rightarrow N$ and *tail* : $E \rightarrow N$ where N is a set of nodes and E is a set of arcs and the functions head and tail indicate the source and target nodes of each arc, respectively.

Note the distinction between a graph and a multi-graph: the former allows only a single arc between any pair of nodes, while the latter admits an arbitrary number of arcs between any pair of nodes.

Definition 4: For a given set X, 2^X represents the *powerset* of X, that is, the set of all subsets of X.

II.1.2 Formal language theory

Definition 5: An *alphabet* Q is a set of symbols. A *string* over an alphabet Q is a finite list of symbols composed of members of Q. The empty string is denoted ε. The set of all finite strings over Q is denoted Q*. The set of all non-empty strings over Q, Q*-{ε}, is denoted Q^+.

For example, let Q = {a,e,i,o,u}. Then:

a iou eeee ε

are all strings over Q.

Definition 6: A *language* over an alphabet Q is a subset of Q*.

Definition 7: The *concatenation* of strings x and y over an alphabet Q, denoted x•y, is the string that results from juxtaposing x with y. Concatenation may be extended over two sets of strings, X and Y, over an alphabet Q, as X•Y = { x•y | x ∈ X, y ∈ Y }.

Definition 8: A *grammar* is a four-tuple ‹V,T,P,S› where V is an alphabet of *non-terminals*, T is an alphabet of *terminals*, P ⊆ (V∪T)*•V•(V∪T)*×(V∪T)* is a finite set of *productions* and S is the *start symbol*. A production ‹X Y Z,a b c d› is written:

X Y Z ::= a b c d

In this case, "X Y Z" is said to be the *left hand side (lhs)* of the production and "a b c d" is said to be the *right hand side (rhs)* of the production. This type of grammar is called a *Chomsky Type 0* grammar or *unrestricted grammar*, referring to the lack of restriction on the lhs and rhs of the productions (see Definitions 11 and 12 for examples of restricted grammars).

Definition 9: Let $G = \langle V,T,P,S \rangle$ be a grammar. The *directly-derives* relation on $(V \cup T)^*$, written $w \models w'$, holds between w and w′ when there exist w_1, v, w_2, $\in (V \cup T)^*$, $v ::= v' \in P$ such that $w = w_1 v w_2$ and $w' = w_1 v' w_2$. The *derives* relation on $(V \cup T)^*$, written as $w \models^* w'$, is the reflexive, transitive closure of $\models$.

Definition 10: Let $G = \langle V,T,P,S \rangle$ be a grammar. The language *generated* by G, denoted $\mathcal{L}(G)$, is defined as $\{ x \mid S \models^* x, x \in T^* \}$, that is, the set of all *terminal strings* that may be derived from the start symbol.

Definition 11: A *context-free grammar* is a grammar $G = \langle V,T,P,S \rangle$ where $P \subseteq V \times (V \cup T)^*$. That is, the lhs of each production consists of a single non-terminal.

Definition 12: A *regular grammar* is a grammar $G = \langle V,T,P,S \rangle$ where $P \subseteq V \times (T^* \bullet (V \cup \{\varepsilon\}))$. That is, a context grammar whose productions are further constrained to limit the rhs to contain a single non-terminal which, if present, must be the rightmost symbol in the production.

Notation: Let $G=\langle V,T,P,S \rangle$ be a regular or context-free grammar. Each production $p \in P$ is of the form $S_0 ::= S_1 S_2 \ldots S_{n_p}$, where n_p is the number of symbols on the right hand side of p. The occurrences of symbols in a production are numbered left to right, $p[0] = S_0$ being the lhs nonterminal, $p[i]=S_i$, $1 < i \leq n_p$ being the rhs symbols. Note that $p[0] \in V$.

II.1.3 Lattices

Definition 13: A *semilattice* L consists of a set of elements E_L, and a binary operator $\wedge_L$, called the *meet* operator, with the following properties:

1. $\forall x \in E_L : x \wedge_L x = x$ (idempotence)
2. $\forall x,y \in E_L : x \wedge_L y = y \wedge_L x$ (commutativity)
3. $\forall x,y,z \in E_L : x \wedge_L (y \wedge_L z) = (x \wedge_L y) \wedge_L z$ (associativity)

When clear from context, the name of a semilattice is used to refer to its set of elements E_L. Often a lattice will be denoted as a pair consisting of the set of elements and the meet operator. The number of elements in a semilattice L is denoted by |L|.

Definition 14: Given a semilattice L with elements $x,y \in L$, define

1. $x \leq_L y$ iff $x \wedge_L y = y$
2. $x <_L y$ iff $x \wedge_L y = y$ and $x \neq y$

Similarly, $x \geq_L y$ means $y \leq_L x$ and $x >_L y$ means $y <_L x$. Note that $\leq_L$ is a partial order (antisymmetric, reflexive transitive relation on the elements of L).

Definition 15: A *chain* in a semilattice L is a sequence of elements $x_0,x_1,\ldots,x_n$ such that $x_i <_L x_{i+1}$, for $0 \leq i < n$; the *length* of a chain is one less than number of elements in the chain.

Definition 16: A semilattice L is said to be *bounded* if for each $x \in L$ there exists a constant b_x such that any chain beginning with x is of at most length b_x. The *height* of a bounded lattice L is the length of the longest chain in L, and is denoted *height*(L).

Definition 17: A *monotonic function* $f : L \rightarrow L$ is a function with the property that $\forall x_1, x_2 \in L$ if $x_1 \leq_L x_2$ then $f(x_1) \leq_L f(x_2)$.

The condition in Definition 17 is equivalent to following [44]:

$$\forall x_1, x_2 \in L. f(x_1 \wedge x_2) \leq_L f(x_1) \wedge f(x_2).$$

Definition 18: A *distributive function* $f : L \rightarrow L$ is a function with the property that $\forall x_1, x_2 \in L.\ f(x_1) \wedge f(x_2) = f(x_1 \wedge x_2)$.

Definition 19: Given a lattice L, a set of functions on L is said to be a *monotonic function space associated with* L if the following conditions are met [44]:

1. Each $f \in F$ is monotonic.
2. There exists an *identity function* $I \in F$, such that $\forall x \in L. I(x)=x$.
3. F is closed under composition, i.e., $f,g \in F$ implies $f \circ g \in F$.
4. L is equal to the closure of $\{\perp_L\}$ under the meet and application functions of F.

II.1.4 Attribute grammars

In his seminal paper [54], Knuth introduces the *attribute grammar* for associating semantics with a context free language. Attribute grammars are used as the primary semantic description mechanism in many compiler generator systems (see Section II.4 below) and are consequently assumed as a given throughout this dissertation. For a formal definition of attribute grammars, the reader is directed to Knuth's original paper, or an introduction to attribute grammars by Alblas [5]. Deransart et al. provide an excellent overview of the state of attribute grammar research up to 1988, including a list of compiler generators based on the formalism [19].

Definition 20: Informally, an *attribute grammar* consists of a grammar, a set of *attributes* and a set of *attribution rules*. Attributes are associated with the nodes in a derivation tree. A given attribute is *synthesized* or *inherited* and is associated with a non-terminal or terminal in the grammar. Synthesized attributes of a given node are defined in terms of the attributes of the node's children. Inherited attributes, in contrast, are defined in terms of the attributes of a node's parent and siblings. *Semantic rules* are associated with each production which:

1. Define values for all synthesized attributes of the lhs non-terminal.
2. Define values for all inherited attributes of all symbols on the rhs.

Example

Figure 4 gives an example of an attribute grammar to compute the set of identifiers occurring in an expression. The non-terminal `E` has a single synthesized attribute, `ids`, used to collect the set of identifiers in the subexpression rooted at an instance of `E`. Multiple occurrences of a non-terminal in a given production are numbered, left to right starting from zero, so that they may be distinguished in the semantic rules (e.g., E_0, E_1 and E_2 in the first production's semantic rule). The terminal `Id` is assumed to have an attribute yielding a value representing the actual identifer it represents (in practice, this would not be the value assigned by the lexical analyzer, but the value assigned during name analy-

sis). Note that the grammar is ambiguous: this is an *abstract syntax*; it is assumed that parsing details such as operator precedence have been abstracted away yielding a tree whose structure reflects the correct association. The figure also shows a sample expression and corresponding derivation tree "decorated" with the attribute values specified by the semantic rules.

The power of attribute grammars

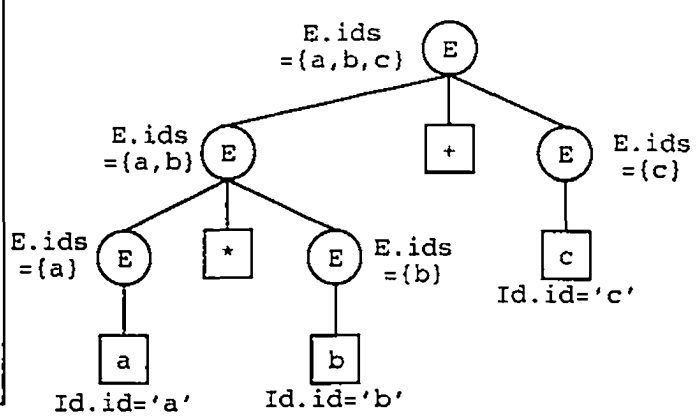
An important aspect of attribute grammars is that arbitrary functions may appear in semantic equations. One result of this generality is that inherited attributes are not essential to the formalism. Synthesized attributes may be used to construct a clone of the derivation tree into the root node. A semantic function at the root then has all the information necessary to compute anything that attribution could have. In some sense, this technique sidesteps the attribute grammar paradigm, providing a useful "back door."

Figure 4 Example attribute grammar to compute the set of identifiers in an expression

Attribute grammar

```
E ::= E '+' E
  E0.ids = E1.ids ∪ E2.ids
E ::= E '*' E
  E0.ids = E1.ids ∪ E2.ids
E ::= '-' E
  E0.ids = E1.ids
E ::= Id
  E.ids = { Id.id }
```

Decorated derivation tree for expression: a * b + c



Circularity and attribute grammars

Definition 20 admits a broad range of possible attribute grammar classes. One class in particular allows attribute grammars having circular attribute equations for at least one derivation tree. That is, a value is a function of itself, which may not be well-defined.

The original solution to this potential problem, proposed by Knuth, was to add a second notion of attribute grammars, the *well-defined attribute grammars (WAGs)*. WAGs are attribute grammars for which no derivation tree can yield cyclic attribute equations. Most attribute grammar based compiler generation systems use a subclass of the well-defined attribute grammars, and thus do not permit circularity in the attribute equations for any derivation tree.

Another possibility is to allow circularity and use a fixed point semantics of the resulting attribute equations [4]. In this case, the value domains of the attributes and semantic functions must be constrained in some way to ensure the equations have a solution. A typical constraint is to limit the attribute domains to finite height lattices and semantic functions to be monotonic. Combined, these two constraints permit solution of the resulting system by iteration.

Lastly, the circularity may be encapsulated and solved in one or more of the semantic functions. Sagiv et al. take this approach to localize transitive circularities (that is, across productions) into direct circularities (within a single production). The local circularities are then replaced with a functional attribute which captures the fixed point [75]. Conceptually, a similar approach is taken in this dissertation. A particular representation of the program is computed into an attribute at the root of the derivation tree. A function is applied to this value which constructs a system of equations and finds a fixed point. Attribution rules distribute the resulting solution throughout the tree.

II.1.5 Denotational semantics

An example in Section IV.7 requires the following definitions from denotational semantics. Stoy [82] and Tennent [85] provide introductions to the subject. The notation is from [66].

Definition 21: A *partially ordered set* $\langle S, \leq \rangle$ is a set S with a *partial order* $\leq$ (antisymmetric, reflexive, transitive relation).

In subsequent definitions $\langle S, \leq \rangle$ is a partially ordered set.

Definition 22: A subset $S' \subseteq S$ has a (necessarily unique) *least upper bound* LUB(S') if $\forall s \in S \,.\, (\text{LUB}(S') \leq s \textit{ iff } \forall s' \in S' \,.\, s' \leq s)$

Definition 23: A non-empty subset $S' \subseteq S$ is a *chain* if S' is countable and $\forall s_1, s_2 \in S \,.\, (s_1 \leq s_2 \textit{ or } s_2 \leq s_1)$.

Definition 24: An element $s \in S$ is *least* if $\forall s' \in S . s \leq s'$.

Definition 25: $\langle S, \leq \rangle$ is a *cpo*, or *domain*, if it has a least element ($\perp$) and any chain has a least upper bound.

Definition 26: A domain is *flat* if any chain contains at most two elements.

Definition 27: A domain is of *finite height* if any chain is finite.

Definition 28: N is the flat domain of natural numbers.

Definition 29: T is the flat domain of truth values.

Definition 30: Given domains $S_1, S_2, \ldots, S_n$, the *separated sum* $S = S_1 + S_2 + \ldots + S_n$ may be constructed with:

1. a new least element, $\perp_S$.
2. *injection functions* "in S_i," (that is, given a member of $s_i \in S_i$, "s_i in S_i" is the corresponding element in S).
3. *enquiry functions* **E** S_i (that is, given a member s of S, **E** S_i yields true *iff* s belongs to S_i)
4. *projection functions* | S_i (that is, given a member s of S, | S_i yields the corresponding member of S_i if s is a member of S_i, otherwise it yields $\perp_{S_i}$.

Definition 31: Given domains $S_1, S_2, \ldots, S_n$, the *Cartesian product* $S = S_1 \times S_2 \times \ldots \times S_n$ may be constructed with *selection functions* $\downarrow i$ (that is, yield the i'th component).

Definition 32: S* is the domain of *lists*: $\{\langle\rangle\} + S + (S \times S) + \ldots$ with functions:

1. *length*, written #.
2. *remove first* i elements, written †i.
3. *concatenate*, written §.

All functions are assumed to be total. For partially ordered sets S and S′, the set of total functions from S to S′ is denoted S-t>S′.

Definition 33: A function $f \in$ S-t>S′ is *continuous* if $f(LUB(S'')) = LUB(\{ f(s) \mid s \in S' \})$ holds for any chain $S'' \subseteq S$ whose least upper bound exists.

The set of continuous functions from partial order S to partial order S′ is denoted by S-c>S′. Both S-t>S′ and s-c>S′ are partially ordered by $f_1 \leq f_2$ *iff* $\forall s \in S' . f_1(s) \leq f_2(s)$. The same holds if S′ is a domain.

Definition 34: An element $s \in S$ is a *fixed point* of $f \in$ S-t>S if f(s) = s. When S is partially ordered, s is the *least fixed point* provided it is a fixed point and s′ = f(s′) *implies* $s \leq s'$. If S is a domain and $f \in$ S-c>S then the least fixed point always exists and is given by $FIX(f) = LUB(\{ f^n(\perp) \mid n \geq 0 \})$. (Note: $f^0 = \lambda s.s$ and $f^i = f \circ f^{i-1}$, $i \geq 1$.)

Notation: $\geq\geq$ is the continuous version of the usual greater-than-or-equal-to relation on the natural numbers (that is, $x \geq\geq y = \perp$ if $x=\perp$ or $y=\perp$). The conditional $b \rightarrow s_1, s_2$ is $\perp$, s_1 or s_2 when $b \in T$ is $\perp$, true or false respectively.

II.1.6 Miscellaneous

Definition 35: The set of all partial functions from a set X to a set Y is written $X \rightharpoonup Y$.

II.2 Definition of a compiler

For the purposes of this dissertation, a *compiler* is a program that accepts a *source program* in some *source language* as input and generates a *target program* in some *target language* as output. At the highest level a compiler should:

1. Ensure that the input submitted is a *valid* source program.
2. Generate a target program whose *meaning* in the target language is equivalent to the meaning of the source program in the source language (assuming the source program is valid).

Ideally, the compiler should also:

3. Generate an *efficient* target program, where efficiency is typically some measure of resource use, such as memory or cpu time.

The first of these three requirements is the best-understood and many tools to automate the construction of the required tasks exist. Methods for obtaining efficient target programs are not as well-understood, and automating their implementation less so.

II.3 Static analysis

Generating an efficient target program typically involves building an intermediate representation of the program being compiled and then applying *optimizing transformations*[1] to the intermediate structure. While some transformations require little knowledge about the program being compiled, the most effective may only be applied when a variety of constraints, involving information about possible executions, are met. That is, the program must be subjected to compile-time analysis, called *static analysis*, to infer information needed to ascertain the applicability of a given optimization [50].

II.3.1 Example: live variable analysis

In order to make effective use of registers, it is very useful to know the *lifetimes* of the variables in a program. A variable is said to be *live* when the value it contains could be used again before the variable is assigned a new value. Clearly, variables whose lifetimes do not overlap can share a register. Similarly, the value in a register does not need to be stored into memory if the variable it represents is not live.

More concretely, consider the program in Figure 5, where the numbers to the left of the code indicate program points (see below). There are three variables in the program, `i`, `f` and `out`. At the end of the program, there are no live variables. The assignment at point 7 uses `f`, making `f` live at point 7. The other values are shown in the column labelled "Live variables" in the figure.

1. Such transformations are usually not optimal in a provable sense. Rather, they are expected to improve performance.

Figure 5 Sample Pascal program

Point	Live variables	Pascal program
		`PROGRAM test;`
		`VAR  i:INTEGER;`
		`  f:INTEGER;`
		`  out:INTEGER;`
		`BEGIN`
1	∅	`  i := 3;`
2	{i}	`  f := 1;`
3	{f,i}	`  WHILE i > 1 DO BEGIN`
4	{f,i}	`  f := f * i;`
5	{f,i}	`  i := i - 1`
6	{f,i}	`  END;`
7	{f}	`  out := f`
8	∅	`END.`

II.3.2 Data flow analysis frameworks

There are a number of details implicit in the preceding example that have been formalized into the notion of *data flow analysis frameworks* [44, 51]. These frameworks typically consist of:

1. A *model* of the program defining the *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. In the example above, this *information space* has abstract states consisting of sets of live variables.

3. A set of *flow equations* relating the abstract state at one point with the points that may immediately precede or follow it at run-time.

Usually the model induces the flow equations, either directly or indirectly. In some cases, the structure of the model may influence the information space [78]. Marlowe and Ryder provide an excellent survey of the these frameworks in [59]. The following sections detail the various aspects of the above components relevant to this dissertation.

II.3.3 Program models for flow analysis

Data flow analysis may be performed on any one of a number of levels [59]. Initial efforts concentrated on *intraprocedural analysis*[2], that is, analyzing the flow of information *within* single procedures [3, 18, 29, 44, 49, 72]. At a higher level, *interprocedural analysis* takes flow *between* procedures into account. In the absence of interprocedural flow information, worst case assumptions must be made at *call sites* (places in the program where a procedure is called) to ensure that all optimizations made are safe.

It is usual in flow analysis to make two assumptions relating to conditionals at the outset. Firstly, when control reaches a conditional branch (e.g., if/then/else), it is assumed that either branch may be taken at run-time. Secondly, it is assumed that the choice made at a conditional branch is independent of any previous choices made. These assumptions are necessary because determining the outcome of an arbitrary conditional is undecidable in general. These assumptions lead naturally to the use of graphs to represent intraprocedural control flow, as outlined in the next section.

2. Also called *global analysis* in the literature.

Graph models of control flow

While there are many program representations used in data flow analysis, the vast majority are based on some form of graph. Intraprocedural control flow is frequently modeled by a *control flow graph* [3, 18, 29, 44, 51]. Although many types of control flow graphs have been used, they may be classified into two basic variants:

1. *Node-based*: in a node-based control flow graph, the nodes of the control flow graph represent straightline pieces of code and an arc from node A to node B indicates the possibility that code represented by node B may be executed immediately after the code represented by node A. Program points are typically assumed to occur at the entry and exit of each node. The nodes are often assumed to be *basic blocks*, that is, maximal fragments of *straightline* (single entry, single exit) code which minimizes the number of points where flow information is computed.
2. *Arc-based*: alternatively, the code may be associated with the arcs in the graph. In this case, the nodes are considered the program points. The advantage of this structure is that at a split in control flow, the sense of the conditional is encoded in the arc itself; it is not necessary to include such information in the abstract state.

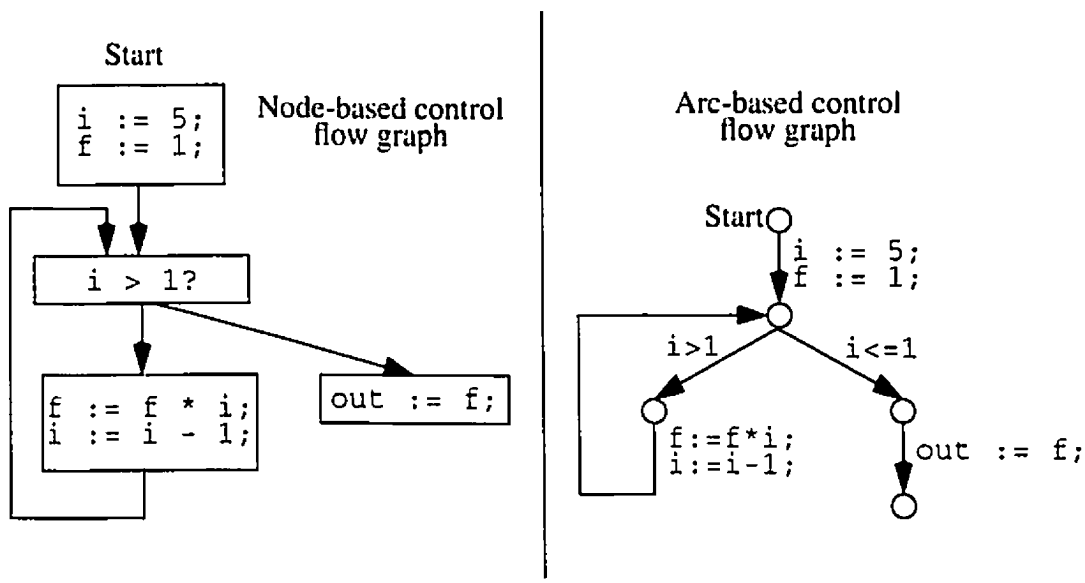
The control flow graph has been successful in the intraprocedural context because it accurately models how control flows within a procedure. Each path in the graph represents a possible execution under the usual assumptions. The arc-based and node-based control flow graphs for the example program are shown in Figure 6.

Interprocedural control flow has also been modeled by graphs. In some instances a *call (multi-)graph* [16, 74] is used in which each node represents a procedure and each arc represents a call site from a *caller* (the procedure containing the call site) to a *callee* (the procedure being called). In this scheme, the call graph and control flow graphs of the individual procedures are used to model the control flow of the program being analyzed. A related approach is to combine the two representations directly into an *interprocedural flow graph* using special "call site nodes," and special *call* and *return* arcs [78]. Recently,

a group at McGill University has introduced the *invocation* graph [32, 81], which is an expanded form of call graph that explicitly distinguishes each possible (non-recursive) call path.

A graph representation is particularly useful for performing transformations on low-level sequential code, as it closely represents the final form of the program. Fragments may be easily moved from one node to another, and subsequent analyses performed on the new structure [3].

Figure 6 Node- and arc-based control flow graphs of program in Figure 5



II.3.4 Information spaces

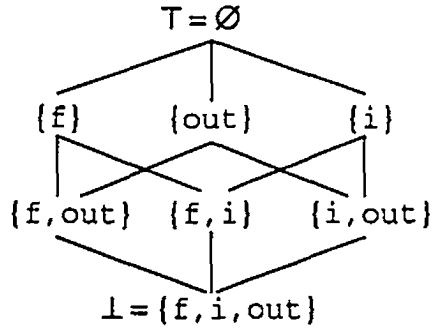
An *information space* is an abstraction of possible program states. Each *abstract state* in an information space represents a set of *concrete states*, that is, program states that may occur at run-time. A data flow analysis associates an abstract state with each program point in the program.

The most common structure for an information space is a bounded semilattice [18, 29, 44, 51, 78, 97]. Each element of the lattice represents an abstract state, and the meet operator is used to summarize two abstract states when control flow merges. The bottom element of the lattice represents worst case information, with respect to the use of the information. If the lattice has no natural top element, an artificial top element meaning "no information known yet" is frequently added to simplify the implementation of the data flow analysis algorithm. *Note that these lattices are "upside-down" with respect to the use of lattices in denotational semantics where the* top element *means* overspecified *and the* bottom element *means* underspecified.

Example

In an intraprocedural live variable analysis the information space is the powerset of the set of variables in the procedure or program. Let Var be the set of variables in a program. Then the live variable lattice for that program is $\langle 2^{Var}, \cup \rangle$. Figure 7 shows a Hasse diagram of the live variable lattice for the example program in Figure 5. The top element is the empty set and the bottom element is the set of all variables.

Figure 7 Live variable lattice for program shown in Figure 5



Monotone data flow analysis frameworks [44]

Kam and Ullman formalize the notion of an information space as a *monotone data flow analysis framework.*

Definition 36: A *monotone data flow analysis framework (MDFAF)* is a triple $D=\langle L,\wedge_L,F\rangle$, where:

1. L is a bounded semilattice with meet $\wedge_L$ (in this treatment it is assumed to have a top element $\top_L$ and a bottom element $\bot_L$).
2. F is a monotone function space associated with L.

II.3.5 Flow equations

The actual analysis is represented by a set of *flow equations* representing the *abstract semantics* of the program. Program points in the control flow model correspond to variables in the flow equations. For a node-based control flow graph, there are two program points for each node, typically labelled IN and OUT, representing the flow information known at entry and exit of the node, respectively. A function on abstract states is associated with each label in the graph which computes the abstract effect of executing the label's code. Arcs in the control flow model represent dependencies among the program states and, thus, the structure of the flow equations.

Backward intraprocedural analysis

The equations themselves depend on the *direction* of information flow. In the original formulations of flow analysis frameworks, information was assumed to flow either *forward* or *backward*. Consider the live variables problem: at the end of the program the set of live variables is whatever the operating system might use (frequently assumed to be the empty set). For the example program, it is assumed that nothing is live at the end of the program

(point 8). Given this fact, the set of live variables at point 7 is the set of live variables live at point 8 minus the element `out` (a value is assigned to this variable), plus the element `f` (this variable is used in the assignment). That is, the abstract information flows "backward" from the end of the program towards the beginning.

For a node-based control flow graph, deriving a set of flow equations representing this kind of backward flow is straightforward. The IN point of the node N (denoted N.in) is a monotonic function of its OUT point (denoted N.out):

$$N.in = f(N.out)$$

The OUT point of node N is the meet of all the IN values if its successors:

$$N.out = \bigwedge_{M \in succ(N)} M.in$$

Flow equations for an arc-based control flow graph are similar.

Forward intraprocedural analysis

In many data flow problems, information flows forward from the start of the program. An example is *constant propagation* [3, 29, 44, 91] which tracks the values of variables as long as they are known to be constant, and uses these values to facilitate *constant folding*. In this case the values of all variables are typically assumed to be "unknown" or zero (depending on the source language) at the start of the program. Information flows forward from one point to the next. The flow equations for such a problem mirror the backwards equations above; for a node N in a node-based control flow graph:

$$N.out = f(N.in)$$

where f is a monotonic function, and

$$N.in = \bigwedge_{M \in pred(N)} M.out$$

Bidirectional intraprocedural analysis

In some analyses information may flow in both directions, that is *bidirectionally.* An example of such a problem is Morel and Renvoise's *partial redundancy elimination* [62]. Flow equations typical of this type of problem define IN values in terms of OUT values and vice versa:

$$N.in = \bigwedge_{M \in pred(N)} f(M.out)$$

$$N.out = \bigwedge_{M \in succ(N)} g(M.in)$$

II.3.6 Interprocedural analysis

The goal of interprocedural analysis is to make a more accurate determination of the effect of a procedure call than provided by worst case assumptions. In a forward analysis, for example, we need an approximation of the abstract state after the call, given the state that holds before the call. Sharir and Pnueli devised two frameworks for such analyses, called the *functional approach* and the *call-string approach* [78]. These two approaches addresss forward flow analysis in the presence of procedure calls. For both methods node-based graphs are used to represent the program being analyzed.

The essence of the functional approach is to compute *phi functions.* A phi function is associated with node in the control flow graph and represents the execution of the program from the beginning of the procedure in which the node occurs up to the exit of the

node. Using a finite state lattice, the values of the phi functions may be computed using an iterative algorithm.

In the call string approach, the idea is to treat calls and returns as normal jumps, encoding the "propagation history" in a *call string*. By tagging the flow information with its call history, it is possible to distinguish separate paths when processing the return from a procedure. In the presence of recursion, there is an unbounded number of possible call strings and some form of approximation must be introduced. Unlike the functional approach, this approach does not require a finite lattice to ensure termination: finite height is sufficient, but not necessary.

II.4 Compiler construction and its automation

Compiler construction differs from most areas of software development in that many aspects of compilation have matured into well-understood and accepted techniques [45]. The overall structure of a compiler is typically composed of the following phases [3, 90]: scanning, parsing, tree construction, semantic analysis (e.g., name analysis and type checking), optimization and code generation. For several of these phases there exist concise abstractions which allow the tasks of the phase to be specified effectively. It is precisely these abstractions that permit the automatic construction of a module implementing a given phase. Table 1 lists a number of phases along with corresponding abstractions and tools developed for compiler generation technology.

Table 1 Abstractions used to describe various phases of compilation

Phase	Abstraction	Example tools/techniques
Scanning	Regular expressions	LEX [58], GLA [33], Mkscan [35]
Parsing	LALR(1) grammars LL(k) grammars	YACC [40], Ilalr [36] ANTLR [67]
Semantic analysis	Attribute grammars	GAG [48], LIGA [46]
Optimization	Tree pattern matching	DORA [23], MUG2 [96], GOSpeL [93]
Code generation	Grammars Tree rewriting Denotational semantics	Graham-Glanville [25] BURS [69, 70] MESS [57]

While there are many factors affecting the success of a compiler generation tool, it seems fair to say that if the abstraction is inadequate for the task being described, the tool is less likely to be used. One potential problem is lack of generality. The GLA scanner generator, for example, assumes that whitespace is not significant in the source language [33]. Thus, it cannot be used to implement a scanner for a language that uses the "off-side" rule for scoping, such as Haskell [37].

Another possible pitfall is that the abstraction does not admit straightforward specification of the required task. For example, there is continued debate over the use of LL(k) vs. LR(k) for parser generators: the latter have greater recognition strength, while the former permit greater semantic control over parsing [68]. This applies to Prolog's *definite clause grammars* [14], for example, which require the grammar to be in LL(1) form. Similarly, it is possible in general to specify scanning tasks using parsers, at the expense of a less intuitive description (a regular grammar rather than a regular expression) and typically increased run-time cost when compared with a scanner generated by a scanner generation tool.

As mentioned in Section II.1.4, the standard attribute grammar formalism permits neither remote access to attributes nor circularity in the attribute equations. Implementations of attribute grammars systems typically have further restrictions regarding attribute dependencies, such as "orderedness" [47]. While these constraints permit the construction of efficient attribute evaluators and evaluator generators, they occasionally make certain tasks difficult. A strictly left-to-right evaluation scheme makes it somewhat cumbersome to perform type checking, for example. Doing a flow analysis directly in an attribute grammar, even if circularity was permitted, would be tedious if the source language permitted non-local control flow such as non-restricted goto statements.

Code generation is less well-understood, resulting in a variety of mechanisms for abstracting the code generation process. An early technique, known as Graham-Glanville code generation, specifies the code generation task using a form of context free grammar and implements the code generator using a form of LR parsing [25]. BURS, which stands for *Bottom-Up Re-write System*, also uses a grammar based specification, but the algorithm is based on a tree pattern matcher implemented using dynamic programming [69, 70].

Even optimizer generators exist. An early effort is the MUG2 system [95, 96], while more recent work includes Sharlit [86], GOSpeL/GENesis [93], and DORA [23]. The general idea behind all these systems is to allow the user to supply transformation/predicate pairs where the transformation typically consists of a pattern and possible replacement whose admissability is determined by the corresponding predicate.

Finally, some systems are designed to translate (various forms of) denotational semantic specifications to compilers. Perhaps the best known is Peter Lee's MESS system which allows a language to be specified in terms of "high level semantics" and "microsemantics" and translates them into a working compiler [57]. Unfortunately, his language

descriptions seem marginally easier to read and understand than a denotational semantic description.

II.5 Techniques and tools for static analysis

As in the case of code generation, there have been a number of techniques and tools devised for specifying and implementing static analyses. Given the prevalence of attribute grammars in compiler generator tools, there have been many efforts to contort the attribute grammar paradigm to solve various classes of flow analysis problems. Other efforts have resulted in tools which are directed primarily at the data flow aspects, generally assuming a particular intermediate representation amenable to data flow analysis.

II.5.1 Static analysis techniques

There are three related techniques used to compute static information about a program for the purposes of optimization. The data flow analysis frameworks described in Section II.3 is known traditionally as *data flow analysis* [3, 29, 86]. A similar approach is *abstract interpretation* [2, 18]. *Partial evaluation*, another related technique, can be viewed as a general form of the constant propagation flow analysis problem.

Abstract interpretation

Abstract interpretation, due to Cousot and Cousot, is based on the idea of interpreting the source program in an *abstract domain*, rather than a *concrete domain*, as is usual in denotational semantics [18]. The goal is to characterize the set of concrete states possible at each program point with an abstract state. A requirement is that the abstract state computed for a given point must represent any possible state the program could be in when

control reaches that point. More formally, let C be the lattice of (collections of) concrete program states and A be the lattice of abstract states. Correctness is established by finding two functions:

1. $\alpha : C \rightarrow A$ – the *abstraction* function.
2. $\gamma : A \rightarrow C$ – the *concretization* function.

such that for all abstract states $a \in A$ we have $\alpha(\gamma(a)) = a$ and for all concrete states $c \in C$ $\gamma(\alpha(c)) \leq c$ ($\langle\alpha,\gamma\rangle$ is an *adjoined pair* of functions[3]).

Unlike the case of data flow analysis in the style of Kam and Ullman [44], the abstract interpretation framework explicitly allows the lattice of abstract states A to be of infinite height. To ensure termination of analyses in the presence of such lattices, a *widening* operator, ∇_A, with the following properties is introduced:

1. $\forall a,a' \in A: (a \nabla_A a') \leq_A (a \wedge_A a')$.
2. Every infinite sequence $s_0, s_1,\ldots,s_n,\ldots$ of the form $s_0 = a_0, s_1 = a_1 \nabla_A s_0,\ldots$, $s_n = a_n \nabla_A s_{n-1},\ldots$ (where $a_0,a_1,\ldots,a_n,\ldots$ are arbitrary abstract states) is not strictly decreasing.

The idea is to insert an occurrence of this operator into the flow equations for each loop in the program. That is, for each loop, there must be at least one flow equation between two successive points of the form:

$$x = x \nabla_A f(x)$$

Because ∇_A is not strictly decreasing over infinite sequences of arbitrary values, this guarantees a fixed point.

3. If the first constraint is weakened to $\alpha(\gamma(a)) \leq a$, $\langle\alpha,\gamma\rangle$ is *semi-adjoined*.

Data flow analysis in a denotational framework

Nielson introduced a framework for performing forward intraprocedural analysis based on denotational semantics [65]. He defines the notion of a *collecting semantics* which associates possible program states with a set of program points. The collecting semantics is based on a *store semantics*, that is, a denotational semantics that maintains a stack [82]. Each AST node is decorated with two program points, "entry" and "exit," uniquely identified by the position of the node in the tree; the semantics then "collects" the set of possible concrete states that may occur at these points. A data flow analysis consists of defining a pair of *semi-adjoined* functions which define the relationship between abstract states and concrete states. As in abstract interpretation, these functions represent the constraint that the abstract state for computed a given point must subsume the set of possible states for that point, as defined by the collecting semantics.

In a subsequent paper [66], Nielson considers the use of his framework to perform program transformations. Here he adds a method for performing intraprocedural backward flow analysis which relies on a continuation passing style semantics (see Section II.6.1 below). The idea is to associate an *abstract continuation* with each program point which summarizes the desired information.

Unfortunately, Nielson's ideas do not lend themselves to the construction of a compiler generation tool. Two features are needed to permit easy prototyping and automatic generation:

1. a user oriented specification mechanism for representing intra- and interprocedural control flow uniformly; and
2. a corresponding interprocedural interpretation for backwards, forwards and bidirectional problems.

Partial evaluation

A related technique that has received considerable attention is *partial evaluation* [26]. In this case, the idea is to *specialize* a program with respect to some of its input yielding a *residual* program. Running the residual program on the remaining input is intended to be equivalent to running the original program on all the input. The technique is useful because knowing some of the input may permit optimizations to be made which increase the efficiency of the residual program, with respect to the original. Constant propagation may be viewed as a form of partial evaluation in which no input is available.

One approach to partial evalution, called *polyvariant mixed computation*, is similar to data flow analysis. Specifically, the idea is to compute sets of states that a program may be in at a given point, assuming a given set of known inputs. A new program is generated which makes use of these states. Unfortunately, partial evaluation does have a few problems, including the possibilities of non-termination and of generating residual programs that do not preserve the termination behaviour of the original program [26].

Abstract interpretation and data flow analysis may be viewed as forms of polyvariant mixed computation in which the abstract program states and flow equations ensure termination. That is, in the standard polyvariant technique, the state lattice is of infinite height and there is no widening operator to prevent non-termination. Also, while abstract interpretation and data flow analysis are concerned with characterizing sets of states, partial evaluation has the added goal of providing an executable residual program.

II.5.2 Existing static analysis tools

Because of the importance of static analysis, a variety of flow analyzer generators have been developed. These tools vary widely, from automating the generation of analyses in a

given compiler [81, 98, 86] to generating analyses within a given compiler generator [95], along with approaches to performing static analysis directly in an attribute grammar [73]. Even Kildall mentions the implementation of a tool for creating classical intraprocedural analyses in his seminal paper [51]. These efforts, along with other relevant projects are discussed in the following sections.

Attribute grammar based systems

A number of attribute grammar techniques have been proposed, including:

1. Babich and Jazayeri's *method of attributes* [9, 10];
2. Farrow's *finitely recursive* attribute grammars [24];
3. Rosendahl's lazy fixpoint evaluation for attribute grammars [73].

Writing attribute grammar specifications for data flow equations, however, is not trivial when the source language permits unstructured control flow or procedure calls. In fact, techniques for performing interprocedural analysis in an attribute grammar have yet to appear.

Graph transformations

In his Ph.D. dissertation [28], Grundman develops an intraprocedural flow analysis tool, based on *graph transformations*, intended to support Farnum's DORA optimizer generator [23] (see below). The general idea is to represent all aspects of a flow analysis as graphs and to perform the analysis, including the computation of the fixed point, as a set of graph transformations based on pattern matching. While Grundman notes the independence of his technique from the DORA environment, it is not clear how easy it would be to extend his graph-based formalisms to interprocedural analysis. And while the conceptual simplicity of using graphs for the entire analysis is intellectually appealing, it seems likely that

the user of a compiler generation tool would not welcome the necessity of contorting well-understood lattices into a graph-based representation.

DORA

Farnum's DORA system is the result of investigating the use of *pattern matching languages* to support the prototyping of compiler optimizers [23]. DORA uses a tree-based intermediate representation, which he calls DILS, and supports intraprocedural data flow analysis using SAL, DORA's tree attribution language. The techniques introduced are of particular interest because while DILS itself explicitly admits continuations (see Section II.6.1 below), Farnum constructs a control flow graph to perform the data flow analysis. Interestingly, Farnum notes that much of the code required for a flow analysis is associated with constructing the control flow graph.

McTAG

An on-going project at McGill University, the *McGill Compiler Architecture Testbed (McCAT)* is addressing "the interaction between compiler techniques and advanced architectural features" [31]. The main thrust of the project is the development of a C compiler capable of supporting a broad range of optimizations. Sridharan has devised a flow analysis framework, called McTAG, to support both intra- and interprocedural analysis [81]. It is based on an intermediate AST form, dubbed SIMPLE, which fully disambiguates the many fuzzy aspects of C and from which all goto statements have been eliminated [22]. Interprocedural control flow is modelled using an *invocation graph*, an interesting variation of the standard call graph in which call chains have been expanded to the first point of recursion. Flow analyses are specified using pattern/action pairs that compute the abstract semantics of each SIMPLE construct. Interprocedural analysis is handled by the call node action. It involves the specification of a pair of functions, map and unmap, that map the

caller's environment into the callee's environment and back, respectively. Call site information for all calls represented by a given invocation graph node is merged. Not surprisingly, this framework is quite well-suited to the analysis of C programs. Extending it to become a more general tool would present some challenges, however: either the SIMPLE language would require extension to handle constructs such as nested procedures explicitly, or the front-end of the compiler would have to translate to the existing SIMPLE constructs, possibly losing information present in the original source program.

MUG2

One of the earliest systems for specifying flow analyses and optimizing transformations is MUG2, designed at the Technical University of Munich [96]. A slightly modified attribute grammar formalism is used to specify intraprocedural flow analyses. The formalism, however, is quite limited, permitting only structured control flow (a complicated technique involving abstract state stacks is presented for handling multi-level exits; general goto statements, however, are not admitted). Also, it is necessary to structure the analysis as a sequence of left to right or right to left AST traversals; general circularity is not allowed.

Sharlit

Sharlit is a tool for building intraprocedural optimizers [86]. A primary component allows the specification and implementation of data flow analyzers, based on relatively low level control flow graphs. The interface is based on control flow graphs, flow values, flow functions, action routines and path simplification rules. Of these, the first three correspond to the usual monotone data flow analysis frameworks. Action routines use the flow analysis results to perform optimizations. Path simplification expressions, based on Tarjan's path problems [83, 84], permit a variety of solution procedures to be specified, from straightforward iteration to interval analysis. The framework cannot be directly applied to the

problem of constructing a flow analyzer generator, however, because it assumes that the control flow model has already been constructed. Furthermore, Sharlit does not address interprocedural analysis.

SPARE

Venkatesh describes a system called SPARE for performing abstract interpretation [88, 89]. This system is based on a compositional denotational semantics, requiring a denotational description of the language being compiled. Because SPARE depends on compositionality, it cannot handle arbitrary control flow constructs, making the system unusable for many languages.

Z1

Yi's Z1 is a system for specifying abstract interpreters [97, 98]. One of the main thrusts of his work is the development of abstract domains, based on finite modular atomic lattices. Such lattices may be projected to yield faster, and generally less precise, analyses. The method of specifying the interpretation itself is reminiscent of a denotational semantics description. In particular, control flow and data flow are intertwined in the description; the user of the system cannot use the intuitive notion of control flow graph to aid in the specification of the interpreter.

II.6 Control flow – the missing link

As shown in previous sections, considerable effort has gone into devising mechanisms and tools for performing static analysis. Curiously, however, very few efforts address the problem of control flow analysis. Babich and Jazayeri's "Method of Attributes," for example, embeds control flow directly into the attribute equations; each analysis contains the same

specification implicitly. McTAG [81] and Z1 [97, 98] both assume an intermediate format in which goto statements have been eliminated. Control flow in the former is embedded into an explicitly defined fixpoint computation. In Z1 the fixpoint computation is implicit; the semantics is relatively straightforward, however, because of the compositional nature of the assumed intermediate representation.

II.6.1 Continuation passing style

In devising a general purpose flow analysis tool, independent of any particular compiler or compiler generation environment, it is necessary to address the issue of non-local control flow directly. In fact, the fundamental problem is the same as handling goto statements in denotational semantics [82]: the meaning of a given programming language statement may not be a function of its syntactic sub-components. The solution in denotational semantics is to use *continuation passing style (CPS)* where the semantics of a given component is parameterized by a function, called a *continuation*, that represents the execution of "the rest of the program." Most statements either pass their continuation on to their sub-components, or in the case of primitives, invoke it as their last operation. Goto statements, however, ignore their continuation entirely and invoke a continuation from the environment (corresponding to the destination of the goto).

Although continuation passing style was originally developed to describe the semantics of imperative languages, it has been used primarily in the area of functional programming. Continuations have been used in the implementation of ML [7] and are even directly available to the user in Scheme [71]. In terms of static analysis, continuations are frequently used in the analysis of functional programs. Shivers used continuations in his work on the static analysis of Scheme [79]. Hughes' strictness analysis is also based on continuations [38].

Given the power (and origin) of continuations, it is reasonable to wonder why they have not yet appeared in the literature as an alternative to the various forms of graph typically used to analyze imperative languages. Perhaps the reason is the inherent adequacy of a control flow graph for intraprocedural analysis: in this case there is no good reason to use continuations. Similarly, call graphs retain many of the comfortable properties of control flow graphs and are an obvious extension to cope with interprocedural control flow, though the "fit" is certainly not as good. Another possibility is that having the program in the form of a control flow graph is useful for the subsequent application of optimizing transformations. It is unclear how to apply an optimizing transformation, such as the movement of an invariant expression out of a loop, to a program represented as a set of recursive functions in continuation passing style.

From another perspective, is it a good idea to require the user of a compiler generation tool to construct a continuation passing style semantics? As shown in the previous paragraph, such a structure would likely serve only a single purpose and the user would then have to go build a control flow graph at some point anyway. A better approach is to devise an intermediate representation that can be described in terms of the abstract syntax of the language that retains the flexibility of control flow graphs but may be used to derive a continuation passing style semantics. Ideally, this structure would also seamlessly integrate the representation of intra- and interprocedural control flow. Such a structure would go a long way towards providing an appropriate abstraction amenable to automation of the construction of static analysis tools. The abstraction developed in this dissertation, the flow grammar, meets these criteria, providing an effective bridge between the practical control flow graph and the theoretical continuation passing style semantics.

II.6.2 Plumb

One direct attempt to specify control flow was Sethi's *plumb* tool [77]. It allows a user to write a continuation passing style semantics of the language which plumb converts to a (non-optimizing) code generator. His paper gives a specification for the intraprocedural aspects of C, and indicates that extension to other language constructs, such as procedure calls should be easily added. It seems, however, that this effort was not continued. Perhaps part of the reason is that the specifications were almost as difficult to read and write as the denotational semantic description they were attempting to replace. In any case, plumb has yet to appear in the literature as the basis of a flow analyzer generator.

II.6.3 On the use of grammars

At the outset it may seem somewhat unusual to use grammars as an intermediate representation in a compiler. There are, however, a number of other projects that have used grammars for similar purposes, or in manners that can be adapted for use in this research.

Aside from the conventional use of grammars to describe programming language syntax, grammars have also been used as a programming formalism in their own right. In his MSc thesis, Silverberg discusses the use of grammars as a useful programming mechanism for describing data and corresponding algorithms [80]. Michaelson discusses the use of grammars as a mechanism for specifying interpreters which permits their automatic construction [60]. The description of both the syntax and semantics of an entire Algol-like programming language is given as a two-level grammar in [13]. More directly related to flow analysis, Jones describes a method for flow analysis which computes a grammar describing the "shapes" of possible input and output arguments present in a lazy higher-order functional program [41].

More recently, research has been directed at analyzing grammars in an area known as *grammar flow analysis* [61, 43]. The idea here is to associate values from a lattice with the non-terminals of a grammar. In the *top-down* formulation, the values of non-terminals on the rhs of a production are constrained by functions of the non-terminal on the lhs. Conversely, in the *bottom-up* formulation, the value for the non-terminal on the lhs is a constrained by a function of the values of the non-terminals on the rhs. In either case, the solution to such problems is a mapping from non-terminals to lattice values which satisfy all the constraints. While the flow grammar notion presented here is not a grammar flow analysis, several of the techniques for computing fixpoint solutions in grammar flow analysis problems discussed in [43] do apply, and are studied in Chapter V. Also, Jeuring and Swierstra describe a functional approach to bottom-up grammar flow analysis problems quite similar to the flow grammar approach taken in this dissertation [39].

II.7 Summary

Successful compiler generation tools make use of the compiler writer's intuition by providing an abstraction suited to the tasks that must be performed. In the case of static analysis, research has been directed primarily at the data flow aspects of the problem. Control flow aspects have seldom been considered, and no easily specified abstraction, suited to subsequent data flow analysis, devised.

III Motivation: generic description of control flow

This chapter explores a key problem associated with automating the construction of flow analyzers: the lack of a uniform mechanism for describing intra- and interprocedural control flow. The contribution of this chapter is, by way of example, a demonstration of how a *grammar* is a *natural extension* to the usual control flow graph used for representing control flow in a compiler. This provides the main motivation for using grammars as the basis of the remainder of the dissertation.

III.1 Introduction

There are numerous compiler construction tools for automatically generating compiler modules ranging from lexical analyzers to code generators. While some of these tools are specifically aimed at automating the production of *optimizing* compilers, none has a particularly intuitive interface for specifying and solving flow analysis problems. One of the main stumbling blocks appears to be the lack of an effective mechanism for describing control flow. This chapter illustrates one possibility for specifying control flow generically

in terms of a programming language's syntax and motivates the use of grammars to model control flow. The presentation may be considered an initial design for a graphical user interface to the *FACT* tool.

III.2 The problem

For most optimizations to take place, various facts about the program under compilation must be computed. For example, if it can be shown that a variable *V* is never used after some point *P* in a program, then any storage for *V* may be freely used for other purposes when control has reached point *P*. This requires a knowledge of all possible paths in the program, and what events may occur on those paths at run-time. This information is the result of *control flow analysis* and *data flow analysis*, respectively. The development of a flow analysis tool needs a generic mechanism for describing both of these phases.

For general-purpose control flow analysis, the problem is to devise an effective means for specifying a *control flow model* in terms of the *abstract syntax* of the programming language being compiled. While there are many important aspects to this problem, two main requirements are:

1. an intuitive interface, and
2. a general control flow schema capable of accurately representing a broad range of control flow patterns.

The remainder of this chapter presents a new solution to this problem which extends the intuitive aspects of the *control flow graph* with the expressive power needed to model procedure calls. *Flow grammars* arise naturally from this extension and their study comprises subsequent chapters.

III.3 Example: AST to control flow graph

Figure 8 shows (a subset of) an abstract syntax suitable for representing Pascal statements. By associating two control flow points with each S production, it is possible to specify an arc-based control flow graph for any AST described by this grammar. Figure 9 shows the AST corresponding to the statements in Figure 5 along with an arc-based control flow graph induced from the tree. Aside from containing more arcs and nodes, the graph shown is similar to the arc-based graph in Figure 6.

III.4 From abstract syntax to control flow

Specifying the control flow for a given abstract syntax requires determining which non-terminals in the grammar represent programming language constructs that (may) give rise to control flow at run-time. Then the number of control flow points for each non-terminal must be determined, along with the control flow pattern implied by the various productions containing the selected non-terminals.

In the example above, each S non-terminal has two associated points, say *in* and *out*, which is typical for an imperative language. The control flow implied by the production $S \rightarrow S\ S$ may specified by the following schema (the non-terminals have been indexed so that they may be distinguished from each other, as in attribute grammar semantic rules):

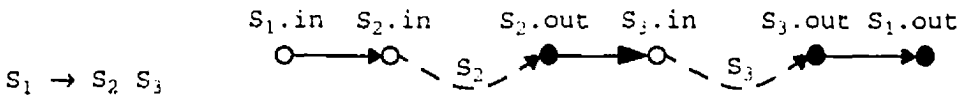


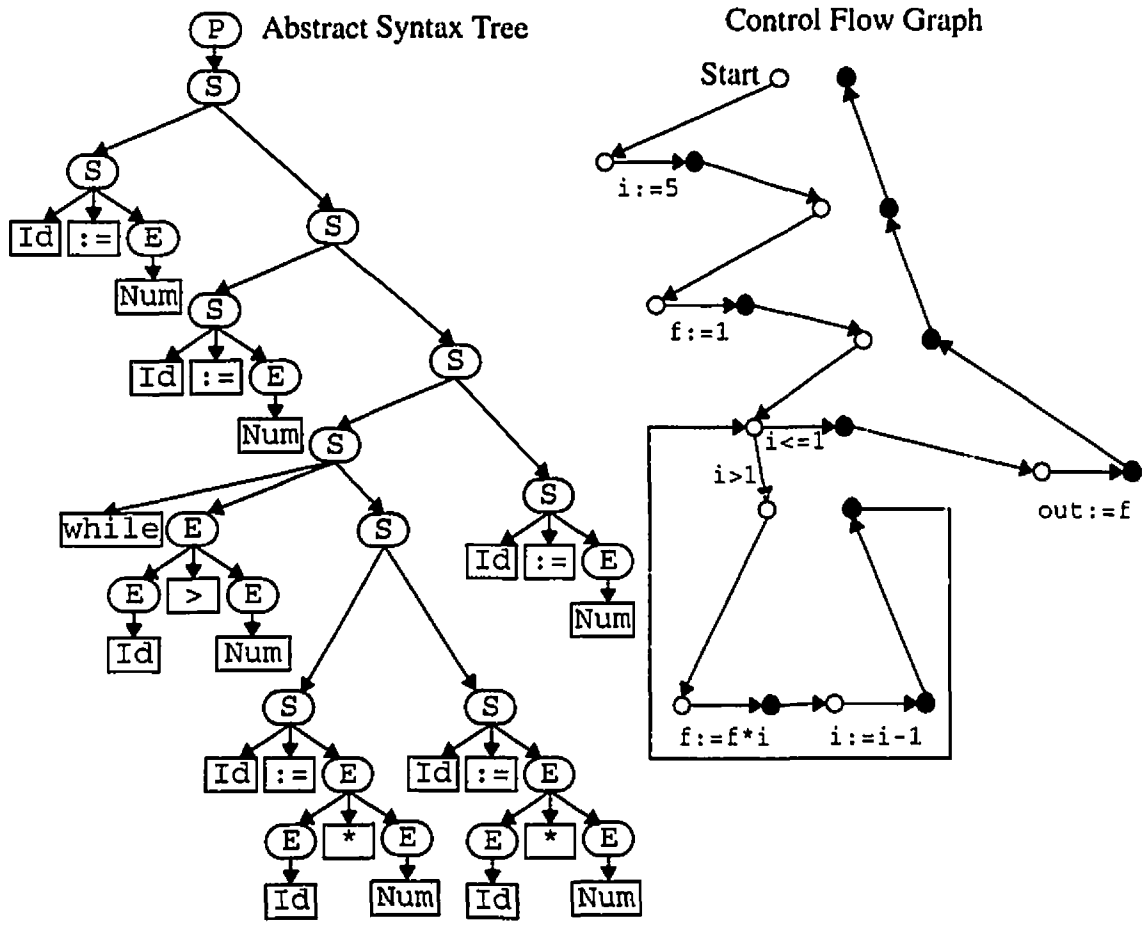
Figure 8 Abstract syntax for a small subset of Pascal statements

```
P → S

S → S S | 'while' E S | Id ':=' E

E → Id | Num | E '-' E | E '*' E | E '>' E
```

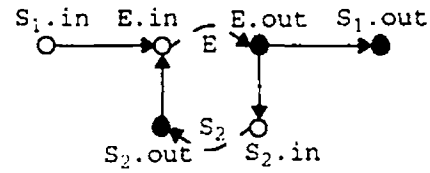
Figure 9 AST and control flow graph with "redundant" nodes of program in Figure 5



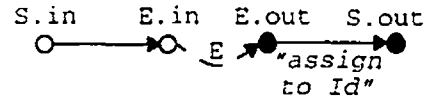
If a given statement is composed of two nested statements, then execution of the entire statement may be effected by executing the first nested statement and then the second. The arcs in the schema represent this by making the start of the outer statement synonymous with the start of the first inner statement (S_1.in, S_2.in), the end of the first inner statement synonymous with the start of the second inner statement (S_2.out, S_3.in), and, finally, the end of the second inner statement synonymous with the end of the outer statement (S_3.out, S_1.out). Note that S_2 and S_3 derive subgraphs (indicated by the broken arcs) which are merged with the schema subgraph to form a graph for the complete statement, in much the same way the attribute dependency relation in an attribute grammar is computed [54]. The entire process also bears a strong resemblance to the well-known algorithm for converting a regular expression to a non-deterministic finite automaton [3], the only difference being that the edges are not labelled with the empty string.

Schema for the remainder of the abstract grammar are equally straightforward to construct:

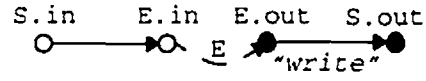
S_1 → 'while' E S_2



S → Id ':=' E



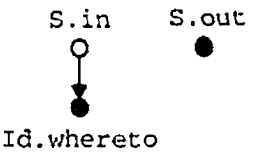
S → 'write' E



Note that in the case of assignment and write statements a run-time action occurs that changes the state of the program. These actions are denoted by italicized labels on the arcs.

As described thus far, control flow among the elements on the right-hand side of a given production is expressible. This includes the "structured programming constructs": sequencing, alternation (if/then/else) and repetition (while/do). It does not, however, allow for more global control flow, including exits from the middle of loops and goto statements. These types of constructs may be handled by allowing the definition of control points in other components of the compiler, such as the symbol table, and importing them to the schema description. In the case of the goto statement, for example, a control point may be associated with the identifier of a label, say in the attribute `Id.whereto`. This gives rise to the following schema:

```
S → 'goto' Id
```

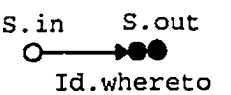


where the gray circle denotes an "external" attribute.

III.5 Procedure calls

The nested nature of procedure calls makes their accurate description with a graph model impossible [78]. The following schema illustrates the desired control flow pattern of a procedure call:

```
S → 'call' Id
```



which means: when control reaches a call statement, it should flow to the beginning of the called procedure (`Id.whereto`) and, upon return, control should flow to the point `S.out`. Note that this is not the same as connecting the `out` node of the last statement(s) of the called procedure with `S.out`, which specifies that control may return to any call site which calls the procedure. Although this behaviour is a possible approximation to the true control flow, it is insufficient for effective interprocedural analysis.

This schema is somewhat more subtle than it may first appear since, as stated above, a graph cannot accurately represent the control flow present in a procedure call. The distinction lay in the juxtaposition of the two nodes on the right hand side of the schema, which introduces a "sub-level" of control flow. In this context, the external gray node represents the effect of all possible executions of the whole of a subprogram, given that control has reached a particular call statement. From a slightly different point of view, this schema can be thought of as *generating* possible executions of a subprogram followed by possible executions of the rest of the program (i.e., statements after the call statement). Thus, the control flow is very much like a production in a context-free grammar:

```
S.in ::= Id.whereto S.out
```

which may be interpreted as: the effect of executing a statement consisting of a call (`S.in`) may be generated by generating all possible executions of the subprogram (`Id.whereto`) followed by whatever may be generated after this call statement (`S.out`).

In fact, all of the graph constructors above may be viewed as productions. With the addition of terminals to indicate the semantics actions that occur between two points, a grammar model of the source program may be constructed. The strings generated by this model reflect possible executions that may occur at run-time. Furthermore, by applying different meanings to the terminals, different data flow analyses may be performed.

III.6 Summary

This chapter addressed the problem of specifying the control flow of programs may be specified in terms of the abstract syntax of the programming language. Construction of an application with a graphical user interface allowing the annotation of a grammar with the graph construction schema shown would be straightforward, and would result in an intuitive tool for use in a compiler generation system. Modeling the effects of procedure and function calls led naturally to the notion of a *(control) flow grammar*. The remainder of this dissertation explores the utility of this powerful new control flow model.

IV Grammars as a basis for semantic description

In this chapter, the use of grammars is explored as a way of describing control flow. The main contribution is the definition of the *flow grammar*, an original model for the uniform representation of intraprocedural and interprocedural control flow, further addressing one of the key problems outlined in Section I.3. The initial sections of this chapter are intended as an introduction for compiler writers who need to understand the underlying principles of flow grammars. It then demonstrated that finding the greatest fixed point of a system of equations derived from a flow grammar safely approximates the meet over all paths generated by the grammar. A subsequent section describes a secondary contribution: an illustration of the relationship between flow grammars and continuation passing style semantics: by abstracting a program in continuation passing style semantics, a flow grammar arises naturally.

IV.1 Overview

The flow analysis of a program must be *conservative*, in the sense that any facts determined about execution of the program must be true regardless of which execution path actually occurs at run-time. This chapter elaborates the notion of using a grammar to represent possible executions introduced in Chapter III by considering the semantics of execution paths. Initially, single executions are addressed. Sets of executions, as needed by flow analysis, are then considered and the set equations resulting from programs with simple procedure calls are found to be equivalent to context free grammars. Finally, the relationship between flow grammars and continuation passing style semantics is considered.

IV.2 Introduction

For a sequential program, an execution may be viewed as a list of the state changing actions performed at run-time [34]. Consider the program fragment in Figure 10, for example. Let the numbers to the left of the code be *program points*, that is, places in the program *between* (potentially) consecutive actions. From this point of view, an execution of this program may be written as a sequence of pairs of points, control flowing from one point to the next. There is only one possible execution of this code, which may be written as the sequence of pairs:

‹1,2› ‹2,3› ‹3,4› ‹4,5› ‹5,6› ‹6,3› ‹3,4› ‹4,5› ‹5,6› ‹6,3› ‹3,7› ‹7,8›

Each pair represents some set of run-time actions that are performed between the two named points. These actions may be defined mathematically by a semantic mapping from pairs to functions on a domain. The effect of executing part of a path may be realized by composing the semantic functions of each pair in the sequence.

Determining the effect of a single execution path, however, does not address the problem of flow analysis. The fact computed for each point in the program must be true, regardless of which path was taken to get to the point. Thus, it is necessary to have a structure representing all possible execution paths along with a mechanism for combining the effects of those paths. This chapter describes a new approach to the representation of the paths, a structure that parallels the events which occur at run-time.

Figure 10 Sample Pascal program.

```
    PROGRAM test(output);
    VAR i:INTEGER;
        f:INTEGER;
    BEGIN
1     i := 3;
2     f := 1;
3     WHILE i > 1 DO BEGIN
4       f := f * i;
5       i := i - 1
6     END;
7     write(f)
8   END.
```

IV.3 Notation

The notation used for the execution path above contains a certain amount of redundancy: for many of the pairs, the initial point uniquely determines the second point in the pair. For example, ‹1,2› has this property and is subsequently written as t_1. Control splits at point 3, however, and thus point 3 occurs as the first point in two pairs: ‹3,4› and ‹3,7›. These pairs

are subsequently written as $t_{3/4}$ and $t_{3/7}$, respectively. The letter "t" represents the fact that these pairs are ultimately terminals in a grammar that represents the source program. Finally, one more transformation simplifies much of the notation in the remainder of the dissertation: the *reverse* of the above string is written as its execution path, which corresponds closely to the functional composition of the underlying components. Thus, the above execution is written as:

$$t_7\ t_{3/7}\ t_6\ t_5\ t_4\ t_{3/4}\ t_6\ t_5\ t_4\ t_{3/4}\ t_2\ t_1 \tag{1}$$

IV.4 Semantics of a single execution path

An execution path may be used for a variety of purposes. In general, a *semantics* is specified: each symbol is mapped to a function on a domain representing the *state* of the program. The function computes the effect of executing the code represented by the symbol. On the one hand, the state in a *concrete semantics* consists of the values of all variables, as well as any I/O streams, in the program. On the other hand, a flow analysis defines an *abstract semantics* in which the state represents only the information needed to compute the desired facts. The remainder of this section explores a concrete and an abstract semantics of the sample Pascal program shown above to demonstrate these ideas and motivate the next section on the semantics of multiple execution paths.

IV.4.1 Concrete semantics

Defining a concrete semantics involves defining both a domain for the program state and a mapping from symbols to functions on that domain. Computing the effect of an execution path involves composing the functions represented by each symbol in the path, in the order they are encountered. Applying the resulting function to an initial state yields the state of the program when execution reaches the end of the path.

Program state

For the example program, the program state consists of the values of the two variables, `f` and `i`, and, since there is a "`write`" statement, the contents of the output stream. This may be represented as a triple ‹*f*, *i*, *o*›, where *f* and *i* are the values of the variables `f` and `i`, and *o* is a list of the output of the program. A distinguished value, "?," is used to indicate the value of a variable that is not defined, as at point 1 of the program, immediately prior to execution. The output is written as a comma-separated list of values, with ‹› representing the empty list. Combining the notation yields ‹?,?,‹›› as the initial state of the program. Finally, there is a distinguished state, *error*, that represents an erroneous computation, such as division by zero. Program execution is aborted when this state is entered.

Semantics of single symbols

For the concrete semantics of a single execution path, each symbol is mapped to a function on the above program state. The effect of execution proceeding from point 1 to point 2, for example, is to assign 3 to the variable `i`, which may be written as the following lambda expression:

$$t[\![t_1]\!] = \lambda\langle f, i, o\rangle.\langle f, 3, o\rangle$$

where t is the semantic functional of (single) terminal symbols

The other assignment statements are handled similarly:

$$t[\![t_2]\!] = \lambda\langle f, i, o\rangle.\langle 1, i, o\rangle$$
$$t[\![t_4]\!] = \lambda\langle f, i, o\rangle.\langle f * i, i, o\rangle$$
$$t[\![t_5]\!] = \lambda\langle f, i, o\rangle.\langle f, i-1, o\rangle$$

In a concrete semantics, both the conditional represented by symbols $t_{3/4}$ and $t_{3/7}$, and the jump from the end to beginning of the loop (t_6), have no effect on the state of the program, and represent the identity function:

$$t[\![t_{3/4}]\!] = t[\![t_{3/7}]\!] = t[\![t_6]\!] = \lambda S.S$$

Finally, the "write" statement appends a value to the output stream:

$$t[\![t_7]\!] = \lambda\langle f,i,o\rangle.\langle f,i,o \S \langle f\rangle\rangle \qquad \textit{where } \S \text{ represents list concatenation}$$

Semantics of single execution paths

Given the above mappings for each symbol, the effect of an execution path is simply the composition of the corresponding functions. For the example execution path (1), the function is:

$$\begin{aligned} &t^*[\![t_7\ t_{3/7}\ t_6\ t_5\ t_4\ t_{3/4}\ t_6\ t_5\ t_4\ t_{3/4}\ t_2\ t_1]\!] = \\ &\quad t[\![t_7]\!] \circ t[\![t_{3/7}]\!] \circ t[\![t_6]\!] \circ t[\![t_5]\!] \circ t[\![t_4]\!] \circ \\ &\quad t[\![t_{3/4}]\!] \circ t[\![t_6]\!] \circ t[\![t_5]\!] \circ t[\![t_4]\!] \circ t[\![t_{3/4}]\!] \circ t[\![t_2]\!] \circ t[\![t_1]\!] \\ &\text{where } t^* \text{ is the semantic functional of strings of terminal symbols} \end{aligned}$$

That is, the meaning of a single path is the composition of the meanings of its individual components. Note that the functions are applied in the order in which the corresponding statements are executed. When applied to the initial state, the above function yields the state ‹6,1,‹6››, representing the fact that the final value of `f` is 6, the final value of `i` is 1, and the output stream after execution contains the single number 6. More precisely, define t^* as follows:

Definition 37: The *forward semantic functional*, t^*, is defined as follows:

$$t^*:T^* \rightarrow F \text{ as:} \begin{cases} t^*[\![\varepsilon]\!] = \lambda s.s \\ t^*[\![t\alpha]\!] = t^*[\![t]\!] \circ t[\![\alpha]\!] \quad t \in T \end{cases}$$

where F is the set of all total functions mapping states to states and t maps terminal symbols to functions over concrete states.

IV.4.2 Abstract semantics

In an abstract semantics, the program state contains only partial information about what state the program may be in at each point in the program. The types of partial information range from whether or not a statement is reachable (in principle) during program execution [2], to whether or not a value always has the same value at a given point.

While information always flows *forwards*, from the beginning to the end of the program, in a concrete semantics, partial information in an abstract semantics may flow forwards, backwards [18, 44], or even in both directions [62]. Both a forward and a backward analysis of the example program are presented below.

Live variables: a backward analysis

Now consider computing live variable sets for the example program. There are only two variables, `i` and `f`, so an abstract state is simply a subset of {`i`,`f`}. The only information known prior to analysis is that neither variable is used once execution is complete. Thus, it is the *final* state, at point 8 of the program that is known to be {}. Because `f` is used in the "`write`" statement represented by t_7, the state at point 7 must be $\{\} \cup \{f\} = \{f\}$. That is,

$$t[\![t_7]\!] = \lambda S.S \cup \{f\}$$

and $t[\![t_7]\!]$ is applied to the state known at point 8 of the program.[1] Clearly, the information is flowing backwards from the end of the program towards the beginning. As with the concrete analysis above, each symbol represents a set of actions that is mapped to a function on the program state, which is now an abstraction of the concrete state.

In a backwards analysis, abstract functions for the assignment statements mimic the inverse of what happens during execution (again because this is a backwards analysis). Consider the assignment represented by t_4, "`f:=f*i`". At run-time, the last event is the assignment of the value computed for the subexpression "`f*i`" to the variable `f`. Thus, between computation of the expression and the store of that value to memory, `f` is not live and should be removed from the abstract state. During computation of the expression, both the variables `f` and `i` are used, and so after the effect of the assignment has been considered, `f` and `i` should be added to the abstract state:

$$t[\![t_4]\!] = \lambda S.(S-\{f\})\cup\{i,f\} = \lambda S.S\cup\{i,f\}$$

The other assignments and expressions in the program map similarly:

$$t[\![t_1]\!] = \lambda S.S-\{i\}$$
$$t[\![t_2]\!] = \lambda S.S-\{f\}$$
$$t[\![t_{3/4}]\!] = t[\![t_{3/7}]\!] = \lambda S.S\cup\{i\}$$
$$t[\![t_5]\!] = \lambda S.(S-\{i\})\cup\{i\} = \lambda S.S\cup\{i\}$$

Finally, the jump represented by t_6 has no effect on the abstract state:

$$t[\![t_6]\!] = \lambda S.S$$

1. Note that t in this subsection is a mapping to functions over abstract states, not concrete states as in the previous subsections (cf. Definition 37).

Determining the set of variables live at a given point during execution is similar to determining the program state in a concrete semantics, except that the functions are composed in the reverse order because information is flowing in the opposite direction. To determine the live variables at the beginning of the execution path above:

$$t[\![t_1]\!] \circ t[\![t_2]\!] \circ t[\![t_{3/4}]\!] \circ t[\![t_4]\!] \circ t[\![t_5]\!] \circ t[\![t_6]\!] \circ t[\![t_{3/4}]\!] \circ t[\![t_4]\!] \circ t[\![t_5]\!] \circ t[\![t_6]\!] \circ t[\![t_{3/7}]\!] \circ t[\![t_7]\!](\{\}) = \{\}$$

As there are no variables live at the beginning of the program for this execution, no variables are used before being assigned a value on this run.

Constant propagation: a forward analysis

A well-known and important forward analysis is *constant propagation* [6, 44, 51, 91]. The essential idea is to determine when variables have constant values, and use this information to simplify subsequent expressions. In this case, information is clearly flowing forward.

The single path semantics is similar to the concrete semantics above. Abstract states consist of the values of all variables, as well as the output stream. However, if the program performs input, the input stream is not assumed to be known. The semantic mapping of a "`read`" statement sets the value of the argument variable to ?, indicating that the value is unknown. In the case of the example program in Figure 10 and the single execution path (1), the final state is the same as for the concrete semantics: ‹6,1,‹6››.

To this point only *dynamic* states have been considered. Conceptually, semantic functions have been constructed to determine the effect of running a program, or part of a program, to determine the program state after a given set of actions have been executed. A program state is computed for a given point each time control passes that point. The goal

of static analysis is to accumulate all such states, not only across a single execution, but across any possible execution of the program at all. This requires a mechanism for *merging* program states and also necessitates the consideration of *multiple execution paths.*

IV.5 Merging abstract states

As described in Kam and Ullman [44], abstract program states used in flow analyses are generally defined as lattices. Each element of the lattice represents one abstract program state, and the meet operator on the lattice is used to merge two elements, say a and b, into a third that represents an abstract state consistent with both a and b.

IV.5.1 Example: live variables

In the case of live variables, the lattice consists of the powerset of the set of variables in the program with a meet operator consisting of set union. More formally, let Var be the set of variables in the program being analyzed. Then the live variable lattice is:

$$LV = \langle 2^{Var}, \cup \rangle \tag{2}$$

The elements of the lattice are ordered by the superset operator. That is, $x \leq_{LV} y$ iff $x \supseteq y$, $x,y \in 2^{Var}$.

IV.5.2 Example: constant propagation

For constant propagation, the lattice elements are partial maps from the variables in the program to their current values. Let Var be the set of variables in the program being analyzed, and R be the set of values that can be assigned to the variables in Var. A lattice ele-

ment *e* may then be seen as a subset of Var×R. To merge two such lattice elements, the set of variable/value pairs must be intersected, since a variable has a constant value in the combined abstract state iff it had the same value in each of the preceding states. However, there is no natural top element, so one must be added. More formally, the constant propagation lattice may be defined as:

$$
\begin{aligned}
&CP = \langle 2^{Var \times R} \cup \top_{CP}, \wedge_{CP} \rangle \\
&\textit{where} \\
&\quad x \wedge_{CP} \top_{CP} = \top_{CP} \wedge_{CP} x = x, \forall x \in 2^{V \times R} \\
&\quad x \wedge_{CP} y = x \cap y, \forall x,y \in 2^{V \times R}
\end{aligned} \tag{3}
$$

In this case, the lattice is ordered by subsets: $x \leq_{CP} y$ iff $y = \top_{CP}$ or $x,y \in 2^{Var \times R}$ and $x \subseteq y$.

IV.6 Semantics of multiple execution paths

If the compiler is to make use of the results of a static analysis for the purposes of optimization, any information provided by that analysis must be true in *all* circumstances. Otherwise, incorrect program transformations may be made introducing errors not present in the original source program. This implies that all feasible execution paths must be considered by the static analyzer when it computes abstract states for each point in the program. This section extends the abstract semantics of single execution paths shown above to *sets* of executions, rather than single executions, in preparation for static analysis.

IV.6.1 Preliminaries

The semantics of a set consisting of a single execution path should coincide with the semantics described above. Specifically, if $t_1\ t_2 \ldots t_n$ is an execution path, define:

$\mathcal{T}_{T,t}[\![\{t_1\ t_2\ ...\ t_n\}]\!](x) = t^*[\![t_1\ t_2\ ...\ t_n]\!](x)$
where

$\mathcal{T}_{T,t}$ is the semantic function for sets of execution paths over an alphabet T and $t: T \rightarrow F$ maps members of the alphabet into functions on a lattice

Note that the semantic brackets on the left differ from those on the right: the former applies to a set of paths, the latter to a single path.

The meaning of a finite set of execution paths is the meet of the individual paths, for all elements in the lattice, L, representing the abstract state. For example, if $\alpha_1, \alpha_2, ..., \alpha_m$ are finite execution paths, then:

$$\forall x \in L: \mathcal{T}_{T,t}[\![\{\alpha_1, \alpha_2, ..., \alpha_m\}]\!](x) = \wedge \{\ t^*[\![\alpha_i]\!](x) \mid 1 \le i \le m\ \} \qquad (4)$$

Given that each α_i is finite, and that t maps each terminal to a monotonic function, these values are clearly computable. Generalizing to infinite sets of execution paths requires somewhat more mathematical machinery, introduced in the following sections, after which the question of the meaning of infinite sets of execution paths is considered.

IV.6.2 Infinite sets of execution paths

Most interesting programs have an unbounded number of possible executions. This set of executions may be represented by a family of set equations reflecting the source program. Returning to the example program in Figure 10, let S_i represent the set of execution paths that start at the beginning of the program and ending at point i, $1 \le i \le 8$. The control flow present in the program yields the family of set equations shown in Figure 11.

Figure 11 Family of set equations representing control flow of program in Figure 10.

$$
\begin{aligned}
S_1 &= \{\varepsilon\} & S_5 &= \{t_4\} \bullet S_4 \\
S_2 &= \{t_1\} \bullet S_1 & S_6 &= \{t_5\} \bullet S_5 \\
S_3 &= \{t_2\} \bullet S_2 \cup \{t_6\} \bullet S_6 & S_7 &= \{t_{3/7}\} \bullet S_3 \\
S_4 &= \{t_{3/4}\} \bullet S_3 & S_8 &= \{t_7\} \bullet S_7
\end{aligned}
$$

As point 1 is the beginning of the program, the set of actions consists solely of the empty string. For point 2, the set is the single path from the beginning of the program to point 2, $\{t_1\}$.

The equations for S_3 through S_6 are all cyclic. Point 3, for example may be expanded by substitution as follows:

$$
\begin{aligned}
S_3 &= \{t_2\} \bullet S_2 \cup \{t_6\} \bullet S_6 \\
&= \{t_2\} \bullet S_2 \cup \{t_6\} \bullet \{t_5\} \bullet S_5 \\
&= \{t_2\} \bullet S_2 \cup \{t_6\} \bullet \{t_5\} \bullet \{t_4\} \bullet S_4 \\
&= \{t_2\} \bullet S_2 \cup \{t_6\} \bullet \{t_5\} \bullet \{t_4\} \bullet \{t_{3/4}\} \bullet S_3
\end{aligned}
$$

Any solution to this equation must contain an infinite number of paths, since S_3 contains at least one path (t_2t_1), and the recursive component of the equation "adds" a path for each path in the solution. There are, however, an infinite number of solutions to this equation. For flow analysis, the desired solution is the one with the fewest paths (called the *least fixed point* (*LFP*) solution): each of the strings in the smallest solution represents a possible path under the usual data flow assumptions (see Section II.3.3). Larger solutions contain extraneous strings that do not represent possible execution paths. Including such paths may result in unnecessarily conservative information (e.g., a variable is deemed to be live at a point where it is not) or incorrect information (e.g., an expression is deemed *partially available* at a point, that is computed on at least one path to the point, when it is not). In the case of S_3 above, the LFP solution is an infinite set:

$$S_3 = \{t_2t_1, t_6t_5t_4t_{3/4}t_2t_1, t_6t_5t_4t_{3/4}t_6t_5t_4t_{3/4}t_2t_1, \ldots\}$$

The LFP solutions for the other equations in the cycle, S_4, S_5 and S_6 are similar. Finally, the solution for S_7 is also infinite, since it is defined in terms of S_6, as is S_8 by transitivity.

IV.6.3 Execution path equations and grammars

The families of set equations describing execution paths bear a strong resemblance to the grammars of formal language theory. In fact, for each of the families of equation above, the LFP solution is equal to a language generated by a grammar. For example, for the equations in Figure 11 yield LFP solutions equivalent to the strings generated by the following grammar:

$S_1 ::= \varepsilon$
$S_2 ::= t_1\ S_1$
$S_3 ::= t_2\ S_2 \mid t_6\ S_6$
$S_4 ::= t_{3/4}\ S_3$
$S_5 ::= t_4\ S_4$
$S_6 ::= t_5\ S_5$
$S_7 ::= t_{3/7}\ S_3$
$S_8 ::= t_7\ S_7$

IV.6.4 Grammars as constraint systems

Another view of a grammar that is useful in this endeavour is that of a constraint system. Each production in a grammar specifies a set inclusion. For example, the grammar in the

preceding section may be viewed as the following system of set inequalities whose LFP is the language generated by the grammar:

$$
\begin{array}{ll}
S_1 \subseteq \{\varepsilon\} & S_5 \subseteq \{t_4\} \bullet S_4 \\
S_2 \subseteq \{t_1\} \bullet S_1 & S_6 \subseteq \{t_5\} \bullet S_5 \\
S_3 \subseteq \{t_2\} \bullet S_2 & S_7 \subseteq \{t_{3/7}\} \bullet S_3 \\
S_3 \subseteq \{t_6\} \bullet S_6 & S_8 \subseteq \{t_7\} \bullet S_7 \\
S_4 \subseteq \{t_{3/4}\} \bullet S_3 &
\end{array}
$$

In terms of an abstract semantics, each inequality implies one or more constraints on the abstract state represented by each non-terminal. The utility of this view is that there is no implied "direction" for the constraints. Furthermore, it mimics the inequalities derived for the purposes of abstract semantic analysis.

We should note that we are slightly abusing notation here by using S_i as both a non-terminal and a set: they are not strictly equivalent usages. Specifically, S_i as a non-terminal generates a specific set of strings while S_i as a set variable has many possible values. The value S_i has in the *least* fixed point solution to the above equations is equal to the set of strings generated by the corresponding non-terminal in the grammar.

IV.6.5 Flow grammars

Combining the preceding ideas yields the following definition of a flow grammar, along with corresponding notions of derivation and generation.

Definition 38: A *flow grammar* (*FG*) is a quadruple G = ‹V, T, P, S› where V is the set of *flow non-terminals* (*FN*s), T is the set *flow terminals* (*FT*s), P is the set of *flow productions* (*FP*s), and S is the *start flow non-terminal* (*SFN*). Each FN represents a control flow point in the program, while each FT represents a primitive program component with respect to the flow analysis. Control flow is represented by the FPs, which are pairs written $\alpha ::= \beta$, where $\alpha \in V$, $\beta \in (V \cup T)^*$. The combined alphabet, $V \cup T$, is called the *flow vocabulary*, after the notion of vocabulary in formal language theory. The SFN corresponds to a point of interest in the program. Finally, we distinguish between *regular flow grammrs (RFGs)* and *context-free flow grammars (CFGs)*.

Figure 12 shows the FG for the program fragment in Figure 10. Each FN is subscripted with its corresponding point in the code. FTs are subscripted with a program point, and when necessary, an additional notation to indicate context. S_1, the SFN, represents the start of the program fragment. The FP "$S_2 ::= t_1\ S_1$," shows how control flow is represented: control reaches point 2, denoted by S_2, from point 1, denoted by S_1, by executing the action corresponding to t_1, the assignment "`i:=5`". Extra context is needed in the case of the expression at the top of the loop: the FT $t_{3/4}$ represents the expression "`i>1`" evaluating to true, while the FT $t_{3/7}$ represents evaluation to false; two FPs use these FTs to represent the join in control flow introduced by the head of the loop.[1]

Figure 12 Complete flow grammar for program fragment in Figure 10.

$G = \langle V, T, S, P \rangle$ where

$V = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}$

$N = \{t_1, t_2, t_{3/4}, t_{3/7}, t_4, t_5, t_6, t_7, t_8\}$

$S = S_8$

$P = \{\ S_1 ::= \varepsilon,$ $\quad S_5 ::= t_4\ S_4,$

$\quad S_2 ::= t_1\ S_1,$ $\quad S_6 ::= t_5\ S_5,$

$\quad S_3 ::= t_2\ S_2 \mid t_6\ S_6,$ $\quad S_7 ::= t_{3/7}\ S_3,$

$\quad S_4 ::= t_{3/4}\ S_3,$ $\quad S_8 ::= t_7\ S_7$ $\quad \}$

1. In Section IV.6.9 a slightly different form of flow grammar is introduced which is more useful in the remainder of this dissertation. The current form is useful for the intuitive introduction of the concept.

Generation of execution paths

As discussed above, the goal of control flow analysis is to develop a finite representation of a program's possible execution paths. Thus, it is natural to ask what execution paths are represented by a given flow grammar, G= ‹V, T, P, S›. Perhaps not surprisingly, the answer is from the formal language theory notions of *derivation* and *generation*.

Consider the following sequence of strings, where each is related to the next by ⊨:

$$\begin{aligned} S_8 &\models t_7\, S_7 \\ &\models t_7\, t_{3/7}\, S_3 \\ &\models t_7\, t_{3/7}\, t_2\, S_2 \\ &\models t_7\, t_{3/7}\, t_2\, t_1\, S_1 \\ &\models t_7\, t_{3/7}\, t_2\, t_1 \end{aligned} \tag{5}$$

S_8 is an incomplete execution path representing the execution of the entire program; $t_7\, S_7$ is an incomplete execution path which ends with the statement "`write(f)`" followed by (that is, preceded at run-time by) an execution path derivable from S_7, and so on. Finally, a string consisting of only FTs is derived: this represents one complete execution path.

Definition 39: The set of execution paths, or *flow strings*, represented by a FG G:

$$\mathcal{L}_G = \{\alpha \mid S \models^* \alpha,\ \alpha \in T^*\}$$

is called the *flow language* (*FL*) *generated* by G; G is said to *generate* each string in $\mathcal{L}_G$.

Three execution paths generated by the FG in Figure 12 are:

$t_7\, t_{3/7}\, t_2\, t_1$

$t_7\, t_{3/7}\, t_6\, t_5\, t_4\, t_{3/4}\, t_2\, t_1$

$t_7\, t_{3/7}\, t_6\, t_5\, t_4\, t_{3/4}\, t_6\, t_5\, t_4\, t_{3/4}\, t_6\, t_5\, t_4\, t_{3/4}\, t_2\, t_1$

Note that, in the absence of aliasing, only the third flow string listed is actually possible at run-time; it is possible, though expensive, for a flow analysis to determine the final value of the variable f statically.

IV.6.6 Semantics of infinite sets of execution paths

Returning to the semantics of execution paths, the goal is to compute the semantics for infinite sets of execution paths. A semantics for sets of execution paths may be specified by mapping each set to a function representing the effect of executing *any* path in the set. In the case of S_3, for example:

$$\begin{aligned}\mathcal{T}_{T,t}[\![S_3]\!](x) &= \mathcal{T}_{T,t}[\![\{t_2t_1, t_6t_5t_4t_{3/4}t_2t_1, t_6t_5t_4t_{3/4}t_6t_5t_4t_{3/4}t_2t_1, \ldots\}]\!](x) \\ &= \mathcal{T}_{T,t}[\![\{t_2t_1\}]\!](x) \wedge \mathcal{T}_{T,t}[\![\{t_6t_5t_4t_{3/4}t_2t_1\}]\!](x) \wedge \ldots\end{aligned}$$

Or, more generally, if $S = \{ \alpha_1, \alpha_2, \ldots \}$ where each α_i is finite, then $\mathcal{T}_{T,t}[\![S]\!]$ is defined as follows:

$$\mathcal{T}_{T,t}[\![S]\!](x) = \mathcal{T}_{T,t}[\![\{\alpha_1\}]\!](x) \wedge \mathcal{T}_{T,t}[\![\{\alpha_2\}]\!](x) \wedge \ldots \quad (6)$$

Kam and Ullman call this the *Meet Over All Paths (MOP)* solution. Perhaps somewhat surprisingly, the MOP solution is not computable in general [44]. Consider the program fragment in Figure 13: the value of c is obviously 7 at point 7, but neither a nor b have known definite values. In the general case of an arbitrary program, we would have to keep a potentially unbounded amount of state around to make this kind of determination. This is, of course, impossible. Instead we must content ourselves with an approximation to the MOP solution which is conservative.

Figure 13 Constant propagation fails to detect a constant value

Point	Program
1	`if (cond) then`
2	`    a := 3; b := 4;`
3	`else`
4	`    a := 4; b := 3;`
5	`fi;`
6	`c := a + b;`
7	`/* c has the value 7 */`

Returning to the running example, we would like to compute approximations for each $\mathcal{T}_{T,t}[\![S_i]\!]$, $1 \le i \le 8$, about which we know the following:

$$\mathcal{T}_{T,t}[\![S_1]\!] = \mathcal{T}_{T,t}[\![\{\varepsilon\}]\!]$$
$$\mathcal{T}_{T,t}[\![S_2]\!] = \mathcal{T}_{T,t}[\![\{t_1\} \bullet S_1]\!]$$
$$\mathcal{T}_{T,t}[\![S_3]\!] = \mathcal{T}_{T,t}[\![\{t_2\} \bullet S_2 \cup \{t_6\} \bullet S_6]\!]$$
$$\mathcal{T}_{T,t}[\![S_4]\!] = \mathcal{T}_{T,t}[\![\{t_{3/4}\} \bullet S_3]\!]$$
$$\mathcal{T}_{T,t}[\![S_5]\!] = \mathcal{T}_{T,t}[\![\{t_4\} \bullet S_4]\!]$$
$$\mathcal{T}_{T,t}[\![S_6]\!] = \mathcal{T}_{T,t}[\![\{t_5\} \bullet S_5]\!]$$
$$\mathcal{T}_{T,t}[\![S_7]\!] = \mathcal{T}_{T,t}[\![\{t_{3/7}\} \bullet S_3]\!]$$
$$\mathcal{T}_{T,t}[\![S_8]\!] = \mathcal{T}_{T,t}[\![\{t_7\} \bullet S_7]\!]$$

Intuitively, the structure of the equations suggests that we should attempt to decompose the right hand sides in terms of the subcomponents in the semantic brackets. In so doing, the set construction operators, $\bullet$ and $\cup$, would be mapped to operations combining functions over abstract states. These operations must be guaranteed to consider the effect of all paths in the set resulting from the specified operation. In the process, however, we lose equality in the above equations and must settle for an approximation, as shown below.

As in Definition 37, the function associated with the set $S_1 = \{\varepsilon\}$ should be the identity function (though on an abstract, rather than concrete, domain in this case):

$$\mathcal{T}_{T,t}[\![S_1]\!] = \mathcal{T}_{T,t}[\![\{\varepsilon\}]\!] = t^*[\![\varepsilon]\!] = \lambda s.s$$

By construction, the meaning of the set concatenation operator is related to function composition. However, as is demonstrated below:

$$\mathcal{T}_{T,t}[\![X \bullet Y]\!] \neq \mathcal{T}_{T,t}[\![X]\!] \circ \mathcal{T}_{T,t}[\![Y]\!]$$

Rather, the best possible, in general, is:

$$\mathcal{T}_{T,t}[\![X \bullet Y]\!] \geq \mathcal{T}_{T,t}[\![X]\!] \circ \mathcal{T}_{T,t}[\![Y]\!]$$

for some decomposition of the paths in X•Y. Similarly, the meaning of the set union operator may also be defined in more than one way, the most obvious being the point-wise meet of the associated functions, but again, only inequality holds:

$$\mathcal{T}_{T,t}[\![X \cup Y]\!] \geq \lambda x.\ \mathcal{T}_{T,t}[\![X]\!](x) \wedge \mathcal{T}_{T,t}[\![Y]\!](x)$$

To gain an appreciation of why only inequalities hold, consider the path set $\{t_0t_1, t_0t_2\}$, which may be decomposed as $\{t_0\} \bullet \{t_1, t_2\}$:

$$\begin{aligned}
&\mathcal{T}_{T,t}[\![\{t_0\}]\!] \circ \mathcal{T}_{T,t}[\![\{t_1,t_2\}]\!] \\
&\quad = \mathcal{T}_{T,t}[\![\{t_0\}]\!] \circ (\lambda l.\mathcal{T}_{T,t}[\![\{t_1\}]\!](l) \wedge \mathcal{T}_{T,t}[\![\{t_2\}]\!](l)) \\
&\quad = t^*[\![t_0]\!] \circ (\lambda l.t^*[\![t_1]\!](l) \wedge t^*[\![t_2]\!](l)) \\
&\quad = t[\![t_0]\!] \circ (\lambda l.t[\![t_1]\!](l) \wedge t[\![t_2]\!](l)) \\
&\quad = \lambda m.(t[\![t_0]\!] \circ (\lambda l.t[\![t_1]\!](l) \wedge t[\![t_2]\!](l))(m)) \\
&\quad = \lambda m.(t[\![t_0]\!](t[\![t_1]\!](m) \wedge t[\![t_2]\!](m)))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{T}_{T,t}[\![\{t_0t_1, t_0t_2\}]\!] \\
&\quad = \mathcal{T}_{T,t}[\![\{t_0t_1\}]\!] \wedge \mathcal{T}_{T,t}[\![\{t_0t_2\}]\!]
\end{aligned}$$

$$= t^*[\![t_0t_1]\!] \wedge t^*[\![t_0t_2]\!]$$
$$= \lambda m.t^*[\![t_0t_1]\!](m) \wedge t^*[\![t_0t_2]\!](m)$$
$$= \lambda m.(t[\![t_0]\!]^\circ t[\![t_1]\!])(m) \wedge (t[\![t_0]\!]^\circ t[\![t_2]\!])(m)$$
$$= \lambda m.t[\![t_0]\!](t[\![t_1]\!](m)) \wedge t[\![t_0]\!](t[\![t_2]\!](m))$$

Unless all the semantic functions are *distributive*, the above functions are not equal in general. Kam and Ullman demonstrate this in the context of intraprocedural analysis on flow graphs [44]. For cases when the semantic functions are monotonic, but not distributive, they suggest an algorithm for increasing the precision of the analysis. When modeled as sets of execution paths, as in this dissertation, their algorithm may be viewed as decomposing the path sets in a more complicated manner than the obvious one suggested by the set equations for the paths. As shown in subsequent sections, the use of function composition and pointwise meet for $\cup$ and $\bullet$, respectively, is the right idea; it is, however, necessary to show how they may be used to yield a computable correct approximation of $\mathcal{T}_{T,t}$.

IV.6.7 Integrating intra- and interprocedural control flow

Representing possible executions as sets of execution paths adds a key advantage when compared with other representation mechanisms: intra- and interprocedural control flow may be integrated into a single representation. Figure 14 shows a recursive version of the running example.

Representing this program as a family of set equations yields:

$$S_1 = \{\varepsilon\} \qquad S_6 = \{t_{6/6}\} \bullet S_6 \bullet \{t_{5/1}\} \bullet S_5 \cup \{t_{2/6}\} \bullet S_2$$
$$S_2 = \{t_1\} \bullet S_1 \qquad S_7 = \{\varepsilon\}$$
$$S_3 = \{t_{2/3}\} \bullet S_2 \qquad S_8 = \{t_7\} \bullet S_7$$
$$S_4 = \{t_3\} \bullet S_3 \qquad S_9 = \{t_8\} \bullet S_8$$
$$S_5 = \{t_4\} \bullet S_4 \qquad S_{10} = \{t_{6/10}\} \bullet S_6 \bullet \{t_{9/1}\} \bullet S_9$$
$$S_{11} = \{t_{10}\} \bullet S_{10}$$

Note that the procedure calls differ from intraprocedural flow only in that a call involves the concatentation of more than two components. Consider the call between points 9 and 10, represented by the equation:

$$S_{10} = \{t_{6/10}\} \bullet S_6 \bullet \{t_{9/1}\} \bullet S_9$$

This may be read as: the set of all execution paths from the beginning of the program (point 7) to point 10 consists of all the paths from point 7 to point 9 (S_9), preceded by entry to the procedure ($\{t_{9/1}\}$), preceded by all the paths representing possible executions of the procedure (S_6), and, finally, preceded by exit from the procedure ($\{t_{6/10}\}$). In terms of FGs, the only difference between intraprocedural and interprocedural control flow is that the former are regular and the latter are context free.

IV.6.8 Semantics of flow grammars

Suppose one had a FG $G = \langle V, T, P, S \rangle$ such that $\mathcal{L}(G) = E$, for some set, E, of execution paths in a program. Further, suppose one also had a mapping $t: T \rightarrow F$ where T is the set of flow terminals in G and F is a monotonic function space over a lattice L. Then one might be able to devise a computable function, say $\mathcal{G}_{T,t}$, such that:

$$\mathcal{G}_{T,t}(\langle V, T, P, S \rangle) \leq \mathcal{T}_{T,t}[\![S]\!] \tag{7}$$

Then $\mathcal{G}_{T,t}$ is a correct, conservative, approximation of $\mathcal{T}_{T,t}$ and may be used to support optimization. In order to show that such a G does, indeed, exist for both regular and context-free FGs, the following definitions and lemmas are necessary. The proof strategy used below for interprocedural analysis using monotonic (as opposed to distributive) functions is, to the author's knowledge, new. In particular, using monotonicity on two levels is novel in the context of the static analysis of computer programs.

Figure 14 Recursive factorial illustrating interprocedural control flow

Point	Program
	`procedure fact;`
1	`begin`
2	`  if i > 1 then begin`
3	`    f := f * i;`
4	`    i := i - 1;`
5	`    fact`
	`  end`
6	`end;`
7	`i := 5;`
8	`f := 1;`
9	`fact;`
10	`write(f);`
11	

Definition 40: Let L be a meet semi-lattice.

> $F(L)=L\rightarrow L$ is the *total function flow analysis lattice (TFL)* containing all the total functions from L to L under the ordering $\leq_{F(L)}$ defined as follows: for $f,g\in F(L)$, $f \leq_{F(L)} g$ **iff** $f(x)\leq_L g(x), \forall x\in L$. In this case, f is said to *approximate* g.

L is said to be the *lower,* or *intraprocedural, lattice*, and F(L) is said to be the *upper,* or *interprocedural, lattice*.

Observation: The corresponding meet operator is: $f \wedge_{F(L)} g = \lambda x.f(x) \wedge_L g(x)$.

Proof: To show: $f \wedge_{F(L)} g = f$ **iff** $f \leq_{F(L)} g$

> **Only-if:** Assume $f \wedge_{F(L)} g = f$. Then $f(x) \wedge_L g(x) = f(x)$, $\forall x \in L$ (def'n. of $\wedge_{F(L)}$). This implies $f(x) \leq_L g(x)$, $\forall x \in L$ (since $\wedge_L$ is the meet of L and $\leq_L$ is the partial order on L).
>
> **If:** Assume $f \leq_{F(L)} g$. Then $f(x) \leq_L g(x)$, $\forall x \in L$ (def'n. of $\leq_{F(L)}$). This implies $f(x) \wedge_L g(x) = f(x)$, $\forall x \in L$ (since $\wedge_L$ is the meet of L and $\leq_L$ is the partial order on L).

Note that $T_{F(L)} = \lambda x.T_L$ and $\perp_{F(L)} = \lambda x.\perp_L$.

Intuitively, a function f approximates g if f is not more precise on any particular input than g. Computing the meet of two functions in F(L) yields the most precise function that approximates both functions. Thus, this construction may be used to lift an intraprocedural lattice to an interprocedural lattice.

Observation: $height(F(L)) = |L| \cdot height(L)$

Proof: The longest possible chain of functions starts with the map $T_{F(L)}$. Now, for each element in L there can be at most height(L) different functions. Thus, in the worst case, each of the |L| elements contributes a sub-chain of height(L). Thus the total length of the longest possible chain is the combined length of the sub-chains, $|L| \cdot height(L)$.

The following fundamental lemma is well-known [44] and is given because of its fundamental role in the development.

Lemma 1: The composition of two monotonic functions f and g, $f \circ g$, over a lattice L is also monotonic. That is, $\forall x,y \in L$, if $x \leq_L y$ then $f \circ g(x) \leq_L f \circ g(y)$.

Proof: Let x, y be arbitrary members of L. If $x \leq_L y$ then $g(x) \leq_L g(y)$ (g is monotonic). Given that $g(x) \leq_L g(y)$, $f(g(x)) \leq_L f(g(y))$ (f is monotonic). That is, $f \circ g(x) \leq_L f \circ g(v)$.

The following lemmas extend commonly known facts about monotonic functions into the realm of total function lattices and are needed in the correctness proof.

Lemma 2: The pointwise meet of two monotonic functions f and g, over a lattice L, denoted $f \wedge_{F(L)} g$, is also monotonic. That is, $\forall x,y \in L$, if $x \leq_L y$ then $(f \wedge_{F(L)} g)(x) \leq_L (f \wedge_{F(L)} g)(y)$.

Proof: Let x, y be arbitrary members of L. If $x \leq_L y$ then $f(x) \leq_L f(y)$ (f is monotonic) and $g(x) \leq_L g(y)$ (g is monotonic). Thus, $f(x) = f(x) \wedge_L f(y)$ and $g(x) = g(x) \wedge_L g(y)$. So, $f(x) \wedge_L g(x) = (f(x) \wedge_L f(y)) \wedge_L (g(x) \wedge_L g(y)) = (f(x) \wedge_L g(x)) \wedge_L (f(y) \wedge_L g(y))$ and, therefore, $f(x) \wedge_L g(x) \leq_L f(y) \wedge_L g(y)$. That is, $(f \wedge_{F(L)} g)(x) \leq (f \wedge_{F(L)} g)(y)$.

Lemma 3: Given three monotonic functions f, g, and h, over a lattice L,

$$f \circ (g \wedge_{F(L)} h) \leq_{F(L)} (f \circ g) \wedge_{F(L)} (f \circ h)$$

that is,

$$\forall x \in L.((f \circ (g \wedge_{F(L)} h))(x) \leq_{F(L)} (f \circ g) \wedge_{F(L)} (f \circ h))(x)$$

Proof: Let x be an abritrary member of L. Then

$$\begin{aligned}(f \circ (g \wedge_{F(L)} h))(x) &= f((g \wedge_{F(L)} h)(x)) \\ &= f(g(x) \wedge_L h(x))\end{aligned}$$

and

$$\begin{aligned}((f \circ g) \wedge_{F(L)} (f \circ h))(x) &= (f \circ g)(x) \wedge_L (f \circ h)(x)) \\ &= f(g(x)) \wedge_L f(h(x))\end{aligned}$$

Now, $g(x) \wedge_L h(x) \leq_L g(x)$ and $g(x) \wedge_L h(x) \leq_L h(x)$, so, by the monotonicity of f:

$$f(g(x) \wedge_L h(x)) \leq_L f(g(x))$$
$$f(g(x) \wedge_L h(x)) \leq_L f(h(x))$$

Thus, $f(g(x) \wedge_L h(x)) \leq_L f(g(x)) \wedge_L f(h(x))$.

Lemma 4: If f_1, f_2, g_1, g_2 are monotonic functions over a lattice L such that $f_1 \leq_{F(L)} g_1$ and $f_2 \leq_{F(L)} g_2$ then $f_1 \circ f_2 \leq_{F(L)} g_1 \circ g_2$. That is, $\forall x \in L, (f_1 \circ f_2)(x) \leq_L (g_1 \circ g_2)(x)$.

Proof: Let $x \in L$ be a member of the lattice. Then $f_2(x) \leq_L g_2(x)$ (because $f_2 \leq_{F(L)} g_2$) and then $f_1(f_2(x)) \leq_{F(L)} f_1(g_2(x))$ (because f_1 is monotonic) $\leq_{F(L)} g_1(g_2(x))$ (because $f_1 \leq_{F(L)} g_1$). Therefore, $f_1(f_2(x)) \leq_{F(L)} g_1(g_2(x))$, as required.

Semantics of a flow grammar

Returning to FGs, we now derive a system of equations from a given FG $G = \langle V,T,P,S \rangle$ where there is a variable for each $N \in V$ and show two things:

1. Solving the equations using fixed point iteration yields a solution in which the values computed for all variables is a monotonic function.
2. The solution for each $N \in V$ approximates $\mathcal{T}_{T,t}[\![\mathcal{L}(\langle V,T,P,N \rangle)]\!]$.

Definition 41: Let $G = \langle V,T,P,S \rangle$ be a FG, L be a finite lattice, F be a monotonic function space on L and $t : T \rightarrow F$. The *equations corresponding to* G *with respect to* t, consist of the equations:

$$\mathcal{K}_N = \wedge_{F(L)} \{ \mathcal{P}(r) \mid \text{“}N ::= r\text{”} \in P \}$$

where

$$\mathcal{P} : (T \cup V)^* \rightarrow (L \rightarrow L)$$
$$\mathcal{P}(\varepsilon) = \lambda s.s$$
$$\mathcal{P}(\alpha t) = \mathcal{P}(\alpha) \circ t[\![t]\!], t \in T, \alpha \in (T \cup V)^*$$
$$\mathcal{P}(\alpha N) = \mathcal{P}(\alpha) \circ \mathcal{K}_N, N \in V, \alpha \in (T \cup V)^*$$

and

$\mathcal{K}_N$ are variables corresponding to the FNs in G

Note that “with respect to t” is often omitted when t is clear from context.

As an example, consider the FG G of the running intraprocedural example (Figure 10):

$$S_1 ::= \varepsilon, \qquad S_5 ::= t_4\ S_4,$$
$$S_2 ::= t_1\ S_1, \qquad S_6 ::= t_5\ S_5,$$
$$S_3 ::= t_2\ S_2 \mid t_6\ S_6, \qquad S_7 ::= t_{3/7}\ S_3,$$
$$S_4 ::= t_{3/4}\ S_3, \qquad S_8 ::= t_7\ S_7$$

The equations corresponding to G are:

$$\mathcal{K}_{S1} = \lambda s.s \qquad \mathcal{K}_{S5} = t[\![t_4]\!] \circ \mathcal{K}_{S4}$$
$$\mathcal{K}_{S2} = t[\![t_1]\!] \circ \mathcal{K}_{S1} \qquad \mathcal{K}_{S6} = t[\![t_5]\!] \circ \mathcal{K}_{S5}$$
$$\mathcal{K}_{S3} = t[\![t_2]\!] \circ \mathcal{K}_{S2} \wedge_{F(L)} t[\![t_6]\!] \circ \mathcal{K}_{S6} \qquad \mathcal{K}_{S7} = t[\![t_{3/7}]\!] \circ \mathcal{K}_{S3}$$
$$\mathcal{K}_{S4} = t[\![t_{3/4}]\!] \circ \mathcal{K}_{S3} \qquad \mathcal{K}_{S8} = t[\![t_7]\!] \circ \mathcal{K}_{S7}$$

where $\mathcal{K}_{Si}$ corresponds to S_i in G.

In order to solve such equations, it is convenient to consider the $\mathcal{K}_i$'s as a vector and re-write the equations using a function on vectors. The preceding equations are equivalent to the following:

$$\langle \mathcal{K}_{S1}, \mathcal{K}_{S2}, \ldots, \mathcal{K}_{S8} \rangle = f(\langle \mathcal{K}_{S1}, \mathcal{K}_{S2}, \ldots, \mathcal{K}_{S8} \rangle)$$

where

$$f(\langle X_1, X_2, \ldots, X_n \rangle) = \langle\ f_1(X_1, X_2, \ldots, X_n),\ f_2(X_1, X_2, \ldots, X_n),\ \ldots,\ f_8(X_1, X_2, \ldots, X_n) \rangle$$

and

$$f_1(X_1, X_2, \ldots, X_8) = \lambda s.s$$
$$f_2(X_1, X_2, \ldots, X_8) = t[\![t_1]\!] \circ X_1$$
$$\ldots$$
$$f_8(X_1, X_2, \ldots, X_8) = t[\![t_7]\!] \circ X_7$$

We note the following three facts:

Lemma 5: There exists a FG G = ‹V,T,P,S›, a finite lattice L, a monotonic function space F on L and $t: T \rightarrow F$ such that the equations corresponding to G have a fixed point solution whose elements are not all monotonic functions.

Proof:

Let G = ‹ {S}, {t}, { "S ::= ε", "S ::= S t_1"}, S›, L = { ⊥, 1, ⊤} where $\perp \leq_L 1 \leq_L \top$ and F consists of the closure under composition of the identity function along with the constant function for each element in L. Finally, let $t[\![t]\!] = \lambda s.s$. Now, the equation corresponding to G in this instance is:

$$\mathcal{K}_S = ((\lambda s.s) \circ \mathcal{K}_S) \wedge_{F(L)} \lambda s.s$$

which is equivalent to:

$$\mathcal{K}_S = \mathcal{K}_S \wedge_{F(L)} \lambda s.s$$

Now, the following function g is a fixed point of this equation, but is not monotonic.

v	g(v)
⊤	⊥
1	1
⊥	⊥

Theorem 9 below, however, requires that all functions computed in the fixed point are monotonic on the lower lattice L. This necessitates the following results which ensure that the fixed point found using standard iterative techniques yield only monotonic functions.

Lemma 6: When all arguments of each f_i are monotonic in L, the resulting function is also a monotonic function in L.

Proof: This follows from Lemmas 1 and 2 combined with the fact that each f_i is defined in terms of the parameters (assumed to be monotonic) and monotonic functions $t[\![t_j]\!]$ using the operators $\wedge_{F(L)}$ and $\circ$.

Lemma 7: Each f_i is itself monotonic in F(L) in all its arguments when applied to functions that are monotonic in L.

Proof:

Suppose $Y, Z \in F(L)$, $Y \leq_{F(L)} Z$, and Y and Z are monotonic in L. Consider $f_i(Y, Y_2, \ldots, Y_n)$ and $f_i(Z, Y_2, \ldots, Y_n)$ where $Y_2,\ldots,Y_n$ are monotonic in L. Now, consider the right hand side of the definition for f_i, which is of the form:

$$C_1 \wedge_{F(L)} C_2 \wedge_{F(L)} \cdots \wedge_{F(L)} C_m$$

where each C_j is the composition of monotonic functions and parameters.

Now consider the value of term C_j in $f_i(Y, Y_2, \ldots, Y_n)$ (call it C_{jY}) and $f_i(Z, Y_2, \ldots, Y_n)$ (call it C_{jZ}). Because C_j is the composition of n functions, $b_p \circ \ldots \circ b_1$, we can consider corresponding functions $b_{pY},\ldots,b_{1Y}$ in C_{jY} and $b_{pZ},\ldots,b_{1Z}$ in C_{jZ}. We show by induction that $C_{jY} \leq_{F(L)} C_{jZ}$:

Basis:

First, either b_{1Y} and b_{1Z} are the same function or b_{1Y} is Y and b_{1Z} is Z; in both cases, $b_{1Y} \leq_{F(L)} b_{1Z}$.

Induction:

Now suppose $b_{iY} \circ \ldots \circ b_{1Y} \leq_{F(L)} b_{iZ} \circ \ldots \circ b_{1Z}$. Consider $b_{(i+1)Y} \circ b_{iY} \ldots \circ b_{1Y}$ and $b_{(i+1)Z} \circ b_{iZ} \ldots \circ b_{1Z}$: either $b_{(i+1)Y} = b_{(i+1)Z}$, in which case:

$$b_{(i+1)Y} \circ b_{iY} \ldots \circ b_{1Y} \leq_{F(L)} b_{(i+1)Z} \circ b_{iZ} \ldots \circ b_{1Z} \text{ by Lemma 4}$$

or $b_{(i+1)Y} = Y$ and $b_{(i+1)Z} = Z$, in which case

$$b_{(i+1)Y} \circ b_{iY} \ldots \circ b_{1Y} \leq_{F(L)} b_{(i+1)Z} \circ b_{iZ} \ldots \circ b_{1Z}, \text{ again by Lemma 4}$$

Therefore $C_{jY} \leq_{F(L)} C_{jZ}$ for all j. Furthermore,

$$C_{1Y} \wedge_{F(L)} \cdots \wedge_{F(L)} C_{mY} \leq_{F(L)} C_{1Z} \wedge_{F(L)} \cdots \wedge_{F(L)} C_{mZ}$$

and so $f_i(Y, Y_2, \ldots, Y_n) \leq_{F(L)} f_i(Z, Y_2, \ldots, Y_n)$, making f_i monotonic in its first argument.

A similar proof may be given for all the arguments of f_i, completing the proof.

Thus, we have monotonicity on two levels: each f_i produces a monotonic function (on L) when given monotonic functions as input and is itself monotonic on F(L) when given monotonic functions as input. If we consider the vector of variables to be a cross-product lattice Ln where

$$\langle a_1,\ldots,a_n\rangle \leq_{Ln} \langle b_1,\ldots,b_n\rangle \text{ iff } a_i \leq_L b_i,\ 1 \leq i \leq n$$

the above equation becomes:

$$\mathcal{K} = f(\mathcal{K})$$

where f is a monotonic function (because all the f_i are monotonic in F(L)).

Now the goal is to find the greatest fixed point (GFP)[1] of this equation and then show that it approximates $\mathcal{T}_{T,t}[\![L(G)]\!]$. To find the GFP solution, we start with:

$$\top_{Ln} = \langle \top_{F(L)}, \top_{F(L)}, \ldots, \top_{F(L)} \rangle$$

and repeatedly apply f. Let f^i denote the i'th iterate of f (i.e., $f^0 = \lambda s.s$, $f^i = f \circ f^{i-1}$, $i \geq 1$).

1. Once again, it should be noted that we are discussing a meet semi-lattice, so must start at the top and work our way down to a greatest fixed point.

Theorem 8: $f^{i+1}(\top_{Ln}) \leq_{F(L)} f^{i}(\top_{Ln})$, $\forall i \geq 0$. Furthermore, all components of $f^{i}(\top_{Ln})$ are monotonic in L.

Proof: By induction.

Basis: Show $f^{1}(\top_{Ln}) \leq_{F(L)} f^{0}(\top_{Ln})$.

$f^{0}(\top_{Ln}) = \top_{Ln}$. Now, each component of $\top_{Ln}$ is $\top_{F(L)}$, a monotonic function on L. Thus, $f(\top_{Ln})$ is a lattice element in Ln each of whose components are monotonic (by Lemma 6), and $f^{1}(\top_{Ln}) = f(\top_{Ln}) \leq_{Ln} \top_{Ln} = f^{0}(\top_{Ln})$ (by Lemma 7).

Induction: Assuming $f^{i}(\top_{Ln}) \leq_{F(L)} f^{i-1}(\top_{Ln})$ and all components of $f^{i}(\top_{Ln})$ are monotonic, must show $f^{i+1}(\top_{Ln}) \leq_{F(L)} f^{i}(\top_{Ln})$ and all components of $f^{i+1}(\top_{Ln})$ are montonic.

By assumption, each component of $f^{i}(\top_{Ln})$ is monotonic in L. Thus, each component of $f(f^{i}(\top_{Ln}))$ is monotonic in L. Then, by Lemma 7, $f(f^{i}(\top_{Ln})) \leq_{F(L)} f^{i}(\top_{Ln})$.

Thus, we have a chain in of decreasing values in Ln. Because L is a finite lattice, this chain is guaranteed to result in a fixed point and all the components of this fixed point are monotonic in L.

Given that the equations corresponding to a FG G =‹V,T,P,S› have a GFP solution for a given t, the next task is to show that this solution approximates $\mathcal{F}_{T,t}[\![\mathcal{L}(G)]\!]$. First, define $\mathcal{G}_{T,t}: G_T \rightarrow F$, where G_T is the set of all grammars with terminals T, as follows:

$\mathcal{G}_{T,t}(\langle V,T,P,S\rangle) = \mathcal{V}_S$
where
$\mathcal{V}_S$ is the value for $\mathcal{X}_S$ in the greatest fixed point solution to the equations corresponding to ‹V,T,P,S›

Theorem 9: If T is a set of flow terminals, $t: T \rightarrow F$ where F is a monotonic function space over a finite lattice L and G = ‹V,T,P,S› is a FG, then $\mathcal{G}_{T,t}(G) \leq_{F(L)} \mathcal{F}_{T,t}[\![\mathcal{L}(G)]\!]$.

Proof: Let $\mathcal{L}(G) = \{ \alpha_1, \alpha_2, \ldots \}$. Then $\mathcal{F}_{T,t}[\![\mathcal{L}(G)]\!] = t^*(\alpha_1) \wedge_{F(L)} t^*(\alpha_2) \wedge_{F(L)} \ldots$ The goal is to show $\mathcal{G}_{T,t}(\langle V, T, P, S\rangle) \leq_{F(L)} t^*(\alpha_j)$, for all α_j. Consider any α_j. Because $\alpha_j \in \mathcal{L}(G)$, there exists a derivation of the form:

$$S = a_0 \models a_1 \models a_2 \models \ldots \models a_n = \alpha_j$$

Let $M : (V \cup T)^* \rightarrow F$ be defined as follows:

$$M(\varepsilon) = \lambda s.s$$
$$M(\alpha t) = M(\alpha) \circ t(t)$$
$$M(\alpha N) = M(\alpha) \circ \mathcal{G}_{T,t}(\langle V, T, P, N\rangle)$$

Note that $M(\alpha_j) = t^*(\alpha_j)$. Now we show that $M(a_i) \leq M(a_{i+1})$, $0 \leq i \leq n\text{-}1$, from which the result follows.

Consider any pair (a_m, a_{m+1}). Because $a_m \models a_{m+1}$, there exist strings β, γ and δ and a nonterminal N such that:

$$a_m = \beta N \gamma$$
$$a_{m+1} = \beta\delta\gamma \qquad \text{and } \text{“}N ::= \delta\text{”} \in P$$

Now, $M(a_m) = M(\beta) \circ \mathcal{G}_{T,t}(\langle V, T, P, N\rangle) \circ M(\gamma)$ and $M(a_{m+1}) = M(\beta) \circ M(\delta) \circ M(\gamma)$. By Definition 41, $\mathcal{G}_{T,t}(\langle V, T, P, N\rangle) \leq_{F(L)} M(\delta)$, and by Theorem 8 $M(\beta)$ and $M(\gamma)$ are monotonic, therefore:

$$M(\beta) \circ \mathcal{G}_{T,t}(\langle V, T, P, N\rangle) \circ M(\gamma) \leq_{F(L)} M(\beta) \circ M(\delta) \circ M(\gamma)$$

and thus, $M(a_m) \leq M(a_{m+1})$, as required.

IV.6.9 On the form of flow grammars

Thus far, productions in FGs are simply context free grammars that happen to match the control flow in a program. For the purposes of data flow analysis, however, it is useful to restrict the form of productions:

> If $G = \langle V,T,P,S\rangle$ is a flow grammar, then each production is of the form $N ::= \alpha$ where $N \in V$ and $\alpha \in (\{\varepsilon\} \cup (V \cup T)^* \bullet V)$. That is, each production either generates the empty string or generates a string of one or more symbols, the last of which must be a nonterminal.

Note that all the examples of grammars thus far already have this form. It is perhaps worth pointing out that the above constraint does not change the set of languages that can be generated: it is straightforward to convert any context free grammar to this form.

Lemma 10: Any grammar G= ‹ V, T, P, S › may be converted to an equivalent grammar G′ = ‹ V′, T, P′, S › such that $\mathcal{L}(G) = \mathcal{L}(G')$ and no production in P′ has a terminal as its rightmost symbol.

Proof: Let $P_1 = \{\text{“}l_1::=r_1\text{”},\ldots, \text{“}l_n::=r_n\text{”}\} \subseteq P$ be the subset of productions in P whose rhs end in a terminal. Let $V_1 = \{N_1,\ldots, N_n\}$ be a set of new nonterminals (i.e., such that $V \cap V_1 = \varnothing$) and let $P' = (P - P_1) \cup \{\text{“}l_1::=r_1N_1\text{”},\ldots, \text{“}l_n::=r_nN_n\text{”}, \text{“}N_1 ::= \varepsilon\text{”},\ldots,\text{“}N_n ::= \varepsilon\text{”}\}$. Finally, $G' = ‹ V \cup V_1, T, P', S ›$. Clearly, G′ satisfies the constraint: all productions ending in a terminal have been removed and the new productions are all of the proper form. That $\mathcal{L}(G) = \mathcal{L}(G')$ is obvious: all nonterminals in V still generate exactly the same strings they did before.

Forward vs. backward flow grammars

Flow grammars may be used to represent both forward and backward flow information. The examples so far have all been oriented towards the forward flow of information. Each nonterminal corresponds to a point P in some program unit and generates strings representing all possible executions from the beginning of the unit to point P. Thus the equations corresponding to such a *forward flow grammar* summarize the forward flow of information.

Alternatively, one can consider the set of possible executions from a given point to the end of the point's program unit. In this case, the solving the equations corresponding to

a *backward flow grammar* yield a summary of the backward flow of information. For the running example program, the backward flow grammar is shown below.

$$
\begin{array}{ll}
S_1 ::= t_1\ S_2 & S_5 ::= t_5\ S_6 \\
S_2 ::= t_2\ S_3 & S_6 ::= t_6\ S_3 \\
S_3 ::= t_{3/4}\ S_4 \mid t_{3/7}\ S_7 & S_7 ::= t_7\ S_8 \\
S_4 ::= t_4\ S_5 & S_8 ::= \varepsilon
\end{array}
$$

This grammar would be suitable for solving the live variables problem in which information flows backward from the end of the program to the beginning.

Fortunately, given a forward FG, it is possible to convert it to a backward FG and vice versa. For intraprocedural FGs (that is regular FGs), the transformation is straightforward:

1. all the productions of the form "$X ::= \alpha Y$" become "$Y ::= \alpha^R X$",[1]
2. for all productions of the form "$X ::= \varepsilon$", add a production of the form "$S' ::= X$" and delete "$X ::= \varepsilon$",
3. if S is the start symbol of the original grammar, add a production of the form: "$S ::= \varepsilon$".
4. S' is the start symbol of the new grammar.

This transformation is equivalent to the transformation on finite state machines for deriving a finite state machine accepting the reverse of a language accepted by a given finite state machine. Note that deriving a set of equations corresponding to a grammar (cf. Definition 41) may implicitly perform this reversal, thus eliminating the need for maintaining a forward and backward FG for intraprocedural analysis.

1. α^R denotes the reverse of α.

Note the special nature of productions generating the empty string: in a forward FG they represent the start of execution of some program unit; in a backward FG they represent the end of execution of some program unit. This motivates the following definition of the "right-reaches" relation which may be used to find the entry/exit points of program units.

Definition 42: The *right-reaches* relation is defined on the set of non-terminals in a FG as follows:

$Z \mathcal{R} Z$ if $Z ::= \varepsilon$
$X \mathcal{R} Z$ if $X ::= \alpha Y$ and $Y \mathcal{R} Z$

That is, for a given nonterminal N, N right-reaches all nonterminals M that may represent the "end" of N's program unit.

Transforming an interprocedural FG from forward to backward, or vice versa, uses $\mathcal{R}$ to determine the appropriate re-writings of nonterminals embedded on the rhs of productions. The transformation of a FG $G = \langle V,T,P,S \rangle$ is as follows:

1. Productions of the form "$X ::= \alpha Y$" become "$Y ::= \beta X$" for all $\beta \in rev(\alpha)$, where rev is defined as follows:

   $rev(\varepsilon) = \{ \varepsilon \}$
   $rev(N) = \{ \varepsilon \}, N \in V$ (note that all nonempty productions have a nonterminal as their rightmost symbol)
   $rev(t\gamma) = rev(\gamma) \bullet \{ t \}, t \in T, \gamma \in (V \cup T)^+$
   $rev(N\gamma) = rev(\gamma) \bullet \{ Z \mid N \mathcal{R} Z \}, N \in V, \gamma \in (V \cup T)^+$
2. Delete all the productions of the form "$X ::= \varepsilon$".
3. For all N found in "$rev(N\gamma)$" of step **1**, add a production of the form: "$N ::= \varepsilon$".
4. Add a production of the form "$S ::= \varepsilon$"
5. For all nonterminals N such that $S \mathcal{R} X$, add a production of the form "$S' ::= X$" and delete "$X ::= \varepsilon$".
6. S' is the new start symbol.

This transformation works for programs that have only single entry point procedures. In the case of languages like PL/I where procedures can have multiple entry points, the transformation does not work as shown by applying the transformation to the following grammar:

$S_1 ::= \varepsilon$ $\quad\quad$ $X_1 ::= \varepsilon$
$S_2 ::= X_2\ S_1$ $\quad\quad$ $X_2 ::= t_1\ X_1$
$S_3 ::= X_3\ S_2$ $\quad\quad$ $X_3 ::= t_2\ X_2$

where S_3 is the start symbol. In this case, there is a "procedure" represented by the productions for X_1, X_2 and X_3 that has two entry points, X_2 and X_3. Applying the interprocedural transformation results in the following grammar:

$S_1 ::= X_1\ S_2$ $\quad\quad$ $X_1 ::= t_1\ X_2$
$S_2 ::= X_1\ S_3$ $\quad\quad$ $X_2 ::= t_2\ X_3$
$S_3 ::= \varepsilon$ $\quad\quad$ $X_3 ::= \varepsilon$

where S_1 is the start symbol (note that there is only one entry point to the program, so it is unnecessary to introduce a new start symbol). Note that this grammar does not distinguish between the two entry points, even though the original grammar did. This problem may be remedied by treating each entry point as a separate procedure and then combining the results of corresponding points after data flow analysis to yield a final solution. Thus, it is assumed throughout the remainder of the dissertation that all procedures are single entry. Moreover, using the above transformations on flow grammars, either directly or indirectly via the construction of flow equations, allows us to use either a forward or backward flow grammar, as appropriate to the circumstances.

IV.7 Flow grammars and CPS semantics

FGs bear a strong resemblance to *continuation passing style (CPS) semantics* [82]. This is not surprising, as both formalisms share the goal of describing programs. The difference is in intent and, as shown in subsequent chapters, use. On one hand, CPS semantics address the mathematical meaning of programs that permit arbitrary control flow. On the other hand, FGs have been developed to support the FACT tool, and to that end must address user interface issues. Chapter III motivated the development of FGs by showing how it is possible to extend the traditional control flow graph in a manner that combines intuition with ease of specification. The remainder of this section explores the relationship between the two formalisms.

IV.7.1 Semantic validity

As discussed in Chapter II, various forms of graph have been used extensively to represent programs during data flow analysis. One question seldom addressed in the literature, however, is the relationship between these graphs and the denotational semantics of the language being analyzed. Flemming Nielson did some work in the early 1980s in which he devised the notion of a *collecting semantics* [65]. This is a denotational description based on a store semantics [82] which associates a "left" and "right" point with each node in the abstract syntax tree of a program. The meaning of the program is a map from these points to the set of states the program could be in when control reaches that point. He also gives a semantic function relating the collecting semantics to the usual control flow graph, but in the absence of goto statements and procedure calls. In a subsequent paper [66], he extends the idea to permit reasoning about valid transformations by replacing one node of the AST with an equivalent node (i.e., for all states on the "left" of the original node, the new node behaves the same). He also introduces a *future semantics* based on CPS intended to repre-

sent backward flow of information. Of particular relevance to the current work are properties relating the sets of states at various points within a construct: these properties are analogous to the "flow pictures" of the preceding chapter. Although he uses CPS, his techniques do not support non-local control flow, such as goto statements. Nor have the techniques been applied to interprocedural analysis.

IV.7.2 Another approach: partial evaluation of CPS

If the graph and grammar control flow models are in fact correct, then it should be possible to deduce them from a denotational semantic description of the source language. The approach taken here is to perform partial evaluation on a store semantics based on CPS to yield a FG. The following example shows how it is possible to relate the denotational semantics to a flow graph or grammar.

IV.7.3 Example: a 'toy' language and its semantics

Figure 15 gives the syntax for a small example language discussed in [66] to which labels and goto statements have been added. From the standpoint of a compiler writer, the grammar shown is a cross between the concrete and abstract syntax of the language where `cmd` and `exp` are non-terminals and `lab`, `ope`, `ide` and `bas` are terminals whose values are assigned by the lexer. In terms of denotational semantics, we have:

1. `cmd` ∈ Cmd (the syntactic category of *commands*)
2. `exp` ∈ Exp (the syntactic category of *expressions*)
3. `lab` ∈ Lab (the syntactic category of *labels*)
4. `ide` ∈ Ide (the syntactic category of *identifiers*)
5. `ope` ∈ Ope (the syntactic category of *operators*)
6. `bas` ∈ Bas (the syntactic category of *basic values*)

For convenience, let Syn = Cmd ∪ Exp. The syntax of labels, identifiers, operators and basic values is left unspecified, as are the semantic functions $\mathcal{O}$ and $\mathcal{B}$ for operators and basic values, respectively.

Figures 16, 17, 18, 19, 20 and 21 define a *CPS store semantics* of this language. The semantic functional, $\mathcal{T}$, is given in Figure 18. Its type is:

Syn -c> (Lab -c> C) -c> C -c> C

That is, $\mathcal{T}$ is a curried function that takes the following arguments:

1. A syntactic construct (Syn, a command or expression).
2. A "label environment" ((Lab -c> C)) mapping labels to continuations (all labels $\texttt{lab}_i$ are assumed to be distinct).
3. A continuation (C) representing the effect of executing the "rest" of the program.

The continuation itself is a map from states (Sta in Figure 16) to "answers" (A in Figure 21). A state is composed of four components:

1. Env: an "environment" mapping identifiers to values (Val).
2. Inp: a list of input values.
3. Out: a list of output values.
4. Tem: a "stack" of temporary values.

States are manipulated by the various functions labeled "body" in the figures.

Example

As a simple example, consider evaluating the command:

```
read i; write i
```

with the input "1."

The initial label environment is the empty function, g = λlab.⊥, and the initial continuation is fin, which simply returns the output. Thus, we wish to evaluate:

$$\mathcal{T}\,[\![\texttt{read i; write i}]\!]\,g\,fin$$

in initial state "init," defined below.

Figure 15 Syntax of a toy imperative language

```
cmd ::= 'begin'
            lab1:cmd1 ';'
            lab2:cmd2 ';'
            ...
            labn:cmdn
        'end' |
        'skip' |
         cmd ';' cmd | ide ':=' exp |
        'if' exp 'then' cmd 'else' cmd 'fi' |
        'while' exp 'do' cmd 'od' |
        'write' exp | 'read' exp |
        'goto' lab
exp ::= exp ope exp | ide | bas
```

Figure 16 Semantic domains

Domains			
T			Booleans
N			integers
Val	=	N + T	values
Env	=	Ide -c> Val	environments
Inp	=	Val*	inputs
Out	=	Val*	outputs
Tem	=	Val*	temporary result stacks
Sta	=	Env × Inp × Out × Tem	states

Figure 17 Auxiliary functions for the conditional

Vcond ∈ Sta -c> T	"verify"
Vcond = λ‹env,inp,out,tem› . #tem ≥≥ 1 → tem ↓ 1 ∈ T , false	
Scond ∈ Sta -c> T	"select"
Scond = λ‹env,inp,out,tem› . (tem ↓ 1) | T	
Bcond ∈ Sta -c> Sta	"body"
Bcond = λ‹env,inp,out,tem› . ‹env,inp,out,tem † 1›	

Simplifying the semantic functional:

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{read i; write i}]\!]\ g\ fin\\
&= \mathcal{T}[\![\texttt{read i}]\!]\ g\ \{\ \mathcal{T}[\![\texttt{write i}]\!]\ g\ \{\ fin\ \}\ \}\\
&= \mathcal{T}[\![\texttt{read i}]\!]\ g\ \{\ \mathcal{T}[\![\texttt{write i}]\!]\ g\ \{\ fin\ \}\ \}\\
&= \text{read}\ \{\ \text{assign}\ [\![\texttt{i}]\!]\ \{\ \mathcal{T}[\![\texttt{write i}]\!]\ g\ \{\ fin\ \}\ \}\ \}\\
&= \text{read}\ \{\ \text{assign}\ [\![\texttt{i}]\!]\ \{\ \mathcal{T}[\![\texttt{i}]\!]\ g\ \{\ \text{write}\ \{\ fin\ \}\ \}\ \}\ \}\\
&= \text{read}\ \{\ \text{assign}\ [\![\texttt{i}]\!]\ \{\ \text{content}\ [\![\texttt{i}]\!]\ \{\ \text{write}\ \{\ fin\ \}\ \}\ \}\ \}
\end{aligned}
$$

Now, we will assume the initial environment is empty, as is the initial output and initial stack. Thus, the initial state init = ‹ λi.⊥, ‹1›, ‹›, ‹› ›. Evaluating the above function at init:

read { assign [[i]] { content [[i]] { write { fin } } } } init
= do (Vread,Bread)
{ assign [[i]] { content [[i]] { write { fin } } } } init
...
= assign [[i]] { content [[i]] { write { fin } } } (Bread init)
= assign [[i]] { content [[i]] { write { fin } } } ‹ λi.⊥, ‹›, ‹›, ‹1› ›
...
= fin ‹ (λi.⊥)[1 / [[i]]], ‹›, ‹1›, ‹› ›
= ‹1›

Figure 18 Semantic function for the toy imperative language

$\mathcal{T} \in$ Syn -c> (Lab -c> C) -c> C -c> C

$\mathcal{T}[\![$ 'begin' lab_1 : cmd_1 ';' lab_2 : cmd_2 ';' ... lab_n : cmd_n 'end' $]\!]\ g\ c$
$= (\text{FIX}\ (\lambda\langle c_1, c_2, \ldots, c_n\rangle .$
$\langle \mathcal{T}[\![cmd_1]\!]\ g'\ \{\ c_2\ \}, \mathcal{T}[\![cmd_2]\!]\ g'\ \{\ c_3\ \}, \ldots, \mathcal{T}[\![cmd_n]\!]\ g'\ \{\ c\ \}\rangle)$
where $g' = g\ [\ c_1 / [\![lab_1]\!], c_2 / [\![lab_2]\!], \ldots, c_n / [\![lab_n]\!]\ ]) \downarrow 1$

$\mathcal{T}[\![$ skip $]\!]\ g\ c$
$= c$

$\mathcal{T}[\![cmd_1$ ';' $cmd_2]\!]\ g\ c$
$= \mathcal{T}[\![cmd_1]\!]\ g\ \{\ \mathcal{T}[\![cmd_2]\!]\ g\ \{\ c\ \}\ \}$

$\mathcal{T}[\![$ ide ':=' exp $]\!]\ g\ c$
$= \mathcal{T}[\![$ exp $]\!]\ g\ \{$ assign $[\![$ ide $]\!]\ \{\ c\ \}\ \}$

$\mathcal{T}[\![$ 'if' exp 'then' cmd_1 'else' cmd_2 'fi' $]\!]\ g\ c$
$= \mathcal{T}[\![exp]\!]\ g\ \{$ cond $(\mathcal{T}[\![cmd_1]\!]\ g\ \{\ c\ \}, \mathcal{T}[\![cmd_2]\!]\ g\ \{\ c\ \})\ \}$

$\mathcal{T}[\![$ 'while' exp 'do' cmd 'od' $]\!]\ g\ c$
$= \text{FIX}\ (\lambda c' . \mathcal{T}[\![$ exp $]\!]\ g\ \{$ cond $(\mathcal{T}[\![cmd_1]\!]\ g\ \{\ c'\ \}, c)\ \})$

$\mathcal{T}[\![$ 'write' exp $]\!]\ g\ c$
$= \mathcal{T}[\![$ exp $]\!]\ g\ \{$ write $\{\ c\ \}\ \}$

$\mathcal{T}[\![$ 'read' ide $]\!]\ g\ c$
$=$ read $\{$ assign $[\![$ ide $]\!]\ \{\ c\ \}\ \}$

$\mathcal{T}[\![$ 'goto' lab $]\!]\ g\ c$
$= g\ [\![$ lab $]\!]$

$\mathcal{T}[\![exp_1$ ope $exp_2]\!]\ g\ c$
$= \mathcal{T}[\![exp_1]\!]\ g\ \{\ \mathcal{T}[\![exp_2]\!]\ g\ \{$ apply $[\![$ ope $]\!]\ \{\ c\ \}\ \}\ \}$

$\mathcal{T}[\![$ bas $]\!]\ g\ c$
$=$ push $[\![$ bas $]\!]\ \{\ c\ \}$

$\mathcal{T}[\![$ ide $]\!]\ g\ c$
$=$ content $[\![$ ide $]\!]\ \{\ c\ \}$

Figure 19 Auxiliary functions

```
apply [[ope]] ∈ C -c> C
    apply [[ope]] = do ( Vapply [[ope]], Bapply [[ope]] )
        Vapply [[ope]] ∈ Sta -c> T                                  "verify"
        Vapply [[ope]] = λ‹env,imp,out,tem› . #tem ≥≥ 2
        Bapply [[ope]] ∈ Sta -c> Sta                                "body"
        Bapply [[ope]] = λ‹env,imp,out,tem› .
                ‹ env, inp, out,‹ O [[ope]] ‹ tem ↓ 1, tem ↓ 2 › › § tem†2 ›
assign [[ide]] ∈ C -c> C
    assign [[ide]] = do ( Vassign [[ide]], Bassign [[ide]] )
        Vassign [[ide]] ∈ Sta -c> T                                 "verify"
        Vassign [[ide]] = λ‹env,imp,out,tem› . #tem ≥≥ 1
        Bassign [[ide]] ∈ Sta -c> Sta                               "body"
        Bassign [[ide]] = λ‹env,imp,out,tem› . [[ide]], inp, out, tem†1 ›
                ‹ env [ tem ↓ 1 / [[ide]] ], inp, out, tem†1 ›
content [[ide]] ∈ C -c> C
    content [[ide]] = λ‹env,imp,out,tem› . ‹ env, inp, out, ‹env [[ide]]› § tem ›
push [[bas]] ∈ C -c> C
    push [[bas]] = λ‹env,imp,out,tem› . ‹ env, inp, out, ‹B [[bas]]› § tem ›
read ∈ C -c> C
    read = do ( Vread , Bread )
        Vread ∈ Sta -c> T                                           "verify"
        Vread = λ‹env,imp,out,tem› . #inp ≥≥ 1
        Bread ∈ Sta -c> Sta                                         "body"
        Bread = λ‹env,imp,out,tem› .‹ env, inp † 1, out, ‹ inp ↓ 1 › § tem ›
write ∈ C -c> C
    write = do ( Vwrite , Bwrite )
        Vwrite ∈ Sta -c> T                                          "verify"
        Vwrite = λ‹env,imp,out,tem› . #tmp ≥≥ 1
        Bwrite ∈ Sta -c> Sta                                        "body"
        Bwrite = λ‹env,imp,out,tem› . ‹ env, inp, ‹ tmp ↓ 1 › § out, tem † 1›
```

Figure 20 Auxiliary functions for the conditional

```
Vcond ∈ Sta -c> T                                              "verify"
Vcond = λ‹env,inp,out,tem› . #tem ≥≥ 1 → tem ↓ 1 ∈ T , false
Scond ∈ Sta -c> T                                              "select"
Scond = λ‹env,inp,out,tem› . (tem ↓ 1) | T
Bcond ∈ Sta -c> Sta                                            "body"
Bcond = λ‹env,inp,out,tem› . ‹env,inp,out,tem † 1›
```

Figure 21 Store interpretation

```
Domains
    S      =    Sta                                    states
    A      =    Out + { "error" }                      answers
    C      =    S -c> A                                continuations
Constant
    fin ∈ C     fin = λ.‹env,inp,out,tem› . out in A
Auxiliary functions
    cond ∈ C × C -c> C
    cond (c1, c2) = LUB ( iftrue c1, iffalse c2 )
            where iftrue ∈ C -c> C
                  iftrue c = λsta . Vcond(sta) → [ Scond(sta) → c, ⊥ ] (Bcond sta)
                  iffalse ∈ C -c> C
                  iffalse c = λsta . Vcond(sta) → [ Scond(sta) → ⊥, c ] (Bcond sta)
    do ∈ (Sta -c> T) × (Sta -c> Sta) -c> C -c> C
    do (Vg, Bg) = λsta . Vg(sta) → c (Bg sta), "error" in A
```

From CPS to a flow grammar

Now the question is how does one derive anything like a flow grammar from the preceding description. The new idea here is to derive a flow grammar by manipulating the equations in the semantic function $\mathcal{T}$. Specifically, consider constructing an attribute grammar

with two attributes for each Syn node, *left* and *right*, which equal the result continuation (i.e., the value of the entire expression) and argument continuation (i.e., argument 'c' in the definition of $\mathcal{T}$). Starting with the most straightforward command, `skip`:

```
cmd ::= 'skip'
```

$\mathcal{T}$[[skip]]g c = c
cmd.left = $\mathcal{T}$[[skip]]g c
cmd.right = c
$\therefore$ cmd.left = $\mathcal{T}$[[skip]]g { cmd.right }

That is, cmd.left may be defined in terms of the value of incoming conditional simply by "evaluating" the semantic function.

As an example of an expression, consider the meaning of an identifier:

$\mathcal{T}$[[ide]] g c = content [[ide]] { c }

Assigning names to continuations as above gives:

exp.left = $\mathcal{T}$[[ide]] g c
exp.right = c

Re-writing gives:

exp.left = content [[ide]] { exp.right }
exp.right = c

This is similar to the previous case, except for the use of the content function: this represents an actual state change outside the flow of control and corresponds to a terminal in a flow grammar. Indeed, the first of these equations may be abstracted to a grammar production:

exp.left ::= $t_{\text{content}\ [\![\ \texttt{ide}\ ]\!]}$ exp.right

where exp.left and exp.right are non-terminals.

Now consider two sequential commands, $\texttt{cmd}_1$ ';' $\texttt{cmd}_2$:

cmd ::= $\texttt{cmd}_1$ ';' $\texttt{cmd}_2$

$\mathcal{T}[\![\texttt{cmd}_1\ \texttt{';'}\ \texttt{cmd}_2]\!]$ g c = $\mathcal{T}[\![\texttt{cmd}_1]\!]$ g { $\mathcal{T}[\![\texttt{cmd}_2]\!]$ g { c } }
$\texttt{cmd}_0$.left = $\mathcal{T}[\![\texttt{cmd}_1]\!]$ g { $\mathcal{T}[\![\texttt{cmd}_2]\!]$ g { c } }
$\texttt{cmd}_0$.right = c
$\texttt{cmd}_1$.left = $\mathcal{T}[\![\texttt{cmd}_1]\!]$ g { $\mathcal{T}[\![\texttt{cmd}_2]\!]$ g { c } }
$\texttt{cmd}_1$.right = { $\mathcal{T}[\![\texttt{cmd}_2]\!]$ g { c } }
$\texttt{cmd}_2$.left = $\mathcal{T}[\![\texttt{cmd}_2]\!]$ g { c }
$\texttt{cmd}_2$.right = { c }

With a little re-writing, this becomes:

$\texttt{cmd}_0$.left = $\texttt{cmd}_1$.left
$\texttt{cmd}_0$.right = c
[$\texttt{cmd}_1$.left = $\mathcal{T}[\![\texttt{cmd}_1]\!]$ g { $\texttt{cmd}_1$.right }]
$\texttt{cmd}_1$.right = $\texttt{cmd}_2$.left
[$\texttt{cmd}_2$.left = $\mathcal{T}[\![\texttt{cmd}_2\ ]\!]$ g { $\texttt{cmd}_2$.right }]
$\texttt{cmd}_2$.right = $\texttt{cmd}_0$.right

Once again, these equations mimic regular productions (chain rules, in fact). Note that the two parenthesized equations represent control flow local to the commands $\texttt{cmd}_1$ and $\texttt{cmd}_2$ – corresponding equations are specified by their respective productions.

Now consider the if/then/else construct:

cmd ::= 'if' exp 'then' cmd 'else' cmd 'fi'

$\mathcal{T}$[['if' exp 'then' cmd_1 'else' cmd_2 'fi']] g c =
    $\mathcal{T}$[[exp]] g { cond ($\mathcal{T}$[[cmd_1]] g { c }, $\mathcal{T}$[[cmd_2]] g { c }) }
cmd_0.left = $\mathcal{T}$[[exp]] g { cond ($\mathcal{T}$[[cmd_1]] g { c }, $\mathcal{T}$[[cmd_2]] g { c }) }
cmd_0.right = c
exp.left = $\mathcal{T}$[[exp]] g { cond ($\mathcal{T}$[[cmd_1]] g { c }, $\mathcal{T}$[[cmd_2]] g { c }) }
exp.right = cond ($\mathcal{T}$[[cmd_1]] g { c }, $\mathcal{T}$[[cmd_2]] g { c })
cmd_1.left = $\mathcal{T}$[[cmd_1]] g { c }
cmd_1.right = { c }
cmd_2.left = $\mathcal{T}$[[cmd_2]] g { c }
cmd_2.right = { c }

Which, with re-writing as above, becomes:

cmd_0.left = exp.left
cmd_0.right = c
[exp.left = $\mathcal{T}$[[exp]] g { exp.right }]
exp.right = cond (cmd_1.left, cmd_2.left) (*)
[cmd_1.left = $\mathcal{T}$[[cmd_1]] g { cmd_1.right }]
cmd_1.right = cmd_0.right
[cmd_2.left = $\mathcal{T}$[[cmd_2]] g { cmd_2.right }]
cmd_2.right = cmd_0.right

In this case there is the added complication of the use of the conditional function, cond, in (*). By partially evaluating this function it is possible to derive a new equation more suited to the needs of data flow analysis. Expanding the definition of cond (cmd_1.left, cmd_2.left) yields:

exp.right = LUB (iftrue(cmd_1.left), iffalse(cmd_2.left))

which may also be written as the two inequalities:

exp.right ≥ iftrue(cmd_1.left)
exp.right ≥ iffalse(cmd_2.left)

Replacing the equality (*) above with these inequalities once again yields a set of straightforward relations among the continuations for a production.

The while loop adds the complication of a fixed point. At first sight, this may appear impenetrable:

```
cmd ::= 'while' exp 'do' cmd 'od'

T[['while' exp 'do' cmd1 'od']] g c
= FIX ( λc' . T[[exp]] g { cond (T[[cmd1]] g { c' }, c ) } )
```

The solution is derived from the desire to find a reasonable meaning for the equation [85]:

```
T[['while' exp 'do' cmd1 'od']] g c
  = T[[exp]] g
      { cond (T[[cmd1]] g
                { T[['while' exp 'do' cmd1 'od']] g { c } },
             c )
       }
```

Proceeding as before, the following continuations may be defined:

```
cmd0.left = T[['while' exp 'do' cmd1 'od']] g c
cmd0.right = c
exp.left = T[['while' exp 'do' cmd1 'od']] g c
exp.right =
      { cond (T[[cmd1]] g
                { T[['while' exp 'do' cmd1 'od']] g { c } },
             c )
        }
cmd1.left = T[[cmd1]] g
                { T[['while' exp 'do' cmd1 'od']] g { c } }
cmd1.right = { T[['while' exp 'do' cmd1 'od']] g { c } }
```

Re-writing gives:

cmd_0.left = exp.left
cmd_0.right = c
[exp.left = $\mathcal{T}$[[exp]] g { exp.right }]
exp.right ≥ iftrue(cmd_1.left)
exp.right ≥ iffalse(cmd_0.right))
[cmd_1.left = $\mathcal{T}$[[cmd_1]] g { cmd_1.right }]
cmd_1.right = cmd_0.left (**)

Note that in (**) the continuation for the entire construct has been substituted for the occurrence on the right hand side yielding the expected circularity in the equations (assuming that cmd_1 does not escape via a goto statement).

Lastly, consider the begin/end and goto commands. These are considered together because their meanings are inherently intertwined: they are the only commands that directly involve the "goto environment" g in the specification. First, using a similar technique used for the while loop above, the begin/end block is re-written as:

$\mathcal{T}$[['begin' lab_1':'cmd_1 ... lab_n':'cmd_n 'end']] g c
= c_1
where rec
c_1 = $\mathcal{T}$[[cmd_1]] g' { c_2 }
c_2 = $\mathcal{T}$[[cmd_2]] g' { c_3 }
...
c_n = ,$\mathcal{T}$[[cmd_n]] g' { c }
g' = g [c_1 / [[lab_1]], c_2 / [[lab_2]],...,c_n / [[lab_n]]]

Aside from the value of g', this can clearly be re-written in terms of cmd_i.left and cmd_i.right, $1 \le i \le n$. Turning to g', it should be clear that this function can be partially evaluated for this language; this is simply one of the tasks of a typical compiler: symbol table management. Further, it can be verified that all labels used in goto statements are, in fact, visible. Thus, the semantic function for the goto statement:

```
cmd ::= 'goto' lab
```

```
T[['goto' lab]]gc
=g[[lab]]
```

may be written as:

```
cmd.left = g[[lab]]
cmd.right = c
```

Note that cmd.right is not used in the preceding equation: the continuation c is simply ignored.

To summarize, in each of the preceding cases, and in fact for the entire language, inequalities (equations when only one constraint is present) of the form:

$$a.b = f(x.y)$$

can be constructed. These correspond naturally to productions in a grammar. Thus, by partially evaluting a store semantics it is possible to derive a grammar representing the control flow of a program, validating the notion of the flow grammar for use in FACT.

IV.8 Summary

This chapter introduced the first important contribution of this dissertation, the *flow grammar*, an original model for the uniform representation of intra- and interprocedural control flow. Two approaches for deriving a flow grammar were given, one starting from the semantics of single execution paths and building up to multiple execution paths, eventually yielding a structure equivalent to a grammar. It was proven that the GFP solution to a set of flow equations corresponding to a flow grammar conservatively approximates the meet of all path strings generated by the grammar. Finally, it was shown how a grammar

results from a particular partial evaluation of a CPS store semantics. The first problem set out in Section I.3 has now been solved.

V Data flow analysis

This chapter addresses the second problem outlined in the first chapter by defining original *interpretations* of flow grammars that allow the specification of data flow analysis problems; the correctness of which is assured by Theorem 9. In addition, *algorithms* for solving the equations resulting from the various interpretations are described. Traditional intraprocedural analysis is recast using regular flow grammars. Original interpretations of context-free flow grammars are defined for interprocedural analyses. A technical result arises from considering the "granularity" of the control flow model in the presence of bidirectional information flow. Several original algorithms for solving interprocedural analyses are presented. The interpretations are important because they provide the skeleton around which flow analyses may be implemented, several examples of which appear in the next chapter.

V.1 Introduction

One of the main strengths of flow grammars is the ability to represent both intra- and interprocedural control flow in a single uniform model. The difference between a strictly intraprocedural flow grammar and an interprocedural flow grammar is that the former is regular, while the latter is context-free. This subtle shift reflects a significant impact of interprocedural control flow on data flow analyses: in the intraprocedural case, taking calling context into account may be necessary to prevent inordinate loss of precision.

V.2 Overview

As discussed in the last chapter, data flow analysis using FGs involves defining an interpretation of the FTs and FNs, along with an information space. The FPs serve as the skeleton for inequalities representing the overall flow problem (Definition 41).

In the case of a RFG, a straightforward bidirectional interpretation is given in which each FT is associated with two functions: a forward flow function and a backward flow function. This generalizes the existing framework of Kam and Ullman [44] allowing, for example, the partial availability problem of Morel and Renvoise [62] to be expressed in the FG framework. Forward and backward analyses are shown to be special cases.

CFGs, used in FACT to represent interprocedural control flow, require a more complex interpretation. The FG formulation relies on *partial function lattices* (*PFLs*), similar to the PHI functions of [78]. It also bears a resemblance to Van Hentenryck et al.'s "multiple" *abstract result* variant for the abstract interpretation of Prolog [87]. Several examples are presented, followed by a discussion of a family of algorithms for solving the resulting problems.

V.3 Intraprocedural analysis

This section introduces an intraprocedural analysis framework based on FGs. While similar to the development of Kam and Ullman [44] and Cousot and Cousot [18], this formulation differs in that it is based on the simple FG structure rather than the control flow graph structure. Referring to the functional fixed points of the last chapter, intraprocedural analysis involves computing the values of the functions at a single input, greatly simplifying the data flow analysis problem. The framework here is more general than that of the previous chapter by associating two flow functions with each FT, one representing the forward information flow and one representing the backward information flow. When combined with the FG, the result is a finite set of equations whose fixed point is the desired solution. This set of equations is equivalent to combining a set of equations for a forwards FG and corresponding backwards FG (cf. Section IV.6.9) using different terminal mapping functions. By Theorem 9, this is guarenteed to consider all possible forward and backward paths from each point in the program. By restricting the form of flow functions, forward and backward flow problems are derived as special cases.

V.3.1 Bidirectional interpretation

Definition 43: A *bidirectional intraprocedural flow grammar problem (BraFGP)* is an instance $\langle G, f, b, \upsilon \rangle$ of a MDFAF $\langle L, \wedge_L, F \rangle$, where:

1. $G=\langle V,T,P,S \rangle$ is a RFG representing the control flow of a program.
2. $f: T \rightarrow F$ is the *forward flow function map* representing the abstract effect of forward control flow through each FT.
3. $b: T \rightarrow F$, the *backward flow function map* representing the abstract effect of backward control flow through each FT.
4. $\upsilon: L$ is an initial approximation of the abstract state at the start of execution of the program.

Together these components represent an intraprocedural program analysis in which information flows in both directions. The FG G induces a set of relationships among the FNs in terms of the FT mappings f and b.

V.3.2 Greatest fixed point solution

Since the MOP solution is uncomputable in general, the strategy is to compute the greatest fixed point (GFP) solution to a set of equations corresponding to the FG. Let $\langle G,f,b,\upsilon\rangle$ be an instance of a MDFAF $\langle L,\wedge_L,F\rangle$. Then the set of flow equations is simply the union of the flow equations corresponding to the forward FG using f in place of t in Definition 41, and the flow equations corresponding to the backward FG using b in place of t in Definition 41. The goal is to compute the solution of the resulting equations at υ.

It is sometimes convenient to view the flow equations as inequalities arising from the individual productions. Suppose one of the flow equations is of the form:

$$\mathcal{K}_{Si} = C_1 \wedge \ldots \wedge C_n$$

This may be re-written as several inequalities as follows:

$$\mathcal{K}_{Si} \leq C_1$$
$$\ldots$$
$$\mathcal{K}_{Si} \leq C_n$$

This view is useful in instances where only a small number (typically one) of the C_i's have changed, in which case $\mathcal{K}_{Si}$ may be recomputed using a single meet of the new value of C_i with $\mathcal{K}_{Si}$ itself.

V.3.3 Algorithms

Various algorithms may be used to solve the flow systems described by instances of BraF-GPs. The absence of specific knowledge about the lattice and function space, however, suggests the use of general iteration algorithms, such as described in [51, 44, 62]. This type of algorithm has the following general structure, where ζ:N→L is the solution map:

$\zeta \leftarrow \lambda n.\top_L$;
repeat
  $\Re$: "generate one or more values for ζ from the flow inequalities/equations"
until done

Whenever a value in ζ changes, there is the possibility that one or more of its dependents no longer meet the flow constraints. In the absence of other information, such dependent values must be reconsidered. Two standard ways of ensuring this are as follows:

1. Fixed iteration: order the inequalities (equations) and at step $\Re$ apply all inequalities (equations) in this order on each iteration. The loop terminates when ζ has not changed for the entirety of an iteration.

   In general, the value of ζ at a given point can only be lowered with respect to the partial ordering on the lattice. Once approximating information has entered the system it can never be removed. Thus, those inequalities involving the constants υ_s and υ_e, representing starting and ending conditions, respectively, need only be applied once. In fact, they can be used to initialize the approximations that hold at the starting and ending points of the program.
2. Worklist: a list is used to keep track of values needing recompilation. When a value in ζ changes, all its dependents are inserted into the queue. Loop termination occurs when the list is empty.

In addition, there are two subvariants of each of these general strategies:

A. the system may be treated as inequalities where a re-computation is of the form:

$$\zeta \leftarrow \zeta\,[\zeta(Y) \wedge_L (f_1 \circ f_2 \circ \ldots \circ f_n(\zeta(X))\,/Y],$$

where $[\![Y]\!] \leq_L f_1 \circ f_2 \circ \ldots \circ f_n([\![X]\!])$ is a constraint in the flow inequality system

B. when the inequalities for each FN have be grouped into equalities using $\wedge_L$:

$$\zeta \leftarrow \zeta[\begin{pmatrix} f_{1,1} \circ f_{1,2} \circ \ldots \circ f_{1,n_1}(\zeta(X_1)) \\ \wedge_L f_{2,1} \circ f_{2,2} \circ \ldots \circ f_{2,n_2}(\zeta(X_2)) \\ \ldots \\ \wedge_L f_{m,1} \circ f_{m,2} \circ \ldots \circ f_{m,n_m}(\zeta(X_m)) \end{pmatrix} /Y]$$

V.3.4 Forward and backward analyses

Given the general bidirectional framework, forward and backward analyses are derived by restriction as follows:

Forward: $b = \lambda t.T_L$

Backward: $f = \lambda t.T_L$

Note that both of these simplify the flow equations to the usual intraprocedural frameworks described in Section II.3. Strategies for solving such systems have been well studied [29].

V.3.5 Effect of granularity

In Section IV.2 it was assumed that FACT had some notion of appropriate "points of interest" for the language being implemented and the desired analyses. There are, however, many possible interesting sets of points for a given program. One possibility is the elimi-

nation of "chains" of productions to reduce the size of the grammar. In some sense, a FG that has been subjected to the basic block transformation has a coarser "granularity" than the original FG. This transformation is benign in a undirectional setting, affecting only the time and space requirements for computing the flow solution. It is interesting to note, however, that in the bidirectional setting the granularity can affect the solution computed, as demonstrated in the following observation.

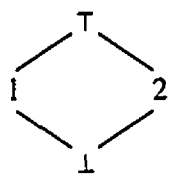
Observation: General bidirectional analyses are sensitive to the "granularity" of the underlying RFG. That is, there exist FG problems $\langle G_1, f, b, \upsilon \rangle$, and $\langle G_2, f, b, \upsilon \rangle$ be instances of a MDFAF $\langle L, \wedge_L, F \rangle$ with RFGs $G_1=\langle V_1,T,P_1,S \rangle$ and $G_2=\langle V_2,T,P_2,S \rangle$ with $\mathcal{L}(G_1)=\mathcal{L}(G_2)$, $V_1 \subseteq V_2$; and $\forall N \in V_1 . \mathcal{L}(\langle V_1,T,P_1,N \rangle)=\mathcal{L}(\langle V_2,T,P_2,N \rangle)$; such that $\exists N \in V_1 . \zeta_2(N) <_L \zeta_2(N)$.

Proof: Consider the flow problems $\langle G_1, f, b, \upsilon \rangle$, and $\langle G_2, f, b, \upsilon \rangle$:

G_1: $X ::= t\ Y$ $Y ::= u\ Z$ $Z ::= \varepsilon$

G_2: $X ::= t\ u\ Z$ $Z ::= \varepsilon$

where G1 and G2 are backward FGs, and L is:



F is the set of all monotonic functions on L. And, finally, f and b are as follows (note that all these functions are not only monotonic, but distributive):

x	f(t) (x)	f(u) (x)	b(t) (x)	b(t) (x)
⊤	1	⊤	⊤	2
1	1	⊤	⊥	⊥
2	⊥	⊥	⊤	2
⊥	⊥	⊥	⊥	⊥

and $\upsilon = \top$.

The flow equations for G_1 are:

$$\mathcal{K}_X = (\delta[\![t]\!]) \circ \mathcal{K}_Y \wedge_{F(L)} \lambda s.s$$
$$\mathcal{K}_Z = (\phi[\![u]\!]) \circ \mathcal{K}_Y \wedge_{F(L)} \lambda s.s$$
$$\mathcal{K}_Y = (\phi[\![t]\!]) \circ \mathcal{K}_X \wedge_{F(L)} (\phi[\![t]\!]) \circ \mathcal{K}_X$$

And the flow equations for G_2 are:

$$\mathcal{K}_X = (\delta[\![t]\!]) \circ (\delta[\![u]\!]) \circ \mathcal{K}_Z$$
$$\mathcal{K}_Z = (\phi[\![u]\!]) \circ (\phi[\![t]\!]) \circ \mathcal{K}_X$$

Now the GFP solution of the first set of equations at input υ is:

$$\mathcal{K}_X(\upsilon) = \mathcal{K}_Y(\upsilon) = \mathcal{K}_Y(\upsilon) = \bot$$

but the GFP of the second set of equations is:

$$\mathcal{K}_X(\upsilon) = \mathcal{K}_Z(\upsilon) = \top.$$

This technical result implies it is important when performing a bidirectional analysis to select an appropriate set of program points. Typically, the set of points correspond to the entries and exits of basic blocks, which, as mentioned above, can be computed automatically via a straightforward FG transformation.

V.4 Interprocedural analysis – introduction

The flow grammar was introduced in the last chapter specifically to permit interprocedural flow analysis. Indeed, in the previous section it was overkill to require the use of functions when simple lattice values suffice (though the argument could be made that loops may be implemented using recursion, in which case an interprocedural problem results). For inter-

procedural analysis, however, the mechanisms are well-suited to the required task. This section describes the general idea by way of an example. Subsequent sections formalize the notions presented so that they may be used in a tool such as FACT.

V.4.1 Example

Consider the program in Figure 22 and its corresponding backwards FG and flow equations, in Figures 23 and 24, respectively, where δ is the terminal mapping function. Suppose that semantic functions have been associated with each of the terminals, and the fixed point of the equations has been computed with GFP solutions $\mathcal{K}_1,\ldots, \mathcal{K}_{10}$ for the variables $\mathcal{K}_{S1},\ldots, \mathcal{K}_{S10}$, respectively. One may compute the abstract state at some point, say 8, of the main program, by applying $\mathcal{K}_8$ to whatever abstract state is known to hold on exit from the program, *exit_state*:

$\mathcal{K}_8$(*exit_state*) ≡ abstract state that holds at point 8 of the program

For a point in the procedure `nfact`, say 3, it is possible to compute the abstract state that holds assuming that a given state, say *ret_state*, holds on return from the procedure:

$\mathcal{K}_3$ (*ret_state*) ≡ abstract state that holds at point 3 during an invocation of the procedure `nfact` in which *ret_state* holds on return

Parameterizing by the calling context allows greater precision to be achieved in many analyses, such as constant propagation. It is important to note, however, that the abstract state computed for a single input at such a point does not necessarily yield a conservative approximation of the abstract state at that point. Instead, all contexts that may occur dur-

ing execution must be taken into account when computing a conservative value for the nonterminal.

Figure 22 An example Pascal program demonstrating interprocedural control flow.

```
    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.
```

Figure 23 Backward flow grammar representing program in Figure 22

$S_1 ::= t_{1/2}\, S_2$
$S_1 ::= t_{1/3}\, S_3$
$S_2 ::= t_2\, S_6$
$S_3 ::= t_3\, S_4$
$S_4 ::= t_{4/1}\, S_1\, t_{6/5}\, S_5$
$S_5 ::= t_5\, S_6$
$S_6 ::= \varepsilon$
$S_7 ::= t_7\, S_8$
$S_8 ::= t_{8/1}\, S_1\, t_{6/9}\, S_9$
$S_9 ::= t_9\, S_{10}$
$S_{10} ::= \varepsilon$

Figure 24 Backward flow equations corresponding to grammar in Figure 22

$$
\begin{aligned}
\mathcal{K}_{S1} &= \mathcal{b}[\![t_{1/2}]\!] \circ \mathcal{K}_{S2} \wedge \mathcal{b}[\![t_{1/3}]\!] \circ \mathcal{K}_{S3} \\
\mathcal{K}_{S2} &= \mathcal{b}[\![t_2]\!] \circ \mathcal{K}_{S6} \\
\mathcal{K}_{S3} &= \mathcal{b}[\![t_3]\!] \circ \mathcal{K}_{S4} \\
\mathcal{K}_{S4} &= \mathcal{b}[\![t_{4/1}]\!] \circ \mathcal{K}_{S1} \circ \mathcal{b}[\![t_{6/5}]\!] \circ \mathcal{K}_{S5} \\
\mathcal{K}_{S5} &= \mathcal{b}[\![t_5]\!] \circ \mathcal{K}_{S6} \\
\mathcal{K}_{S6} &= \lambda s.s \\
\mathcal{K}_{S7} &= \mathcal{b}[\![t_7]\!] \circ \mathcal{K}_{S8} \\
\mathcal{K}_{S8} &= \mathcal{b}[\![t_{8/1}]\!] \circ \mathcal{K}_{S1} \circ \mathcal{b}[\![t_{6/9}]\!] \circ \mathcal{K}_{S9} \\
\mathcal{K}_{S9} &= \mathcal{b}[\![t_9]\!] \circ \mathcal{K}_{S10} \\
\mathcal{K}_{S10} &= \lambda s.s
\end{aligned}
$$

V.4.2 Interprocedural live variables

To illustrate the ideas of the previous section more fully, consider the problem of computing live variables for the program in Figure 22. The appropriate terminal symbol function mappings are shown in Figure 25. Analysis proceeds as follows: after the program has completed execution, the only live values are those used by the operating system.[1] Assuming that the operating system accesses neither `i` nor `f`, the set of variables live at point 10 of the program is simply:

$$
\begin{aligned}
&\{ \text{ variables live at point 10 } \} \\
&= \mathcal{K}_{10}(\{ \text{ variables live at program exit } \}) \\
&= \mathcal{K}_{10}(\{\})
\end{aligned}
$$

1. The output stream could also be considered; doing so, however, adds no insight.

$$= (\lambda s \,.\, s)\,(\{\,\})$$
$$= \{\,\}$$

Continuing backwards through the equations, it is easily seen that the set of variables live at point 9 is:

$$\mathcal{K}_9\,(\{\,\}) = (\lambda x \,.\, (x \cup \{\mathtt{f}\}))\,^{\circ}\,(\mathcal{K}_{10}\,(\{\,\})) = \{\mathtt{f}\}$$

implying that at point 9 of the program, the variable `f` is live.

Determining the set of variables live at point 8 is somewhat more involved:

$$\begin{aligned}\mathcal{K}_8\,(\{\,\}) &= \mathcal{b}\,[\![t_{8/1}]\!] \circ \mathcal{K}_1 \circ \mathcal{b}\,[\![t_{6/9}]\!] \circ \mathcal{K}_9\,(\{\,\}) \\ &= \mathcal{b}\,[\![t_{8/1}]\!]\,(\mathcal{K}_1\,(\{\mathtt{f}\}))\end{aligned}$$

This implies that the value of $\mathcal{K}_1$ must be computed for the input $\{\mathtt{f}\}$. In some sense, the abstract function for the procedure `nfact` must be "invoked" at this point to determine what variables are live, given that `f` is live after execution of the procedure. It is possible to continue analyzing the main program, however, assuming that no variables are live at the beginning of the call; when a new value is computed for $\mathcal{K}_1$, these values must be recomputed. The complete iteration using this technique is shown in Figure 26. The current value of a function is written as a set of input/output pairs, where the input is separated from the output by the $\rightarrow$ symbol. For example, the fact that $\mathcal{K}_9\,(\{\,\}) = \{\mathtt{f}\}$, with nothing currently known of the other values in the domain of $\mathcal{K}_9$ is written: $\mathcal{K}_9 = \{\{\,\} \rightarrow \{\mathtt{f}\}\}$.

Figure 25 Terminal set mappings for live variable analysis of program in Figure 22

$$
\begin{array}{ll|ll}
\mathcal{b}[\![t_{1/2}]\!] & = \lambda x\,.\,(x \cup \{i\}) & \mathcal{b}[\![t_{6/5}]\!] & = \lambda x\,.\,x \\
\mathcal{b}[\![t_{1/3}]\!] & = \lambda x\,.\,(x \cup \{i\}) & \mathcal{b}[\![t_{6/9}]\!] & = \lambda x\,.\,x \\
\mathcal{b}[\![t_{2}]\!] & = \lambda x\,.\,(x - \{f\}) & \mathcal{b}[\![t_{7}]\!] & = \lambda x\,.\,(x - \{i\}) \\
\mathcal{b}[\![t_{3}]\!] & = \lambda x\,.\,(x \cup \{i\}) & \mathcal{b}[\![t_{8/1}]\!] & = \lambda x\,.\,x \\
\mathcal{b}[\![t_{4/1}]\!] & = \lambda x\,.\,x & \mathcal{b}[\![t_{9}]\!] & = \lambda x\,.\,(x \cup \{f\}) \\
\mathcal{b}[\![t_{5}]\!] & = \lambda x\,.\,(x \cup \{f, i\}) & \mathcal{b}[\![t_{10}]\!] & = \lambda x\,.\,x
\end{array}
$$

Figure 26 Complete live variable computation for the program in Figure 22

State	Iteration Number			
	1	2	3	4
$\mathcal{K}_6$	Φ	{{f}→{f}}	{{f}→{f},{f,i}→{f,i}}	{{f}→{f},{f,i}→{f,i}}
$\mathcal{K}_5$	Φ	{{f}→{f,i}}	{{f}→{f,i},{f,i}→{f,i}}	{{f}→{f,i},{f,i}→{f,i}}
$\mathcal{K}_4$	Φ	{{f}→{}}[b]	{{f}→{}, {f,i}→{}}	{{f}→{i},{f,i}→{i}}
$\mathcal{K}_3$	Φ	{{f}→{i}}	{{f}→{i},{f,i}→{i}}	{{f}→{i},{f,i}→{i}}
$\mathcal{K}_2$	Φ	{{f}→{}}	{{f}→{}, {f,i}→{i}}	{{f}→{}, {f,i}→{i}}
$\mathcal{K}_1$	Φ	{{f}→{i}}	{{f}→{i},{f,i}→{i}}	{{f}→{i},{f,i}→{i}}
$\mathcal{K}_{10}$	{{}→{}}	{{}→{}}	{{}→{}}	{{}→{}}
$\mathcal{K}_9$	{{}→{f}}	{{}→{f}}	{{}→{f}}	{{}→{f}}
$\mathcal{K}_8$	{{}→{}}[a]	{{}→{i}}	{{}→{i}}	{{}→{i}}
$\mathcal{K}_7$	{{}→{}}	{{}→{}}	{{}→{}}	{{}→{}}

a. And add element {f}→{} to set $\mathcal{K}_6$.
b. And add element {f, i}→{} to set $\mathcal{K}_6$.

This is a fixed point computation in which the values being computed are elements from a domain of *partial functions*. The remainder of this chapter explains appropriate function lattices and gives mathematical interpretations of bidirectional, forward and backward problems as was done for the intraprocedural case.

V.4.3 Interpretation

Proceeding as in Section V.3 above, forward, backward and bidirectional interpretations of context-free FGs are defined. Unlike the intraprocedural case, however, it is more convenient to consider first the backward and forward analyses and then combine them into a bidirectional interpretation. An equivalent forward interpretation based on flow graphs appears in [78], while both the backward and bidirectional interpretations are original.

Backward interpretation

Definition 44: A *backward interprocedural flow grammar problem (berFGP)* is an instance $\langle G, \delta, \upsilon \rangle$ of a MDFAF $\langle L, \wedge_L, F \rangle$, where:

1. $G=\langle V,T,P,S \rangle$ is a CFG representing some program.
2. $\delta: T \rightarrow F$, the *backward flow function map* representing the abstract effect of backward control flow through each FT, with respect to the program.
3. υ is the initial value known to hold at the end of execution.

The goal is to compute a function representing the abstract meaning of each non-terminal in the program. Equations are derived as in Definition 41 and, as before, these equations may be separated into a set of flow inequalities using $\leq_{F(L)}$. The derived set of backwards flow inequalities for the example is Section V.4.1 is shown in Figure 27.

Figure 27 Flow inequalities derived from backwards flow grammar in Figure 23

$$\mathcal{K}_{S1} \leq_{F(L)} \mathcal{b}[\![t_{1/2}]\!] \circ \mathcal{K}_{S2}$$

$$\mathcal{K}_{S1} \leq_{F(L)} \mathcal{b}[\![t_{1/3}]\!] \circ \mathcal{K}_{S3}$$

$$\mathcal{K}_{S2} \leq_{F(L)} \mathcal{b}[\![t_2]\!] \circ \mathcal{K}_{S6}$$

$$\mathcal{K}_{S3} \leq_{F(L)} \mathcal{b}[\![t_3]\!] \circ \mathcal{K}_{S4}$$

$$\mathcal{K}_{S4} \leq_{F(L)} \mathcal{b}[\![t_{4/1}]\!] \circ \mathcal{K}_{S1} \circ \mathcal{b}[\![t_{6/5}]\!] \circ \mathcal{K}_{S5}$$

$$\mathcal{K}_{S5} \leq_{F(L)} \mathcal{b}[\![t_5]\!] \circ \mathcal{K}_{S6}$$

$$\mathcal{K}_{S6} \leq_{F(L)} \lambda s.s$$

$$\mathcal{K}_{S7} \leq_{F(L)} \mathcal{b}[\![t_7]\!] \circ \mathcal{K}_{S8}$$

$$\mathcal{K}_{S8} \leq_{F(L)} \mathcal{b}[\![t_{8/1}]\!] \circ \mathcal{K}_{S1} \circ \mathcal{b}[\![t_{6/9}]\!] \circ \mathcal{K}_{S9}$$

$$\mathcal{K}_{S9} \leq_{F(L)} \mathcal{b}[\![t_9]\!] \circ \mathcal{K}_{S10}$$

$$\mathcal{K}_{S10} \leq_{F(L)} \lambda s.s$$

Forward interpretation

Definition 45: A *forward interprocedural flow grammar problem (ferFGP)* is an instance $\langle G, f, \upsilon \rangle$ of a MDFAF $\langle L, \wedge_L, F \rangle$, where:

1. $G=\langle V,T,P,S \rangle$ is a CFG.
2. $f: T \rightarrow F$, the *forward flow function map* representing the abstract effect of forward control flow through each FT.
3. υ is the initial flow value at the start of execution.

Again using Definition 41, we derive a set of flow equations from the *forward* FG. The GFP solution to these equations yields sufficient information to compute flow values for each point in the program. And, as shown above, flow inequalities may be used instead of equations.

Bidirectional interpretation

Definition 46: A *bidirectional interprocedural flow grammar problem (BerFGP)* is an instance ‹G, f, b, υ› of a MDFAF ‹L,$\wedge_L$,F›, where:

1. G=‹V,T,P,S› is a CFG.
2. f:T→F is the *forward flow function map* representing the abstract effect of forward control flow through each FT.
3. b:T→F, the *backward flow function map* representing the abstract effect of backward control flow through each FT.
4. υ is the initial value at the start of execution.

For this to work correctly, either a forward or backward FG must be used to derive both the forward and backward sets of equations. This implies that all procedures must be single entry and single exit (cf. Section IV.6.9). Furthermore, this construction constrains *both* entry and exit of all procedure to be less than or equal to the identity function. Thus, it is not possible in general to restrict a BerFGP to either a ferFGP or berFGP.

V.4.4 Avoiding unnecessary values

As described above, there are a number of inherent problems with using TFLs as domains of interpretation. Consider, for example, a berFGP instance ‹‹V,T,P,S›, b, υ› of a MDFAF ‹L,$\wedge_L$,F› with GFP solution ζ. Computing a flow value from L for each FN X in the main program (assuming the point is not also within a procedure) is straightforward: simply apply ζ(X) to the flow value known to hold at the beginning of the program. To compute the flow value for a FN X within a procedure, however, requires the set of all flow values with which the procedure(s) containing X may be "invoked." Because the procedures may be recursive, this may, in turn, require what is essentially another flow analysis to determine all such inputs! Clearly, a demand-driven approach is more appropriate. This allows

both the computation of the flow function fixed points and the required set of inputs for each procedure to be combined.

One way of viewing this idea is to define a lattice of partial functions, and use it instead of total functions as before. That is the approach taken here. It should be noted, however, that the idea of avoiding computation of unneeded values goes back at least to Sharir and Pnueli's Φ functions [78].

Partial function lattices

Definition 47: Let L be a meet semi-lattice.

> $PF(L) = L \rightharpoonup L$ is the *partial function flow analysis lattice* containing all partial functions from L to L under the ordering $\leq_{PF(L)}$ defined as follows: for $f,g \in F(L)$, $f \leq_{PF(L)} g$ **iff** $dom(f) \supseteq dom(g)$ and $f(x) \leq_L g(x), \forall x \in dom(g)$. In this case, f is said to *approximate* g.

Observation: The corresponding meet operator, $\wedge_{PF(L)}$, is:

$$f \wedge_{PF(L)} g = \lambda x. \begin{cases} f(x), & x \in dom(f) - dom(g) \\ g(x), & x \in dom(g) - dom(f) \\ f(x) \wedge_L g(x), & x \in dom(f) \cap dom(g) \end{cases}$$

Proof: To show: $f \wedge_{PF(L)} g = f$ **iff** $f \leq_{PF(L)} g$

> **Only-if**: Assume $f \wedge_{PF(L)} g = f$. Then $dom(f)=dom(f \wedge_{PF(L)} g) \supseteq dom(g)$. Thus $dom(g)-dom(f)=\emptyset$, so $(f \wedge_{PF(L)} g)(x)=f(x) \wedge_L g(x)$ [def'n. of $\wedge_{PF(L)}$] = f(x) [by assumption], $\forall x \in dom(g)$. So $f(x) \leq_L g(x), \forall x \in dom(g)$ [since $\wedge_L$ is the meet of L and $\leq_L$ is the partial order on L] and thus $f \leq_{PF(L)} g$.
>
> **If**: Assume $f \leq_{PF(L)} g$. Then $dom(f) \supseteq dom(g)$ and $f(x) \leq_L g(x)$, $\forall x \in dom(g)$ [def'n. of $\leq_{PF(L)}$]. This implies $f(x) \wedge_L g(x)=f(x)$, $\forall x \in dom(f) \cap dom(g)$ [since $\wedge_L$ is the meet of L and $\leq_L$ is the partial order on L]. Now, $(f \wedge_{PF(L)} g)(x)=f(x)$, $\forall x \in dom(f)-dom(g)$ [def'n. of $\wedge_{PF(L)}$] and $(f \wedge_{PF(L)} g)(x)=f(x)$, $\forall x \in dom(g)-dom(f)$ [vacuous: $dom(g)-dom(f)=\emptyset$]. Thus, $f \wedge_{PF(L)} g=f$.

Note that $T_{PF(L)} = \Phi$, the partial function defined at no inputs, and is $\perp_{PF(L)}=\lambda x.\perp_L$, the total function that maps all inputs to bottom.

V.4.5 A note on compositionality

An implicit goal of the interprocedural interpretation has been to ensure compositionality. A more general approach to mapping productions is to introduce higher order functions to map each production. For example:

$$X ::= t\ Y\ u\ Z$$

in a backward analysis could be interpreted as:

$$\mathcal{K}_X ::= \mathcal{F}(\delta[\![t]\!], \mathcal{K}_Y, \delta[\![u]\!], \mathcal{K}_Z)$$

where $\mathcal{F}: F(L)\times F(L)\times F(L)\times F(L) \rightarrow F(L)$. This causes many problems, however. Since each FP would have to be interpreted individually, grammar transformations would be very difficult to apply effectively. At the same time, computing partial functions would not be possible unless restrictions were placed on the interpretation functions, because there is no way of knowing in advance which specific arguments will be queried.

V.5 Algorithms

As with the intraprocedural problems above, there are many possible algorithms for solving interprocedural flow problems. Given the intended use of FACT, it seems that the most appropriate is a general purpose iteration. Once the flow equations have been determined, a data flow analysis is initiated by requesting the values of the various flow functions, given some initial value or values known to hold at the start or end of program execution.

From that point, the iteration may proceed in any of a number of ways to compute the required values. In this section a design space classifying several possibilities is presented.

The main goal is to minimize the number of function applications and meet operations performed. We proceed in the style of Jourdan and Parigot, who have developed algorithms for solving "grammar flow analysis problems" [43]. Solving FG interpretations differs in that information does not flow either strictly bottom-up or top-down, a basic requirement for their algorithms. These algorithms are variants of an algorithm originally presented by Sharir and Pnueli [78], but, in addition, are intended to reduce the amount of computation for certain problems.

V.5.1 Overview of the algorithms

As with all fixed point iteration algorithms, the variables of the flow equations are initialized to some value and then the equations are applied iteratively, yielding less precise approximations to satisfy more constraints, until all the constraints in the flow equations are satisfied. A backwards problem has been assumed throughout; corresponding algorithms for forward analyses are easily derived.

V.5.2 Notation

Let $G=\langle V,T,P,S\rangle$ be a FG. Each production $p \in P$ is of the form $X_0 ::= X_1\ X_2 \ldots\ X_{n_p}$, where n_p is the number of symbols on the right hand side of p. The occurrences of symbols in a production are numbered left to right, $p[0] = X_0$ being the left hand side FN, $p[i]=X_i$, $0 < i \leq n_p$ being the right hand side symbols.

V.5.3 Generic iteration

A straightforward algorithm, shown in Figure 28, is simply to iterate over the productions in the grammar computing new approximations for each point in the program. After an iteration with no changes the algorithm terminates. Note that the lattice L must be finite to guarantee termination; otherwise it is possible to add an unbounded number of values to the domain at the step labelled initiate. The various labels indicate the main components of the algorithm. By modifying and re-arranging these components, many algorithms of differing character may be derived.

Note that the order of production selection in this general algorithm is unspecified. This admits the possibility that the processing of the productions "goes against the flow," slowing the movement of information. Thus, one immediate improvement is to order the productions to ensure that when a value is computed, the values upon which it depends are precise as possible. A reverse topological sort of the productions by their embedded FNs is appropriate for a backwards analysis.

V.5.4 Design space

The possible changes to the generic algorithm of the previous section may be classified as follows:

1. select-prod 1: Should the flow system be treated as equalities or inequalities?
2. select-value: Should individual inputs to the flow functions be propagated individually or in groups? If the latter, what criteria should be used to decide the groups?
3. select-prod 2: Is a worklist or production iteration approach more appropriate?
4. guess: When a new value is added to the domain of a partial function, what estimate should be used for the current value?

5. compute: What bounds, if any, should be placed on the amount of computation? What strategies can be used to meet such bounds?

These sets of questions form five design dimensions. Except for the last dimension, the effect of the other dimensions is only to change the time and space needed to compute a fixed point solution. The last dimension adds the possibility of reducing the computation at the expense of precision.

Equalities vs. inequalities

In the general algorithm the system is being treated as a set of inequalities. At the select-prod stage, there are at least two other obvious possibilities. For example, given the following FPs for S_1:

$$S_1 ::= t_{1/2}\ t_{2/3}\ S_3$$
$$S_1 ::= t_{1/5}\ t_5\ S_3$$
$$S_1 ::= t_{1/6}\ t_6\ S_3$$
$$S_1 ::= t_{1/2}\ t_{2/4}\ S_4$$

In the general algorithm, each FP implies the computation of one meet for each value in the domain of S_1. Alternatively, the FPs may be grouped in an effort to eliminate some of the redundant meets. For example, applying the productions as follows:

$$S_1 ::= t_{1/2}\ t_{2/3}\ S_3 \mid t_{1/5}\ t_5\ S_3 \mid t_{1/6}\ t_6\ S_3$$
$$S_1 ::= t_{1/2}\ t_{2/4}\ S_4$$

That is, by both leftmost and rightmost nonterminals. In this case, the form of applying an equation is similar to the general case; however, on each iteration, it is possible that fewer meets are performed, as in this case (or any case where there are three or more values are grouped). Another possibility is to group all the productions:

$$S_1 ::= t_{1/2}\ t_{2/3}\ S_3 \mid t_{1/5}\ t_5\ S_3 \mid t_{1/6}\ t_6\ S_3 \mid t_{1/2}\ t_{2/4}\ S_4$$

In this case, it is not necessary to take the meet with the current value of $\zeta(S_1)$ at each input point in the domain of the partial function. In a work list style algorithm, however, it has the disadvantage that unnecessary function computations may occur (in the worst case, when $\zeta(S_4)$ changes at some input, but $\zeta(S_3)$ has not, three unnecessary iterations of compute are performed, along with two meets).

Propagation of function inputs

During solution, each $\zeta(X)$ represents a partial function. The domain of this function value may grow as the solution is computed. Each element in the domain may be treated separately, inducing what are essentially many separate fixed point computations, or the input elements may be grouped in various ways. In the general algorithm, the entire partial function is treated as a unit, with computations occurring on all needed inputs simultaneously (select-value iterates across all values in the domain of the partial function). Other possibilities include grouping inputs that are: comparable, incomparable, or enter the domain on the same iteration.

Each of the approaches has advantages and disadvantages. Treating all inputs as a single unit is straightforward to implement, but has the disadvantage that values that have already reached a fixed point may continue to be recomputed unnecessarily. Individual propagation eliminates the unnecessary recomputation, but may involve an increased number of iterations over various parts of the grammar; it also requires finer-grained bookkeeping to ensure all required recomputations occur (that is, the select-prod and select-value phases would be re-arranged). The other possibilities offer trade-offs of varying degree, though all require additional bookkeeping and may imply computation that does not directly support finding the fixed point (e.g., comparing a new element to the existing

elements in the domain). Intuitively, treating the inputs that enter the domain at the same iteration is a reasonable compromise.

Work list vs. production iteration

As in the intraprocedural case, a work list (i.e., dynamic ordering) may be employed. The general method of iterating over the productions is both easy to implement and its correctness obvious. The latter has the possibility of significantly reducing the amount of computation needed to compute the fixed point. Consider, for example, that most programs have an initialization section that may reach a fixed point very quickly: if all productions are reevaluated, many recomputations for this section may occur. Care must be taken when implementing a worklist algorithm, however, to ensure that all required computations occur.

A re-working of the general algorithm into a worklist algorithm is shown in Figure 29. A FN X is added to the worklist when $\zeta[X]$ has changed, implying that all values for the dependents must be updated. In an actual implementation, select-prod need not iterate over all the productions in the grammar; an initialization phase can make a single pass over the grammar to compute a mapping from N to 2^P indicating which productions need reconsideration when a nonterminal changes value.

Various hybrid algorithms are also possible. For example, the flow system may be partitioned along "procedural" boundaries with each partition treated as a unit. A work list can be applied at one level, say the interprocedural level, and a production iteration at the other. This trades of bookkeeping for some possibly unnecessary recomputation.

It may be advantageous to compute a "fixed point" of a partition based on the current estimates from other partitions before actually adding values to the domains of called partitions. For example, consider a procedure call within a loop:

```
y := 0;
x := 3;
while (y<x) do begin
   y := y+1;
   call proc;
   x := x * 3
end;
```

If live variable analysis is being performed, for example, it is unnecessary to add {x} to the domain of the partial function representing `proc` when first encountering the procedure call. If the fixed point of the loop is computed first, assuming the current estimate of the flow value for `proc` (say T=∅), only the value {x,y} is added to the domain, eliminating a (possibly) unnecessary fixed point computation. Unfortunately, this is likely to be expensive in general. A promising alternative is to ensure that all dependents of each value have been computed at least once. In an ordering scheme, this may be effected by iterating through the equations twice before adding values to domains of called functions. In many cases, this should eliminate many of the unnecessary values.

Adding values to domains

At the initiate stage of the general iteration algorithm (Figure 28), the partial function for some FN has been found not to contain a needed value in its domain. In order to continue evaluation of the production currently being processed, a value must be chosen to represent the effect of the function, the so-called guess. By choosing the constant T_L, one is assured that the GFP is not missed. This may, however, induce a number of unnecessary computations, particularly if it causes the addition of a value to the domain of some partial

function that is not needed. Another approach is to guess a value based upon what is currently known about the partial function being queried

```
guess    newv ← ⊤L;
         for v′∈ dom(ζ[p[i]]) do
              if v≤Lv′ then newv ← newv ∧L ζ[p[i]](v′) fi
```

It simply computes an estimate based on those inputs whose GFP solution is known to be above the solution for the current input.

Complexity and the need for limiting computation

For the sake of completeness, we briefly analyze the time complexity of the main variants of the above algorithms. Let G = ‹V, T, P, S› be a flow grammar and δ be a terminal mapping for an instance of a MDFAF ‹L,$\wedge_L$,F›. Then the worst case time complexity of the first algorithm (Figure 28) may estimated as follows:

```
   O(|V|)                              cost of initialization loop
+ (   O(|P|)                           cost of the loop labeled select-prod
    * O(|L|)                           cost of the loop labeled select-value
    * (O(F)+O(E)+O(1)+O(M))            cost of the compute through insert steps
    * O(|L|) * O(|V|) *
      * (O(height(L))     )            cost of outer loop in the worst case
```

The O(F), O(E) and O(M) components refer to the costs of computing terminal functions, equality and meets of the lower level lattice. Moreover, the final three cost components result from the possible worst case behaviour of the outermost loop: in the worst case all values are re-computed for every point in the program at every lattice element and only one changes. For a given point at a given input, this can occur at most height(L) times. Note that the loop where all exits are found (**initiate**) has been considered constant time in this analysis because all procedures may be viewed as having (pre-computed) single exits.

It should be noted, however, that this is quite pessimistic in general: for most problems the set of inputs needed at each point is not likely to approach the size of the lower level lattice. The main program, for example, is only computed at a single value. Though not shown here, the worst case time complexity of the second algorithm (Figure 29) is the same.

Because the potential height of the partial function lattices (O(|L|) * O(height(L))) in some instances implies that considerable computation may be needed to find even the GFP solution, it may be necessary to place limits on the fixed point computation. One solution is to limit the types of lattices that the user may specify. This is the route taken in the Z1 system [97]: the set of allowable lattices is limited to ensure polynomial height, and thus polynomial time fixed point computation. Other possibilities include adding variants of widening operators to the computation [18].

Another easily adapted strategy is to limit the domain of each partial function to, say, I inputs. When the limit is reached, $\perp$ is added to the domain of the partial function. From that point on, all requests for new values use an existing approximating value. Such a value is guaranteed to be present because the input $\perp$ approximates everything, and $\forall x \in L.\zeta[X](\perp) \leq \zeta[X](x)$. The limit may simply be a constant. Or perhaps some function determinable from the FG, say the number of call sites to a given procedure, can be used to limit domains of the functions independently.

Using these strategies eliminates the restriction that the underlying lattice be finite. Obviously, the penalty is that the fixed point found may not yield the required information. By allowing the user to manipulate the various parameters, a (hopefully) acceptable tradeoff between running time and solution precision can be attained.

V.6 Summary

Two main contributions of this chapter are the formalization of flow grammar interpretations and algorithms for solving the resulting problems. An interpretation of RFGs admitting bidirectional data flow was introduced. A technical result showing that the granularity of a FG affects the solution obtained in the presence of bidirectional information flow. Lattices of total functions were explored to allow the development of an interpretation suitable for CFGs representing interprocedural control flow. To solve the problem of determining flow within individual procedures, partial function lattices were examined. This had the additional benefit of reducing the amount of computation needed to find fixed points. Separate forward and backward interpretations of CFGs were defined. It was determined that intraprocedural problems differ fundamentally from interprocedural flow problems in the presence of bidirectional information flow. Finally, several FG-based algorithms for solving interprocedural flow problems were given. The next chapter illustrates the use of these frameworks to solve various problems. Appendix A details a number of the practical issues involved in implementing fixed point algorithms.

Figure 28 Interprocedural backward fixed point iteration

Input: instance $(G=\langle V,T,P,S\rangle, \delta[\![\bullet]\!])$ of MDFAF $\langle L,\wedge_L,F\rangle$, with initial value init

Output: $\zeta: V \to (L \to L)$

Method:

```
initialize      foreach X∈V do
                    if S ℛX then ζ[X] ← {(init,init)} else ζ[X] ← ∅ fi
                endfor;
                repeat convergence ← true;
select-prod         foreach p∈P do
                        let X=p[0], Y=p[n_p]
select-value            for v∈dom(ζ[Y]) do
                            newv ← v;
compute                     for i=n_p-1 to 1 do
                                if p[i]∈T then
                                    newv ← δ[[p[i]]] (newv)
                                else if v∈dom(ζ[p[i]]) then
                                    newv ← ζ[p[i]] (newv)
                                else
initiate                            convergence ← false;
                                    foreach Z such that p[i] ℛZ do
                                        if newv∉dom(ζ[Z]) then
                                            ζ[Z] ← ζ[Z] ∪ {(newv,newv)}
                                        fi
                                    endfor;
guess                               newv ← ⊤_L
                                fi
                            endfor;
                            if v∈dom(ζ[X]) then
                                if newv ≠ ζ[X](v) then
update                              convergence ← false;
                                    let oldv=ζ[X](v)
meet                                ζ[X] ← ζ[X]![(newv∧_L oldv)/v]
                                fi
                            else
insert                          convergence ← false;
                                ζ[X] ← ζ[X] ∪ {(v,newv)}
                            fi
                        endfor
                    endfor
                until convergence
```

Figure 29 Interprocedural backward fixed point worklist algorithm

```
initialize      worklist ← ∅;
                foreach X∈V do
                    if S ℛX then
                        ζ[X] ← {(init,init)}; worklist ← worklist ∪ {X}
                    else
                        ζ[X] ← ∅
                    fi
                endfor;
                while worklist≠∅ do
                    let W be some member of worklist;
                    worklist ← worklist - {W} for some W∈worklist;
select-prod         foreach p∈P such that W∈{p[i] | 1≤i≤n_p} do
                        let X=p[0], Y=p[n_p]
select-value            for v∈dom(ζ[Y]) do
                            newv ← v;
compute                     for i=n_p-1 to 1 do
                                if p[i]∈T then
                                    newv ← b[[p[i]]] (newv)
                                else if newv∈dom(ζ[p[i]]) then
                                    newv ← ζ[p[i]] (newv)
                                else
initiate                            foreach Z such that p[i] ℛZ do
                                        if newv∉dom(ζ[Z]) then
                                            ζ[Z] ← ζ[Z] ∪ {(newv,newv)};
                                            worklist ← worklist ∪ {X}
                                        fi
                                    endfor;
guess                               newv ← ⊤_L
                                fi
                            endfor;
                            if v∈dom(ζ[X]) then
                                if newv ≠ ζ[X](v) then
update                              let oldv=ζ[X](v)
meet                                ζ[X] ← ζ[X][(newv∧_L oldv)/v];
                                    worklist ← worklist ∪ {X}
                                fi
                            else
insert                          ζ[X] ← ζ[X] ∪ {(v,newv)};
                                worklist ← worklist ∪ {X}
                            fi
                        endfor
                    fi
                endfor
            endwhile
```

VI Applications

This chapter demonstrates the generality of the flow grammar methodology introduced in Chapters IV and V by applying flow grammars to real data flow analyses. In doing so, the final problems outlined in Section I.3 are addressed. The main contributions are: original adaptations of a broad range of useful analyses to flow grammar frameworks and the demonstration of several techniques that can be used in the flow grammar realm to solve common difficulties arising in flow analysis problems. Appendix A documents the implementation effort. Taken together, these adaptations confirm the utility and practicality of the flow grammar model: in spite of the varying representations traditionally used to perform these analyses, they may all be performed within the flow grammar framework. Such generality is important for the technique to be used as the basis of FACT.

VI.1 Introduction

This chapter describes several analyses that have been constructed with a prototype implementation of the FACT tool. Techniques for using the FG framework are presented, dem-

onstrating how various results from the field of data flow analysis may be used. Specific analyses include interprocedural live variables, intraprocedural partial redundancy elimination [62], two aliasing analyses and an interprocedural procedure variable analysis:

- *interprocedural live variables* – the live variables problem has been used throughout this dissertation to demonstrate various aspects of the FG approach to data flow analysis. Section VI.2 uses a result of Knoop and Steffen [52] to perform interprocedural live variable analysis that takes into account the effect of dynamically allocated local variables. This is an important problem for performing a number of tasks, including: storage allocation, register allocation, and code improving transformations (e.g., removal of unnecessary saves of registers to memory) [3]. The extension to interprocedural analysis alleviates the need to make conservative assumptions at procedure calls.
- *aliasing analysis* – in languages that permit aliasing (that is, multiple names for a single object) it is crucial to determine the possible aliasing relationships [20, 55, 56]. Even keeping a value in a register across statements may be unsafe if some form of aliasing analysis is not performed.
- *partial redundancy elimination* – this is actually an optimization technique that happens to be solvable as a set of flow analyses [63]. By both eliminating redundant computations and moving invariant computations out of loops [3], it helps reduce the amount of time spent performing optimization.
- *procedure variable analysis* – performing interprocedural analysis is complicated considerably by the presence of procedure variables, also known as *function pointers*. In the absence of other information, any call site involving a procedure variable it must be assumed to call any procedure the source language would permit at that point. This can severely curtail the effectiveness of interprocedural analysis, particularly in a language such as C [20, 32].

These analyses are representative of the types of analyses for which the FACT tool is intended. They provide valuable information to the optimizer and are based on information that is either readily available or easily computed in an attribute grammar based compiler.

VI.2 Classical flow analysis re-visited

Hecht has classified a number of problems as "classical" flow analysis [29]. These include *live variables, reaching definitions, available expressions* and *very busy expressions*. The intraprocedural versions of these problems all have the common characteristic that the abstract state is a finite set that may be represented as a bit string. Sharir and Pnueli developed an initial interprocedural extension [78] that was subsequently strengthened by Knoop and Steffen [52]. This section overviews the traditional approach to solving these problems, and then presents a flow grammar approach that embodies the work of Knoop and Steffen. The live variables problem is used as the running example, although the techniques apply to the other classical analyses as well.

VI.2.1 Problem: live variables

Live variables, described in Sections IV.5.1 and V.4.2, is a backward analysis that associates a truth value with each variable at each point in the program. Recounting the basic idea, a variable is *live* at some point P in a program if the value it has at P may be used again before being assigned a new value. The set of variables live at the end of a program is typically the empty set.

VI.2.2 Example

Figure 30 shows a small Pascal program and corresponding node-based control flow graph. Each node is labelled with the set of variables live on entry to the node (IN) and exit from the node (OUT), as would typically be computed using standard intraprocedural flow analysis techniques [3, 29]. This example illustrates the kinds of optimizations that are made possible by live variable analysis. First, notice that `f` and `i` are never live simul-

taneously and may consequently be allocated to the same memory. Second, it is very likely that `s` would be allocated memory (for use during the execution of the `readln` and `writeln`) and assigned to a register for the duration of the two loops. After execution of the second loop the value of s would normally be stored back into its assigned memory location. However, because it is not live after executing the second loop (IN of the `writeln` node), this store may be safely deleted.

VI.2.3 Standard flow equations

The standard flow equations for live variable analysis follow the usual pattern described in Section II.3.5 for a backward problem. As stated previously, the information space is the powerset of the variables in the program with set union as the meet operator, giving the following structure for each node N in the control flow graph.

$$N.IN = f_N(N.OUT)$$

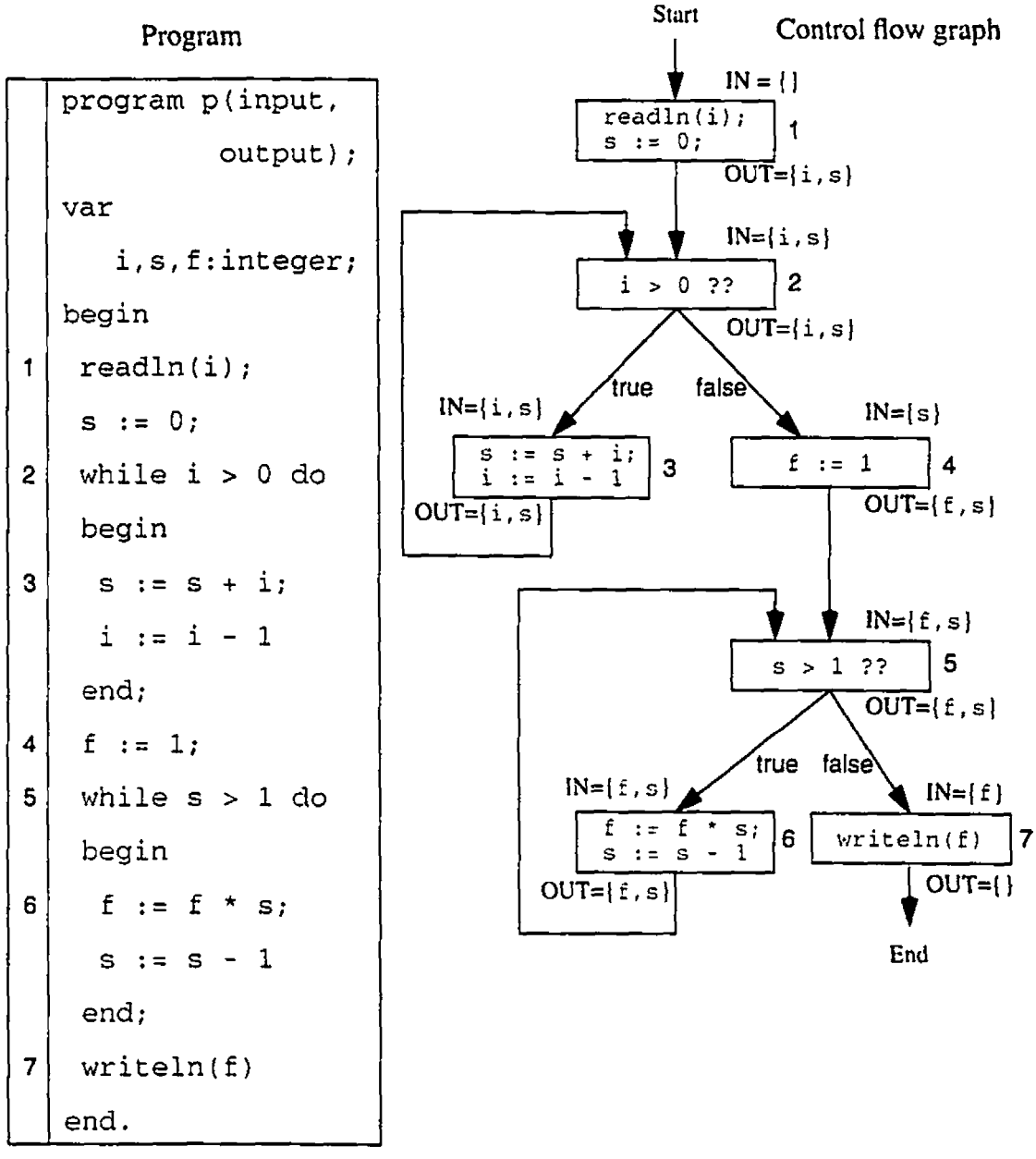
$$N.OUT = \bigcup_{M \in succ(N)} M.IN$$

where f_N computes the change in live variables due to node N; and N.OUT = { } for all nodes N that have no successors (that is, nodes representing exits).

VI.2.4 Discussion

Live variables is typically performed intraprocedurally as shown in the preceding section. The following section shows how to perform the same intraprocedural analysis in the FG framework. Subsequent sections extend the live variables problem to handle procedure calls, including a discussion of parameters and local variables.

Figure 30 Example of traditional intraprocedural live variable analysis



VI.2.5 FG interpretation: intraprocedural live variables

An intraprocedural FG interpretation involves defining a MDFAF $\langle L,\wedge_L,F\rangle$ and then, for a given program, an instance of a BraFGP. Let Var be the set of variables in the program, then $L = 2^{Var}$, $\wedge_L = \cup$, and F is the compositional closure of the following set of functions:

$$\{\ \lambda s.s-\{i\} \mid i \in Var\ \} \cup \{\ \lambda s.s\cup\{i\} \mid i \in Var\ \} \cup \{\ \lambda s.Var\ ,\ \lambda s.\{\}\ ,\ \lambda s.s\ \}$$

For completeness, the BraFGP for the program in Figure 30 is shown in Figure 31.

Figure 31 BraFGP for program in Figure 30

$$
G=\begin{array}{l}
S_1 ::= t_1\ S_2 \\
S_2 ::= t_2\ S_3 \\
S_3 ::= t_{3/4}\ S_4 \\
S_3 ::= t_{3/7}\ S_7 \\
S_4 ::= t_4\ S_5 \\
S_5 ::= t_5\ S_6 \\
S_6 ::= S_3 \\
S_7 ::= t_7\ S_8 \\
S_8 ::= t_{8/9}\ S_9 \\
S_8 ::= t_{8/12}\ S_{12} \\
S_9 ::= t_9\ S_{10} \\
S_{10} ::= t_{10}\ S_{11} \\
S_{11} ::= S_8 \\
S_{12} ::= t_{12}\ S_{13} \\
S_{13} ::= \varepsilon
\end{array}
$$

$$
\begin{array}{lll}
b[\![t_1]\!] = \lambda s.s-\{i\} & f[\![t_1]\!] = \lambda s.\{\} \\
b[\![t_2]\!] = \lambda s.s-\{s\} & f[\![t_2]\!] = \lambda s.\{\} \\
b[\![t_{3/4}]\!] = \lambda s.s\cup\{i\} & f[\![t_{3/4}]\!] = \lambda s.\{\} \\
b[\![t_{3/7}]\!] = \lambda s.s\cup\{i\} & f[\![t_{3/7}]\!] = \lambda s.\{\} \\
b[\![t_4]\!] = \lambda s.s\cup\{i,s\} & f[\![t_4]\!] = \lambda s.\{\} \\
b[\![t_5]\!] = \lambda s.s\cup\{i\} & f[\![t_5]\!] = \lambda s.\{\} \\
b[\![t_7]\!] = \lambda s.s-\{f\} & f[\![t_7]\!] = \lambda s.\{\} \\
b[\![t_{8/9}]\!] = \lambda s.s\cup\{s\} & f[\![t_{8/9}]\!] = \lambda s.\{\} \\
b[\![t_{8/12}]\!] = \lambda s.s\cup\{s\} & f[\![t_{8/12}]\!] = \lambda s.\{\} \\
b[\![t_9]\!] = \lambda s.s\cup\{f,s\} & f[\![t_9]\!] = \lambda s.\{\} \\
b[\![t_{10}]\!] = \lambda s.s\cup\{s\} & f[\![t_{10}]\!] = \lambda s.\{\} \\
b[\![t_{12}]\!] = \lambda s.s\cup\{f\} & f[\![t_{12}]\!] = \lambda s.\{\}
\end{array}
$$

$$\upsilon = \{\}$$

VI.2.6 FG interpretation: interprocedural live variables

The absence of liveness information across a procedure call can limit the optimization of a program. For example, all variables currently being held in registers must typically be written to memory prior to a call. In this case, interprocedural analysis may reveal that

some variables are dead and the store eliminated. While a straightforward interpretation using the intraprocedural lattice (given in the preceding section) as the base is possible, some source language constructs, such as automatically allocated local variables, may benefit from a more elaborate analysis. Subsequent sections show how Knoop and Steffen's Interprocedural Coincidence Thereom [52] may be employed in FG interpretations for handling automatic locals and three parameter passing mechanisms. Knoop, Steffen and Rüthing use their result in essentially the same way in their recent "flow analysis tool kit [53]."

VI.2.7 Local variables

Most imperative programming languages allow the declaration of automatically allocated local variables within procedures. Each time a procedure is called, a fresh set of local variables is created for use during the execution of the procedure. Upon return from the procedure, the memory for these variables is deallocated. The use of local variables in a program complicates flow analysis in a subtle way that requires care to ensure conservative, but not excessively imprecise, results. Throughout this section the live variables problem is used as the running example, though the techniques developed may be applied to other classical flow analyses with slight modification.

Leaf procedures

First consider the straightforward case of a non-recursive procedure containing a local variable such as the procedure `p` in Figure 32. We make the following observations about the variable `local`:

1. At the end of an execution of `p`, the variable `local` is deallocated and thus cannot be live on exit from the procedure. That is, in a live variable analysis,

local variables must be eliminated from the current set of live variables when a procedure exit is encountered.

2. On entry to procedure p a new incarnation of the variable `local` is created. Prior to its creation, it cannot be live (i.e., a variable that does not exist cannot be live). Thus, the abstract semantics must also eliminate the set of local variables from the abstract state on entry to a procedure.

Thus, the following function suffices for representing the effect of entering and exiting p in this case:

$$f[\![t_{call\ p}]\!] = f[\![t_{exit\ p}]\!] = \lambda x.x - \{\texttt{local}\}$$

Using these definitions does, indeed, yield a conservative solution to the set of flow equations implied by this program.

Figure 32 Program containing a local variable

```
                program localvar1;
                  var global1,global2:integer;
                  procedure p;
                    var local:integer;
                  begin
1                   if (local <> 0) then begin
2                     global1 := global1 div 2;
3                     local := global1;
4                     global2 := local
                    end
5                 end;
                begin
6                 readln(global1);
7                 p;
8                 write(global2)
9               end.
```

Recursive procedures

Now consider a slight re-writing of the example program in which a single recursive call has been added, as shown in Figure 32. Applying the previous semantic functions results in the following equation for $\mathcal{K}_{S4}$:

$$\mathcal{K}_{S4} = (\lambda x.x\text{-}\{\texttt{local}\}) \circ \mathcal{K}_{S1} \circ (\lambda x.x\text{-}\{\texttt{local}\}) \circ \mathcal{K}_{S5}$$

Because `local` is used in the assignment immediately following point 5, it is clearly live at point 5, regardless of calling context. Applying the equation above clearly eliminates `local` from the set of variables live at point 4. Unfortunately, closer examination of the program reveals that `local` is in fact live at point 4. This conclusion may be derived from the following observations:

1. When `p` is called, a fresh version of `local` is created. Thus, any assignment to `local` in a recursive call cannot affect a suspended version of the variable.
2. There is no aliasing in this program, thus assignments to `global1` and `global2` cannot affect any incarnation of `local`.

Because there is no way for the recursive call to affect the incarnation of `local` visible at point 5, and `local` is live at that point, the same incarnation of `local` *must* be live at point 4.

This analysis suggests a possible solution: when a recursive call is encountered, the liveness of each local variable must be "remembered" so that it may be restored after the abstract effect of the procedure call has been computed. An easy way of effecting this is the use of *shadow* (called *dormant* in [15]) variables: each local variable has an associated shadow used to save the state of the local across a call to the procedure containing the local. Knoop and Steffen have shown that a single shadow is sufficient to guarantee a conservative result for the live variables program, in which the MOP solution coincides with the GFP solution [52].

Figure 33 Program containing a recursive procedure with a local variable

```
                program localvar2;
                  var global1,global2:integer;
                  procedure p;
                    var local:integer;
                  begin
1                   if (local <> 0) then begin
2                     global1 := global1 div 2;
3                     local := global1;
4                     p;
5                     global2 := local
                    end
6                 end;
                begin
7                 readln(global1);
8                 p;
9                 write(global2)
10              end.
```

Adding shadow variables has two main effects on the analysis: the lattice is richer and the flow equations for procedure entry and exit no longer simply remove the set of local variables. If V is the set of variables in a program, let:

$$V_s = \{\, v' \mid v \in V, v \text{ not global} \,\} \text{ such that } V \cap V_s = \varnothing$$

be a new set of shadow variables and:

$$V' = V \cup V_s$$

Then the new live variable lattice is $2^{V'}$. For the recursive example program, V′ = {global1, global2, local, local′}.

The effect of procedure exit may be computed as follows:

$$\delta[\![t_{exit\ p}]\!] = \lambda x.\text{if } (\texttt{local} \in x) \text{ then } (x - \{\texttt{local}\}) \cup \{\texttt{local}'\} \text{ else } x$$

That is, if local is live at exit, this must be remembered (in local′) and local must be removed from the set of live variables, since the new incarnation cannot possible be live. Computing the effect of procedure entry effectively reverses the effect of exiting:

$$\delta[\![t_{call\ p}]\!] = \lambda x.\text{if } (\texttt{local}' \in x) \text{ then } ((x - \{\texttt{local}'\}) \cup \{\texttt{local}\}) \text{ else } (x - \{\texttt{local}\})$$

If the previous incarnation of local is live after the call, this ensures that it is again alive before the call.

VI.2.8 Parameters

Given the above solution to local variables, allowing certain types of parameters requires little extra effort. Value parameters, result and value/result parameters can all be handled in the flow grammar framework by modeling them as local variables with appropriate assignment statements. Of these three types of parameter, value parameters are by far the most common, occurring in Pascal (default mechanism), C (default), and Ada ("in" parameters) and are considered in some detail below. Modeling the other two mechanisms is then shown to be similar.

Value parameters

A parameter passed by value bears a strong resemblance to an initialized local variable. Both are created upon entry to a procedure and both are given an initial value. The difference is that one is initialized with a value from the caller while the other receives a locally defined value. Figure 34 shows a small example program. At run time, the call at point 8 performs the following actions:

1. A new instance of `valueparam` is created.
2. An assignment is made from `global1` to this new instance of `valueparam`:
   ```
   valueparam := global1;
   ```
3. A new instance of `local` is created and assigned the value 10:
   ```
   local := 10;
   ```
4. The body of the procedure is executed in this new environment.

Note that this can easily be generalized to multiple parameters and local variables. An abstract semantics can be derived directly from the above observations. The only remaining difficulty arises if a local variable is passed in a recursive call, as at point 4. In this case, care must be taken to affect either the "current" or its "shadow," as appropriate.

Figure 34 Program with initialized local variable and value parameter

```
          program localvar2;
            var global1,global2:integer;
            procedure p(valueparam:integer);
1             var local:integer := 10;
            begin
2             if (local <> valueparam) then begin
3               global1 := global1 div 2;
4               p(local)
              end else
5               global2 := - global1;
6           end;
          begin
7           readln(global1);
8           p(global1);
9           write(global2)
10        end.
```

To make these ideas more concrete, the entry and exit terminal flow functions for live variable analysis of the two call sites in the program are as follows. For the call site at point 4:

$$b[\![t_{4/1}]\!] = \lambda x.(x - \text{locals}) \cup \text{shadows} \cup \text{actuals}$$

where

locals = {local, valueparam}

shadows = (if local′ ∈ x then {local} else {})

$$\cup(\text{if } \texttt{valueparam}' \in x \text{ then } \{\texttt{valueparam}\} \text{ else } \{\})$$

actuals = {local}

$$\mathcal{G}[\![t_{6/6}]\!] = \lambda x.(x - \text{locals}) \cup \text{shadows}$$

where

locals = {local, valueparam}

shadows = (if local ∈ x then {local'} else {})
$$\cup(\text{if } \texttt{valueparam} \in x \text{ then } \{\texttt{valueparam}'\} \text{ else } \{\})$$

And for the call site at point 8:

$$\mathcal{G}[\![t_{8/1}]\!] = \lambda x.(x - \text{locals}) \cup \text{shadows} \cup \text{actuals}$$

where

locals = {local, valueparam}

shadows = (if local' ∈ x then {local} else {})
$$\cup(\text{if } \texttt{valueparam}' \in x \text{ then } \{\texttt{valueparam}\} \text{ else } \{\})$$

actuals = {global1}

$$\mathcal{G}[\![t_{6/9}]\!] = \lambda x.(x - \text{locals}) \cup \text{shadows}$$

where

locals = {local, valueparam}

shadows = (if local ∈ x then {local'} else {})
$$\cup(\text{if } \texttt{valueparam} \in x \text{ then } \{\texttt{valueparam}'\} \text{ else } \{\})$$

Result and value/result parameters

In some sense, result parameters are the mirror of value parameters. Instead of being assigned a value upon creation and its value forgotten on deallocation, a result parameter is not implicitly initialized but its value is passed back to the caller on exit from a procedure. Thus, their abstract semantics may be handled similarly. At the same time this also allows the framework to handle functions, since the return value of a function is equiva-

lent to a result parameter. Semantically, value/result parameters are a combination of value and result parameters, and may be treated by combining the two techniques.

Reference parameters

Because reference parameters introduce aliases, they cannot directly be handled by the above techniques. They may, however, be handled by the points-to analysis described in the next section by observing that reference parameters are merely syntactic sugar pointer manipulation.

VI.3 Aliasing

Aliasing, whereby a single value has more than one name within a program, has long been the nemesis of optimizing compilers in general, and flow analyzers in particular [3, 11, 12, 15, 32, 64]. It can arise from a number of programming language constructs including arrays, reference parameters, pointers and equivalence statements.

As a starting point, Aho, Sethi and Ullman's "simple pointer language" and corresponding aliasing algorithm [3, pp. 649-652] was implemented in the FACT prototype. The five rules given for determining the effect of assignments with respect to aliasing were easily translated into attribution rules for constructing appropriate FT mappings in the FACT prototype.

A second, more interesting analysis, called *points-to analysis*, has also been implemented using the prototype. Points-to analysis, devised by the McCAT group at McGill University for analyzing C programs, involves the computation of a points-to relation for each point in the program [20, 32]. The variant discussed here is primarily concerned with the aliasing relationships that occur among variables on the run-time stack (as opposed to

values stored in the heap).[1] Given a points-to relation, a conservative estimate of possible alias pairs may be computed by performing a closure operation. The following sections describe this effort and how the FG realization differs from the original.

VI.3.1 Problem: points-to analysis

Points-to analysis computes, for each point in the program, a relation indicating possible relationships among the variables present at that point. A given relationship may be *definite*, indicating that the relationship always hold when control reaches the point, or *possible*, indicating that the relationship may hold when control reaches the point. This is a forward problem bearing some resemblance to constant propagation and range analysis. Only relationships among stack variables are tracked explicitly. Array elements are not tracked separately, but as single units. Similarly, the heap is treated as a large anonymous array, on the assumption the pointers from the heap back to the stack are rare [20].

Points-to relations

Points-to is a ternary relation indicating how variables point to one another during execution as shown below:

1. points-to(x,y,D) holds at some point in the program when x *definitely* points to y. This can arise, for example, after the following assignment:

```
x = &y;
```

2. points-to(x,y,P) holds at some point in the program when x *possibly* points to y. This can arise when control flow paths merge, such as at `S1` below:

1.This analysis may be viewed as a store semantics using a non-standard domain of "abstract stacks."

```
        if (cond)
          x = &y;
S1:
```

In both cases x is said to be the *source* of the relationship and y is said to be the *target*. The analysis associates a points-to relation with each program point, taking into account the effect of assignments between program points and ensuring that conservative approximations are made at control flow merges.

VI.3.2 Example

Figure 35 shows a small C program from Emami's thesis [20, p. 84] which illustrates a number of features of points-to analysis. At the beginning of the program nothing points to anything ({ } at point 1). After the first assignment (between points 1 and 2) a definitely points to b, and after the second b definitely points to c. In Emami's implementation, analyzing the call to f causes the creation of a new *abstract stack* with the extra "hidden" variables 1_m and 2_m, representing two possible levels of dereference of m (m is a pointer to a pointer to an int). A mapping between these values and out-of-scope variables is created so that the effect of the call may be deduced on exit. In this case, the map contains two pairs: (1_m, b) and (2_m, c) indicating that, on entry, m points to b which points to c. Thus, at point 5, m definitely points to 1_m and 1_m definitely points to 2_m. Assigning the address of x, a global variable, to n yields the obvious state at point 6. The next assignment, "*m = &x;" causes a dereference through m to 1_m, which now points to x, too. The abstract effect of exiting the procedure is to "unmap" the effects of the call on 1_m and 2_m to the variables indicated in the previously computed map. Thus, after the call, b (represented by 1_m during this particular call) now points to the global x. The reader is directed to Emami's thesis for justification of this technique.

Figure 35 C program with aliasing [20, Figure 4.13, p. 84]

Point	Points-to relation	Program
		`int x;`
		`main()`
		`{`
		`int **a, *b, c;`
1	{ }	`a = &b;`
2	{ (a,b,D) }	`b = &c;`
3	{ (a,b,D),(b,c,D) }	`f(a);`
4	{ (a,b,D),(b,x,D) }	`}`
		`void f(int **m)`
		`{`
		`int *n;`
5	{ (m,1_m,D), (1_m,2_m,D) }	`n = &x;`
6	{ (n,x,D), (m,1_m,D), (1_m,2_m,D) }	`*m = &x;`
7	{ (n,x,D), (m,1_m,D), (1_m,x,D) }	`}`

VI.3.3 Standard flow equations

There are two main components to Emami's formulation, which is based on the McTAG analysis framework [81]. The intraprocedural part is of the gen/kill type [3]. That is, each flow function is of the form:

$$\lambda s \,.\, (s - \text{kill}) \cup \text{gen}$$

where gen and kill depend on the basic statement the function represents. The interprocedural part involves defining the map and unmap functions, along with constructing the

invocation graph. Each node in this graph represents a call chain. Flow information for each invocation graph node is merged as analysis proceeds; the map information, however, permits considerable disambiguation in the caller.

The following sections detail the gen and kill sets as applied to the FG framework. In particular, the relation is cast into a lattice in the next section, and then the intra- and interprocedural FG adaptation of the analysis is described.

VI.3.4 Points-to lattice

Not surprisingly, the points-to relation may be viewed as a lattice. The elements of the lattice are the relations that may arise during analysis, represented as sets of triples. More precisely, let Var be the set of "variables" in some program P. A *points-to triple* is a member of the cross-product Var×Var×Rel, where Rel={D,P} represents the *certainty* of the association (where D ≡ "Definite", P ≡ "Possible"). It is useful to consider the Rel set as a lattice with the following meet operator:

$\wedge_{Rel}$	D	P
D	D	P
P	P	P

(Note that this is equivalent to Boolean conjunction with D representing "true" and P representing "false.")

On first glance, one would hope that the *points-to lattice*, L_{p_t}, would simply be a powerset, such as $2^{Var \times Var \times Rel}$. Unfortunately, this admits contradictory information, such as {‹x,y,D›, ‹x,z,D›} where y≠z, which implies that x definitely points to two distinct enti-

ties, y and z. Thus the elements of L_{Pt} are restricted to a subset of $2^{Var \times Var \times Rel}$. First, define the following functions:

$$Def(S{:}2^{Var \times Var \times Rel}) = \{ \langle x,y,D \rangle \mid \langle x,y,D \rangle \in S \}$$
– all "definite" points-to triples

$$Pos(S{:}2^{Var \times Var \times Rel}) = \{ \langle x,y,P \rangle \mid \langle x,y,P \rangle \in S \}$$
– all "possible" points-to triples

Now define the following predicate:

$$OkayPt(S{:}2^{Var \times Var \times Rel}) = \forall x \in \{ x \mid \langle x,y,D \rangle \in Def(S) \}. \left| \{ \langle x,y,r \rangle \mid \langle x,y,r \rangle \in S \} \right| = 1$$

That is, an element of $2^{Var \times Var \times Rel}$ represents a feasible points-to relation if all variables x that definitely point to an object point to a *single* object.

A possible meet operator for sets of such triples is Emami's Merge function [20, pp. 49], defined as:

$$Merge(S_1,S_2) = Dset \cup Pset$$
where $Dset = \{ \langle x,y,D \rangle \mid \langle x,y,D \rangle \in [Def(S_1) \cap Def(S_2)] \}$
$Pset = \{ \langle x,y,P \rangle \mid \langle x,y,r \rangle \in [(S_1 \cup S_2) - Dset] \}$

That is, when two sets from different paths are to be merged, if x definitely points to y on both paths, then it still definitely points to y after control flow has merged (Dset). The remaining triples represent information that may hold, and are thus regarded as "possibly points-to" relationships (Pset).

Now suppose the Merge function were used as the meet operator and consider the concomitant partial order relation:

$$S_1 \leq S_2 \text{ iff } S_1 = \text{Merge}(S_1, S_2)$$

Unfortunately, the resulting partial order has no top element. Clearly, when two sets of triples are merged the resulting set is at least as large as the larger of the two input sets. Thus, one would hope that the empty set is the top element. It is easy to see that this is not the case:

$$\text{Merge}(\{ \langle x,y,D \rangle \}, \varnothing) = \varnothing \cup \{ \langle x,y,P \rangle \} = \{ \langle x,y,P \rangle \}$$

(That is, if a given definitely points-to relationship is present on only one path, it must be converted to a possibly points-to relationship to ensure safety.) So, $\{ \langle x,y,D \rangle \} \not\leq \varnothing$ and $\varnothing \not\leq \{ \langle x,y,D \rangle \}$. Thus, in addition to the set of "okay" triples defined by OkayPt above, an artificial top element, T_{PT}, is added and the Merge operator is extended in the obvious manner. It should be noted that Emami adds a similar "special" element, which she calls BOTTOM, to handle exits from loops: after a break or continue statement, the constant BOTTOM is used to indicate no further propagation of information. In the FACT formulation, the requirement for the element is a natural consequence of the FG framework.

Thus, the elements of the final intraprocedural points-to lattice for a given set of variables, Var, consists of the following elements:

$$L_{PT} = \{ S \mid S \in 2^{Var \times Var \times Rel}, \text{OkayPt}(S) \} \cup \{ T_{PT} \}$$

VI.3.5 Intraprocedural analysis

In Emami's thesis, intraprocedural control flow is considered in two separate components: "basic statements" which affect the points-to information directly and "compositional control flow statements" which involve the merging of information. Basic statements are generally assignment statements while control flow statements include conditionals and

loops. The abstract effect of both types of statement are specified directly, including the fixed point computation needed to compute the abstract effect of loops. It should be noted that the analysis relies on an intermediate representation, called SIMPLE, which is essentially a subset of C similar to three address code. An important characteristic of SIMPLE is that it does not allow arbitrary transfer of control (goto statements). Gotos in the original source program have been eliminated implying that the control flow is compositional [22], admitting a straightforward specification of control flow effects.

Mapping basic statements

In FACT, basic statements are represented by FTs. A typical assignment statement relevant to points-to analysis might be:

```
/* assign the address of y to what x points to */
x := &y;
/* now x "definitely points-to" y */
```

Emami identifies six basic variations of such assignment statements that can change points-to relationships, shown in Figure 36. The semantics of these cases follow the same basic pattern:

λinput.if input = T_{PT} then T_{PT}
else (gen $\cup$ (change - kill))
where
kill = kill_func(input)
change = change_func(input)
gen = gen_func(input)

The specifics of the three functions, kill_func, change_func and gen_func, depend on the case being mapped:

1. kill_func – there are two possibilities for the kill function, one for direct assignments and the other for indirect assignments. In the direct assignment "x = ...," x is overwritten implying that all the objects to which it may have pointed prior to the statement may be eliminated:

$$\text{kill\_func}_{\text{"x = ..."}} = \lambda rels.\{ (x,x_1,rel) \mid (x,x_1,rel) \in rels\}$$

For an indirect assignment, "*x = ...," if x definitely points to some object, say x1, then x1 is guaranteed to be overwritten and its current set of possible values removed:

$$\text{kill\_func}_{\text{"*x = ..."}} = \lambda rels.\{ (x_1,x_2,rel) \mid (x,x_1,D) \in rels, (x_1,x_2,rel) \in rels\}$$

2. change_func – this function transforms definite relationships to possible relationships when their certainty comes into question due to an indirect assignment. As there is no uncertainty as to which variable is being modified in a direct assignment, the corresponding change function is simply the identity function:

$$\text{change\_func}_{\text{"x = ..."}} = \lambda rels.rels$$

Now consider an indirect assignment, "*x = ..." If x possibly points to x1 and x1 definitely points to x2 prior to such an assignment, there is no guarantee that x1 still points to x2 after the assignment. Thus, the certainty of x1 pointing to x2 must be lowered to possible:

$$\text{change\_func}_{\text{"*x=..."}} = \lambda rels.\ (rels - \{ (x_1,x_2,D) \mid (x,x_1,P),(x_1,x_2,D) \in rels \}) \cup \{(x_1,x_2,P) \mid (x,x_1,P),(x_1,x_2,D) \in rels\}$$

3. gen_func – each basic assignment generates new points-to information that may, or may not, depend on the set of relationships that hold immediately prior to execution of the assignment statement. First, consider the assignment "x = &y". Clearly, after executing this assignment, x must point to y, so:

$$\text{gen\_func}_{\text{"*x = \&y"}} = \lambda rels.(\{ (x,y,D) \})$$

The remaining cases may be derived using similar reasoning and are shown in Figure 37.

Figure 36 Basic statements for intraprocedural points-to analysis

	L-value	R-value	R-value dereference
direct assignment	`x = &y;`	`x = y;`	`x = *y;`
indirect assignment	`*x = &y;`	`*x = y;`	`*x = *y;`

Figure 37 Basic statement generation functions

Basic Statement	Relationship generation function
`x = &y;`	λrels.({ (x,y,D) }) – assign address of x to y (i.e., x points to y)
`x = y;`	λrels.({ $(x,y_1,rel) \mid (y,y_1,rel) \in rels$ }) – assign y to x (i.e., x can now point to whatever y could)
`x = *y;`	λrels.({ $(x,y_1,rel_1 \wedge_{Rel} rel_2) \mid (y,y_1,rel_1), (y_1,y_2,rel_2) \in rels$ }) – dereference y and assign to x (i.e., must get the possible values of y and then "dereference" them – x can point to any of the resulting values)
`*x = &y;`	λrels.({ $(x_1,y,rel) \mid (x,x_1,rel) \in rels$}) – assign address of y to what x points to (i.e., things x could point to now can point to y)
`*x = y;`	λrels.({ $(x_1,y_1,rel_1 \wedge_{Rel} rel_2) \mid (x,x_1,rel_1),(y,y_1,rel_2) \in rels$ }) – assign y to what x points to
`*x = *y;`	λrels.({ $(x_1,y_2,rel_1 \wedge_{Rel} rel_2 \wedge_{Rel} rel_3) \mid (x,x_1,rel_1),(y,y_1,rel_2), (y_1,y_2,rel_3) \in rels$ }) – dereference y and assign to what x points to

Interprocedural analysis

As in the case of the classical analyses in Section VI.2, handling interprocedural control flow in FACT is primarily a matter of effecting the appropriate argument transmission and

ensuring safety in the presence of recursion. In fact, the technique of shadow variables may be used to advantage in this analysis.

In the absence of recursion, the analysis is straightforward. Entry to a function involves performing abstract assignments from the actuals to the formals. Similarly, exit from a procedure involves removing all points-to triples from the state whose first element is a local parameter or local whose lifetime has ended. In addition, on function exit it is possible to check for dangling references to such locals, such as when control returns from point 5 to point 2 in Figure 38.

Figure 38 A dangling reference at point 2 that should probably generate a warning

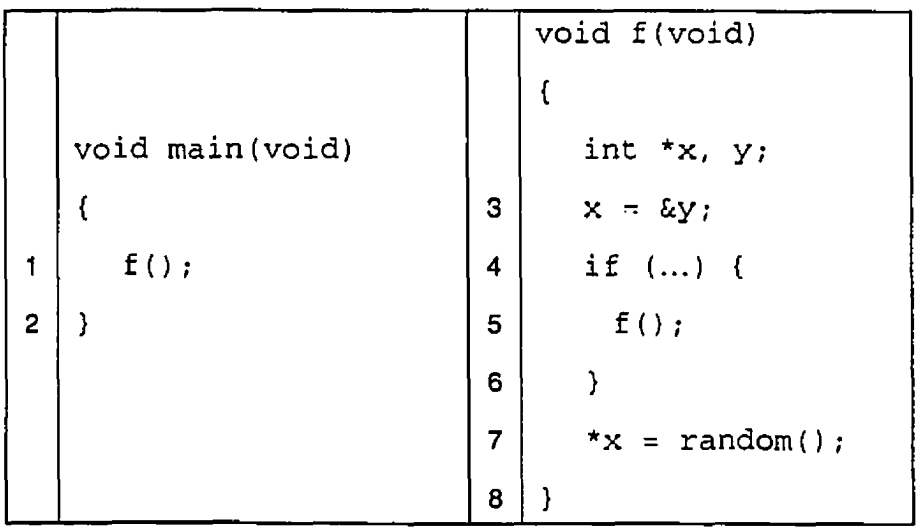
```
   int *x;                          |    void dangle(void)
   void main(void)                  |    {
   {                                |       int local;
1     make_dangle();                |4      x = &local;
2  /* {(x,?local?,D)} */            |5   /* {(x,local,D)} */
      *x = -559038737;              |    }
3  }                                |
```

This scheme fails in the presence of recursion, however. Specifically, consider the program in Figure 39. It is unsafe to delete the set of triples having x as the first component when control returns from point 5 to point 6. Although one incarnation of x is deallocated on function exit, there may be another incarnation from a previous invocation which still points to y: this relationship must still be represented in the abstract state.

An additional problem with the simple scheme is unnecessary lack of precision. One common analysis problem is the MOD problem which computes an approximation of

the set of variables that may be modified by a statement [16, 17, 72]. In the case of the example program, it must be concluded that the indirect assignment at point 7 can modify any incarnation of y, even though it is clear that only the current incarnation can actually be modified by the call.

Figure 39 Recursion prevents the elimination of points-to triples at function exit



```
                                    void f(void)
                                    {
   void main(void)                     int *x, y;
   {                              3    x = &y;
1     f();                        4    if (...) {
2  }                              5      f();
                                  6    }
                                  7    *x = random();
                                  8 }
```

Shadow variables for interprocedural analysis

Adding shadow variables to the lattice alleviates both the problems described in the previous section. Let Var′ = { v, v′ | v∈ Var, v not global, v′ unique new variable} be the augmented set of variables (cf. Section VI.2). As expected, using Var′ as the basis of the intraprocedural lattice allows the use of the intraprocedural functions defined above for basic statements. The procedure entry and exit semantics, of course, need to be altered to make use of the shadow variables.

As in the case of classical analyses, the operations performed are really abstractions of the run-time actions of activation record creation and destruction. Thus, the

abstract semantics of procedure entry "push" the current variables into the hidden variables, while procedure exits "pop" hidden variables back into current variables. Because of the nature of points-to analysis, however, these operations are somewhat more subtle.

It appears that the most complicated case is a directly recursive procedure call with parameter transmission. Consider the program in Figure 40. In the call at point 9, the *new* incarnation of w points to the *previous* incarnation of y. Thus, the abstract effect of this call must use the correct shadow and current variables.

Figure 40 Directly recursive function with parameter passing

```
   int   a,  b;
   int  *c, *d;
   int **e,**f;

   void main(void)
   {
1     f = &d;
2     d = &b;
3     p(f);
4  }
```

```
   void p(int **w)
   {
      int **x, *y, z;

5     *w = &a;
6      y = &z;
7      x = &y;
8     if (...)
9       p(x);
10 }
```

An appropriate abstract semantics for procedure entry to a procedure p involves the following three steps, where ptset is the set of points-to relations that hold on entry to p:

1. Target transfer: for all triples in ptset, transfer the target variables in the scope of p from current to shadows giving $ptset_1$.
2. Parameter transmission: for each (actual, formal) pair, perform the abstract effect of the assignment "`formal = actual`" using $ptset_1$ as the current set of triples, yielding $ptset_2$.
3. Source transfer: for all triples in $ptset_1$, transfer the source variables in the scope of p from current to shadow status giving $ptset_3$. In addition, definite points-to triples must be lowered to possible certainty when the transferred source variable "points to" more than one object. This ensures that a member of the lattice results.

The final set of triples is then ($ptset_2 \cup ptset_3$). Steps 2 and 3 mimic the run-time actions involved in a typical implementation of a procedure call: actual parameters are assigned to a fresh set of corresponding formals and then a fresh set of locals is created. The subtlety arises when a value pointing to a local is passed as an argument, as in the example program. In this case the formal should point to the shadow variable. Thus, the targets of the incoming triples are modified prior to performing parameter passing.

Returning from a procedure p involves undoing the effect of the call. This involves two steps, again assuming the set of triples holding immediately prior to exit is ptset:

1. Current local elimination: triples whose source is a variable local to the called procedure are eliminated from ptset yielding $ptset_1$. This imitates the deallocation of the activation record.
2. Shadow local re-instatement: for all triples in $ptset_1$ convert variables corresponding to shadows of locals in p to current variables.

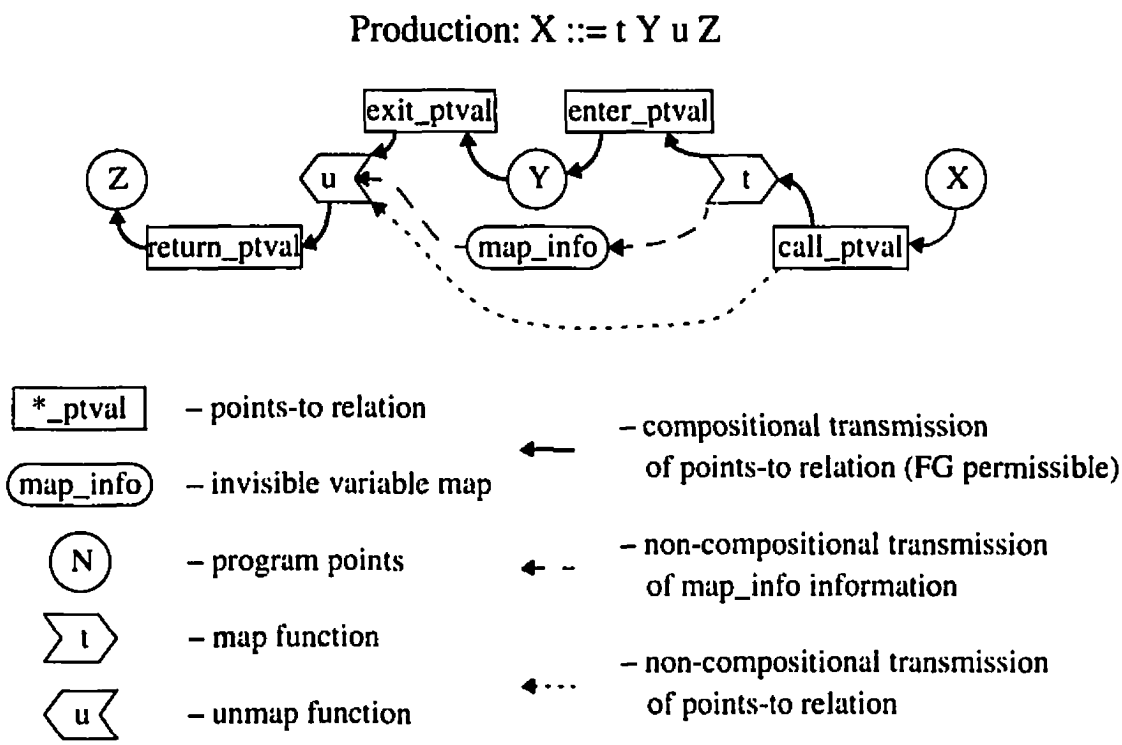
The result of the second step is the abstract state that holds in the calling procedure. Note that any triple in $ptset_1$ whose target is a current local of p is a dangling pointer: it points to storage that has just been deallocated and a warning message may be printed.

VI.3.6 Discussion

The interprocedural analysis technique given in the preceding sections departs considerably from Emami's thesis. Her analysis is based upon the McTAG analyzer generator scheme, inheriting the invocation graph structure and associated map/unmap functions for computing the effect of calls and returns. She takes advantage of scoping information in the map and unmap functions to limit the size of points-to relations and avoid redundant computation. The basic idea is to create "invisible variables" to represent variables that are accessible (i.e., by indirection) but not in the scope of the called procedure. By keeping track of what the invisible variables represent on a given call it is possible to transfer the effect of the procedure call on the caller's relationships.

Figure 41 shows the information flow when a procedure call is processed in Emami's analysis. First, invisible variables are created along with a mapping that records the correspondence between the real variables and the invisible counterparts (the dashed arc from t to map_info). At the same time the points-to relationships are transformed to involve these invisible variables (solid arc from t to enter_ptval). The body of the procedure is then analyzed using enter_ptval as the abstract state (arc from enter_ptval to Y). Analysis of the body the procedure yields the points-to information applicable on exit from the procedure (arc from Y to exit_ptval). The unmap function then taks the mapping information (map_info), the procedure exit state (exit_ptval) and the original points-to triples (call_ptval) to compute the points-to information that holds when the procedure returns (return_ptval).

Figure 41 Information flow in Emami's points-to analysis



To implement this formulation directly in FACT would require some mechanism for transmitting the mapping information as well as the originating points-to information (broken arcs in Figure 41). There are at least two possible approaches:

1. The FG analysis framework may be generalized. Instead of a compositional strategy for mapping the FPs, a separate function could be associated with each production. This function would take the meanings of the various components of the production as arugments and construct an appropriate abstract meaning. E.g.:

   P: X ::= t Y u Z $\qquad\qquad \mathcal{K}_Z \leq \mathcal{F}_P(f[\![u]\!], \mathcal{K}_Y, f[\![t]\!]\mathcal{K}_X)$

   This permits virtually arbitrary information flow and can clearly handle the flow represented by the broken arcs in Figure 41.

2. The lattice may be augmented to include a "stack" of mapinfo/ptset pairs and associated call strings which keep the required historical information:

Stack = (T* × MapInfo × PtSet)
where T is the set of FTs (T* is the call string component)

In order to keep the size of the stacks bounded, the semantic functions for FTs representing procedure call and return must examine the incoming tuple to determine if a recursive call is to be processed. In that event the existing pair is used rather than creating appending a new one onto the current stack.

The first approach has the disadvantage that a separate combination function must be supplied for each FP, which seems difficult to generalize into a usable tool (cf. Section V.4.5). The latter demonstrates the generality of the FG framework.

In summary, FACT has been used to implement points-to analysis, an important data flow problem. The original formulation of points-to analysis is based on Sridharan's data flow analysis framework devised for use in the McCAT compiler development environment [81] and uses non-compositional interprocedural data flow. Implementing points-to analysis in FACT required re-working the intraprocedural flow information space (that is, sets of points-to triples) into a lattice and devising a compositional method for interprocedural data flow. As originally specified, the information space was very close to a lattice: an obvious ordering was already present with only the addition of the top element required (also suggested by Emami, but for different reasons). Using the techniques of Section VI.2.8 a compositional interprocedural interpretation was developed that was suitable for use in the FG framework. Thus, with a little effort on the part of the FACT user, it is possible to perform sophisticated interprocedural analyses whose original specification is not compositional.

VI.4 Adaptation of the Morel/Renvoise partial redundancy algorithm

An interesting technique pioneered by Morel and Renvoise, known as *partial redundancy elimination*, subsumes both the code motion and common subexpression elimination opti-

mizations [62]. This problem is a valuable test of the FG methodology and FACT tool because it involves several separate analyses contributing to the final solution. In its original formulation it is an intraprocedural bidirectional problem. A unidirectional variant of the problem has been devised by Schröer [76]. This second variant has been implemented using the FACT prototype, and maps exceptionally well to the FG framework.

Partial redundancy elimination attempts to reduce the number of times a given expression in a program is computed at run-time. The idea is to move computations to points where they are expected to be executed less frequently, and assigning the result of the computation to a temporary location, typically a register. Subsequent instances of the computation which are then redundant (i.e., guaranteed to have the same value) are replaced by a reference to the temporary. Morel and Renvoise have cleverly culled the required information about placement and deletion by combining the results of three classical flow analyses in a fourth bidirectional analysis.

The unidirectional variant arises from a number of observations about the original problem. In the original framework a standard basic block control flow graph is used; in the modified framework, extra empty blocks are inserted on all edges in the original control flow graph. Flow information is associated with the entry and exit of each block; that is, there is a control flow point before and after each basic block. (These two changes result in a finer granularity than that of the original formulation.) Combined, these two modifications allow the bidirectional flow analysis performed in the original formulation as a unidirectional problem. Furthermore, within a FG, empty blocks can be represented by chain FPs. A basic block with points on either side can be represented by a FP in the obvious manner. In fact, because the problem is now unidirectional, granularity is no longer a factor and basic block transformation need not be performed if the user does not wish it.

VI.4.1 Problem: possible placement

A computation is *partially redundant* at some point in a program if its value has been computed on at least one path to the point without an intervening assignment to any of its operands. These points are computed by computing the solution to the *possible placement* problem. Roughly, the computation of an expression E may be placed at some point P in a program if it is *anticipated* (computed on all paths from P before any of its operands are modified) and *partially available* (computed on some path to P without an intervening modification to any of its operands). Computations are then placed as far "forward" as the possible placement solution permits. The following sections detail these ideas.

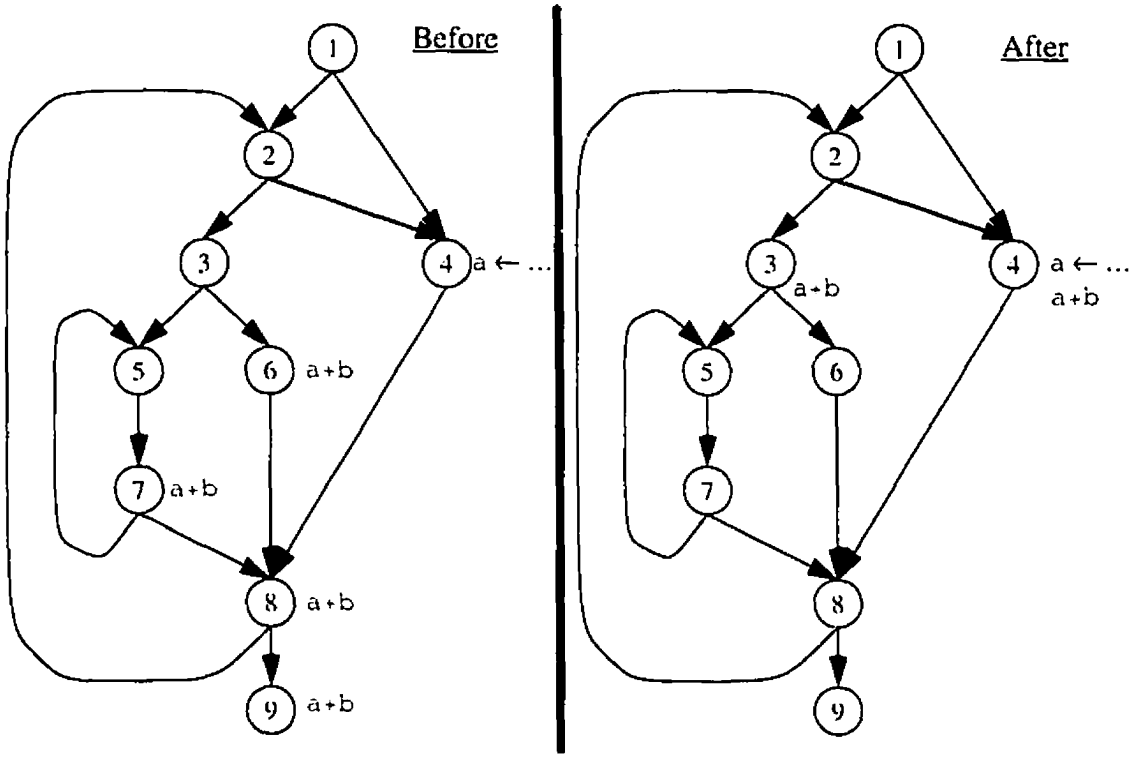
VI.4.2 Example

This technique is most easily understood in terms of a single expression within a control flow graph. Figure 42, reproduced from [62], shows two control flow graphs whose numbered nodes correspond to basic blocks. Occurrences of the expression `a+b` indicate nodes where the expression is computed (6,7,8 and 9 in the original graph on the left side). Node 4 contains an assignment to `a`, the only place where an operand of `a+b` is modified. By inspection, it is clear that the occurrence of `a+b` in node 9, for example, is rededundant: it is always computed by node 8, the only predecessor of node 9. In addition, it is possible to move the computations of `a+b` from nodes 6 and 7 to node 3. Note that node 7 is in the inner loop and its removal in this case results in one less addition per iteration of nodes 5/7. Further, moving the addition to node 3 causes no harm, because it is always computed on all emanating paths before the value of `a` changes. One last change is to note that `a+b` is available on two paths into node 8 (from 6 and 7), but not 4 (which modifies `a`). Thus, if a computation of `a+b` is inserted at the exit of 4 it can be eliminated from 8, too. The final result of these modifcations is shown in the control graph on the right hand side of the figure.

VI.4.3 Preliminaries

As an antecedent to both the original and modified algorithms, various properties of the program must be determined. These include a number of *local* and *global properties* associated with each expression in the program. Gathering the set of all expressions in the program simplifies the implementation, and is done using standard attribute grammar techniques.

Figure 42 Partial redundancy example control flow graph



Local properties

In both the original and unidirectional formulation the following local properties are associated with basic blocks. In the FACT implementation they are associated with the terminals in the FG.

1. *Transparency: TRANSP(t).* The set of all expressions E that are "transparent" for a terminal t; that is, none of E's operands is modified by the execution of t.
2. *Local availability: COMP(t).* The set of all expressions E that are "locally available" for a terminal t; that is, if t represents at least one computation of E that occurs *after* any modification of the operands of E in t (if any).
3. *Local anticipability:ANTLOC(t).* The set of all expressions E that are "locally anticipated" for a terminal t; that is, if t represents at least one computation of E that occurs *before* any modification of the operands of E in t (if any).

As with the universal set of expressions, these properties are computed using attribution rules.

Global properties

In the unidirectional formulation, solutions to the following data flow analyses are needed for computation of "districts" (see below).

1. *Availability: AVAIL(N).* An expression E is "available" at a nonterminal N if there exists a computation of E on every path from the beginning of the program to N without intervening modification to any of the operands of E. AVAIL(N) is the set of all such expressions for each point in the program. This is also known as the classical flow analysis "available expressions" – a forward intersection problem. The flow grammar formulation of this problem is given in Figure 43.
2. *Partial availability: PAVAIL(N).* An expression E is "partially available" at a nonterminal N if there exists a computation of E on at least path from the beginning of the program to N without intervening modification to any of the operands of E. PAVAIL(N) is the set of all such expressions for each point in the program. This is simply a union variant of the availability problem – see Figure 44.

Within FACT these two analyses could be combined and the solutions computed simultaneously, as shown in Figure 45.

Figure 43 Availability system

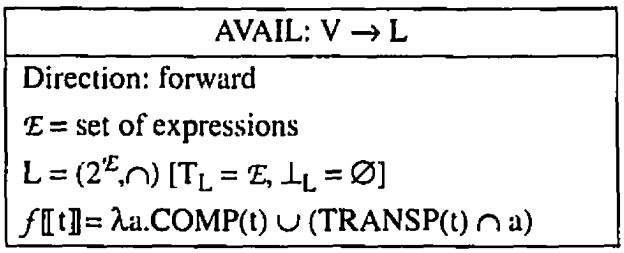


Figure 44 Partial availability system

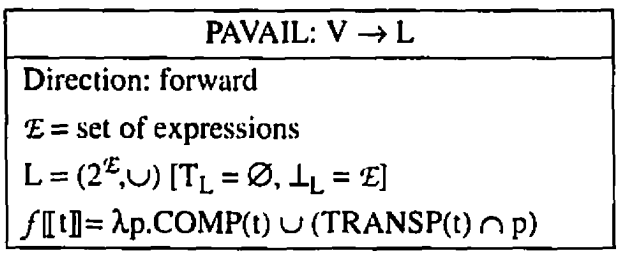
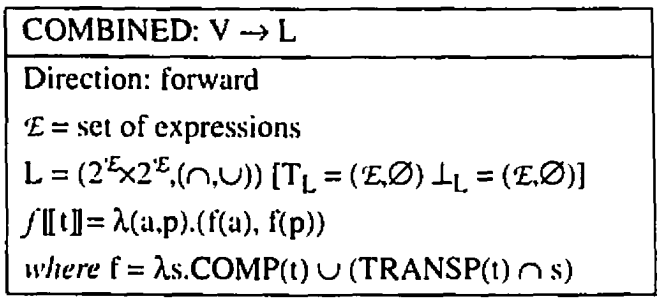


Figure 45 Combined availability and partial availability system



VI.4.4 Standard flow equations

The standard flow equations for possible placement assume:

1. The program is in the form of basic blocks, each having an IN and OUT point, as do the solutions to the preceding global properties. That is, the

AVAIL and PAVAIL solutions are separated into $AVIN_b/AVOUT_b$ and $PAVIN_b/PAVOUT_b$, respectively, for each basic block b.

2. An additional global system, called *anticipability*, has been solved giving $ANTIN_b/ANTOUT_b$. It is analogous to ANTLOC above, but propagated backwards throughout the program. The confluence operator is intersection: for an expression to be anticipable at the end of a block it must be anticipated on entry to all succesors of the block.

Given these values, the possible placement equations are:

$$PPIN_i = \begin{cases} \emptyset, \text{ if i is the entry block, otherwise} \\ \left(CONST_i \cap \bigcap_{j \in pred(i)} \left(PPOUT_j \cup AVOUT_j\right) \cap \left(ANTLOC_i \cup TRANSP_i \cap PPOUT_i\right)\right) \end{cases}$$

$$PPOUT_i = \begin{cases} \emptyset, \text{ if i is the exit block, otherwise} \\ \bigcap_{k \in succ(i)} PPIN_k \end{cases}$$

where $CONST_i = ANTIN_i \cap (PAVIN_i \cup \neg ANTLOC_i \cap TRANSP_i)$

VI.4.5 Discussion

The essential idea here is that a computation of an expression may be placed at a given point if it is guaranteed to be computed on all paths emanating from the point and the expression is available on at least one path to the point. These constraints are embodied in the flow equations. $CONST_i$ is a "fudge factor" that represents expressions that are partially redundant or indirectly anticipated in block i. Later formulations of the framework drop this term [42]. The reader is directed to the original paper for further information [62].

The following sections present a different approach yielding similar results requiring neither the bidirectional equations nor the anticipability system. This approach has been successfully implemented using the FG framework using the FACT prototype.

VI.4.6 Intraprocedural analysis using districts

The final analysis computes a set of *districts* in which it may be profitable to move a computation. A district is, roughly, a maximal section of program in which an expression E is both transparent and *anticipated* (i.e., there is a computation of E at every exit of the district). Thus, there are different districts associated with each expression in the program. All the districts can be computed simultaneously with a flow analysis using a powerset lattice representation. A final post-processing pass over the grammar uses the results of this analysis to determine where computations can be profitably inserted, while a traversal of the AST tags the redundant computations to be deleted.

Figure 46 shows the district analysis system, which is a backward analysis problem. The details of why this analysis works may be found in the original paper. Intuitively, the FT map incorporates the informal definition of district given in the previous paragraph. Consider an expression E. The various components of the equation for $f[\![t_{i/j}]\!]$ may be understood as follows:

1. ANTLOC($t_{i/j}$): to start a district, a computation of E is needed.
2. $d \cap$ TRANSP($t_{i/j}$): a district for E may be extended if its operands are not modified by an action.
3. PAVAIL(S_i): if E is available on at least one path entering the district, then moving E to other entries of the district may be profitable, and can do no harm . The "entry" to a terminal $t_{i/j}$ is S_i. Note that deleting this term from the equation enables more possibilities for code movement, at the expense of occasional unprofitable movement [42].

Figure 46 District system

DISTRICT: $V \rightarrow L$
Direction: backward
$\mathcal{E}$ = set of expressions
$V = \{ S_0, S_1, \ldots, S_n \}$ – FNs of the FG
$L = (2^{\mathcal{E}}, \cap)$ $[T_L = \mathcal{E}, \perp_L = \varnothing]$
$f[\![t_{i/j}]\!] = \lambda d.PAVAIL(S_i) \cap (ANTLOC(t_{i/j}) \cup (d \cap TRANSP(t_{i/j})))$

Insertion and deletion

Given the solution to the districts problem, the places where computations of expressions are to be inserted and deleted may be computed in a straightforward manner. Computing the insertions involves a single iteration over the FPs, making use of the information gathered from the previous analyses. Deletions may be computed by a traversal of the AST of the program.

Consider a district D for an expression E. The district flow system ensures the presence of a computation of E at every exit of D. Because D is transparent for E, evaluating E anywhere in D is guaranteed to compute the same value. By inserting computations of E at entries to D where E is not already available, all the computations of E at the exits become redundant and may be deleted. This reasoning is captured in the following formula:

$$INSERT(N) = DISTRICT(N) \cap \bigcap_{L::=\alpha N \in P_N} \neg (DISTRICT(L) \cup AVAIL(L))$$

An algorithm to compute INSERT is shown in Figure 47.

Figure 47 Algorithm for computing INSERT, given DISTRICT and AVAIL

Input: G=‹V,T,P,S›
AVAIL : V → L
DISTRICT : V → L

Output: INSERT : V → L

Method:

INSERT ← DISTRICT;
foreach "X ::= α Y" ∈ P **do**
  INSERT[Y] ← INSERT[Y] ∩ ¬(DISTRICT[X] ∪ AVAIL[X])

After the insertions have been made, any locally anticipable expression in a district may be deleted, which may be expressed as follows:

$$DELETE(t_{i/j}) = ANTLOC(t_{i/j}) \cap DISTRICT(S_i)$$

Note that actions, and thus expressions and their deletions, are associated with FTs.

VI.4.7 Summary

Partial redundancy elimination is a useful flow analysis problem that subsumes several classical code optimizations, including code motion and common subexpression elemination. Its solution in the FACT tool is interesting because:

1. A design goal of the FACT tool was to permit multiple interacting analyses. Partial redundancy elimination requires solution of a number of flow analysis problems as a precursor to the final placement analysis: successful imple-

mentation of the analysis in the FACT prototype clearly demonstrates that this goal has been met.

2. A post-processing phase is needed to determine the actual deletion and insertion of computations. The FG intermediate representation makes this straightforward.
3. Lastly, the utility of FGs for relatively low level analyses has been demonstrated.

VI.5 Procedure variables

Procedure and label variables present an important challenge to any flow analysis generation system such as FACT. The control flow model must represent all feasible traces to ensure that flow analyses provide conservative information. Analyzing programs with procedure and label variables results in a "chicken-and-egg" situation: to do a flow analysis, one needs a complete representation of a program; to determine the sets of possible values for procedure and label variables, one needs to do a flow analysis! This chapter explores this dilemma and presents an approach for dealing with the problem that can be used in FACT.

VI.5.1 Introduction

To this point it has been assumed that a FG G representing some program was *complete* in the sense that the language generated by G contains all feasible traces. Procedure variables (PVs) and label variables, however, represent control flow choices that are not determined until run-time. On the one hand, the only way to ensure at the outset that all feasible traces are generated by the FG is to add productions for all possible invocations at each call through a PV. This is, however, extremely pessimistic and may yield unsatisfactory information [32]. On the other hand, it is incorrect, and possibly disastrous, in general to use an

incomplete FG to perform an analysis. Note that because label variables represent a restricted form of PVs, this chapter discusses only the latter.

The approach taken in this section is to start with a complete FG and effectively perform a range analysis as in [92] using a technique similar to conditional constant propagation [91]. The fixpoint solution to this problem may be used to elide productions from the FG before performing subsequent analyses. Another possible approach is to start with an incomplete FG and allow the analysis to add FPs "dynamically" as PV call sites are processed. Termination becomes the main issue in this case: care is needed to ensure that the analysis does not attempt to build an infinite grammar.

An important problem at the periphery of the PV issue is *aliasing*. Many programming languages allow multiple names, or aliases, for a given unit of data. The most obvious example of aliasing is the pointer: the address of an object in memory. Other examples include reference parameters and arrays. Such constructs complicate the situation further: not only must the values of PVs be tracked, so must sets of possible aliases. One particularly difficult language is C: not only does it encourage aliasing through the use of pointers, it allows pointers to objects of one type to be cast to pointers of another type. The points-to analysis, described in Section VI.3.1 above, may be combined with the technique in this section to handle this complication [21].

VI.5.2 Example

The program fragment in Figure 48 illustrates the basic difficulty that arises when attempting to analyze code in the presence of procedure variables. When attempting to build the FP for the call between points 6 and 7, it is not known in general whether x will have the

value `inc` or `dec` at run-time. To be conservative, both possibilities must be considered, yielding the following FPs:

$$S_6 ::= t_{6/1}\, S_1\, t_{2/7}\, S_7 \quad | \quad t_{6/3}\, S_1\, t_{4/7}\, S_7$$

Unfortunately, this can yield information that is too imprecise to allow useful optimization, particularly in a language such as C where type information cannot be used to eliminate possible procedure values [32]. Ideally, one would like to perform a flow analysis to do this, which would determine that `x` has the value `inc` in this case (note that there could be arbitrarily many statements between the assignment and subsequent call to `x`.

Figure 48 Procedure variables

```
   var x:procedure;
       i:integer;
   procedure inc;
   begin
1      i:=i+1
2  end;
   procedure dec;
   begin
3      i:=i-1
4  end;
   ...
5     x := inc;
6     call x;
7  ...
```

VI.5.3 Discussion

Procedure variable tracking is one flow analysis for which there do not exist any flow equations that may be considered "standard." The root of the problem can be traced back to the relationship between CPS style semantics and data flow analysis (cf. Section IV.7): a data flow analysis abstracts away a good portion of the information present in the standard semantics. In particular, the data flow analysis represents a semantics in which each identifier use involving control flow has been resolved. To be safe, all continuations that the identifier might represent must be included. However, this can be "undone" by performing a data flow analysis using an abstract state designed to capture sets of possible continuations. Adding an element to the lattice to indicate the infeasibility of a given continuation permits the "no information" to be propagated.

VI.5.4 General approach: value tracking

The general approach to handling this problem is to compute a set of possible values for each procedure variable at each point in the program. Let V be the set of procedure variables and C be the set of procedure constants in a program. Then the intraprocedural lattice $L = (V \rightarrow 2^C, \wedge_L)$ where:

$$f \wedge_L g = \lambda v.f(v) \cup g(v)$$

The meaning of an assignment to a procedure variable in the program such as:

```
x := p;
```

is (a variant of):

$\lambda s.\lambda v.s[\{p\}/x]$, if $p \in C$ (a procedure constant)
$\lambda s.\lambda v.s[s(p)/x]$, if $p \in V$ (a procedure variable)

This leaves the meaning of a call through a procedure variable, such as:

```
call x;
```

which is described in the next section.

VI.5.5 Lattice extension

The basic requirement of eliminating information flow may be achieved by adding a special element to the lattice representing the absence of information. If L is the lattice given above, define $L_{\otimes}$ as follows:

$$L_{\otimes} = (L \cup \{\otimes\}, \wedge_{L_{\otimes}}), \textit{ where}$$

$$f \wedge_{L_{\otimes}} g = \begin{cases} f, \text{ if } g = \otimes \\ g, \text{ if } f = \otimes \\ f \wedge_{L} g, \text{ otherwise} \end{cases}$$

The idea is that $\otimes$ represents a control flow path that is currently not known to be feasible: when it is combined with any other control flow path, the result is simply whatever is specified by the other path. A further requirement is needed to ensure that such paths are, indeed, ignored: all semantic functions in the MDFAF function space are constrained to map $\otimes$ to $\otimes$. That is, if $F_{\otimes}$ is a MDFAF associated with $L_{\otimes}$, then $\forall f \in F_{\otimes}: f(\otimes) = \otimes$. Note that this is a general technique, and may be used in other analyses, such as conditional constant propagation [91], where data flow information is used to determine possible control flow.

It is worth noting that $\top_L$ cannot be used as the "ignore" element. Top represents the static information that nothing is currently known about any given procedure variable. Flow functions applied to $\top_L$ need not return $\top_L$ in general. Thus, $\top_L$ does not represent the absence of control flow; a straightforward solution is to add a special element, as above.

Now the meaning of a call to a procedure variable becomes a matter of examining the current data flow information to decide which paths require propagation and which do not. In the case of the program in Figure 48, the effect of the call to x may be specified as follows:

$$\mathcal{K}_{S7} = f[\![t_{2/7}]\!] \circ \mathcal{K}_{S1} \circ f[\![t_{6/1}]\!] \circ \mathcal{K}_{S6} \wedge_{PF(L_{\mathcal{Z}})} f[\![t_{4/7}]\!] \circ \mathcal{K}_{S3} \circ f[\![t_{6/3}]\!] \circ \mathcal{K}_{S6}$$

where

$f[\![t_{2/7}]\!] = f[\![t_{4/7}]\!] = \lambda s.s,$

$$f[\![t_{6/1}]\!] = \lambda s.\{ \begin{matrix} \otimes, (s = \otimes) \text{ or } ((s \neq \otimes) \text{ and } (\texttt{inc} \notin s(x))) \\ s[\{\texttt{inc}\}/x], ((s \neq \otimes) \text{ and } (\texttt{inc} \in s(x))) \end{matrix} \quad \text{and}$$

$$f[\![t_{6/3}]\!] = \lambda s.\{ \begin{matrix} \otimes, (s = \otimes) \text{ or } ((s \neq \otimes) \text{ and } (\texttt{dec} \notin s(x))) \\ s[\{\texttt{dec}\}/x], ((s \neq \otimes) \text{ and } (\texttt{dec} \in s(x))) \end{matrix}$$

Thus, the actual flow information is propagated to the FPs for the procedure `inc` only if `inc` is known to be a possible value of x at the point of call ($f[\![t_{6/1}]\!]$) and similarly for `dec` ($f[\![t_{6/3}]\!]$). Note, however, that $\otimes$ is still propagated through the FPs for both `inc` and `dec` – it simply does not contribute to the solution.

VI.6 Summary

This chapter is an original demonstration of how the flow grammar framework may be applied to a broad range of analyses found in the literature. Taken together, these adapta-

tions give strong evidence to the generality of the use of flow grammars and their suitability as a basis for a compiler construction. Specific techniques discussed include the use of "shadow variables" to increase the precision of analyses in the presence of recursion and the original use of a special top lattice element to represent the infeasibility of paths. The implementation of many important analyses, including the classical analyses of Hecht, two forms of aliasing analysis, a procedure variable analysis and a partial redundancy analysis show the applicability of the framework and the utility of the prototype FACT tool. Together, these examples are strong witness to the generality of the FG framework and its realization in the FACT tool: we believe that most flow analyses the user of a compiler generation environment would require may be expressed and solved using this system.

VII Conclusions and future work

Flow analysis is an important prerequisite for performing effective optimization in a compiler. Automating the construction of flow analyzers has been hampered by the lack of a uniform model for describing intra- and interprocedural control flow. The main contribution of this dissertation is the *flow grammar* which addresses this need. Flow grammars provide an excellent mechanism both for describing control flow and for specifying data flow analyses.

VII.1 Contributions

The main contributions of this dissertation include the following:

- A new framework for interprocedural analysis: the *flow grammar* control flow model. Using a flow grammar it is possible to represent an entire program, including procedure calls, using a single uniform structure. That is, representing a program with interprocedural control flow is not fundamentally different from representing one without procedure calls – a grammar suffices in both cases. This is of considerable advantage

when constructing a compiler generation tool. Just as one does not typically specify scanners with finite state machines, one should not be forced to use contorted attribute equations to solve flow analyses. Nor should one be required to use graphs to model interprocedural control flow where they are not suitable.

- A proof that the greatest fixed point solution to a set of equations corresponding to a flow grammar G with a given interpretation of the flow terminals conservatively estimate the effect of all execution paths in $\mathcal{L}(G)$. This validates the use of grammars for data flow analysis.
- General *interpretations* of flow grammars that permit a broad range of flow analyses to be specified. This is achieved by associating flow functions with the terminals in a flow grammar and mapping the productions to a set of equations over variables corresponding to the nonterminals. For intraprocedural problems a single element of the lattice is computed. In the interprocedural case, however, a partial function from lattice elements to lattice elements is computed for each nonterminal. Further, in the intraprocedural case the bidirectional interpretation contains the forward and backward variants. Perhaps somewhat surprisingly, however, in the interprocedural case the bidirectional interpretation is formed as the union of the forward and backward interpretations.
- A variety of *algorithms* for solving flow grammar interpretations are introduced. Practical issues involved in actually implementing several of these algorithms are discussed in Appendix A. One surprising result is that it is not necessarily the case that performing the "basic block" transformation reduces the number of lattice operations needed to solve a flow analysis problem.
- Evidence that the flow grammar model serves as a basis for prototyping flow analyses, by way of an *implementation* of the FACT tool. Several important, representative, analyses have been implemented:

  1. Interprocedural variants of classical bit vector analyses [29].
  2. Two forms of alias analysis: a simple intraprocedural analysis [3] and Emami's interprocedural points-to analysis [20].

3. A variant of the Morel/Renvoise partial redundancy elimination algorithm [62] based on the notion of districts [76].
4. Procedure variable analysis using a general technique to limit information propagation along infeasible paths. This example demonstrates value tracking and introduces the notion of an "ignore" lattice value for avoiding the transmission of information along invalid paths.

VII.2 Future work

This research opens a number of new directions for future research, including:

- An implementation of the FACT tool that uses a GUI interface to specify the control and data flow aspects of static analyses. The resulting specification would then be translated into attribute grammar equations to generate the flow grammar, solve the resulting equations and distribute the solution throughout the abstract syntax tree.
- A companion tool to FACT that transforms FG intermediate representation using the results of flow analyses. Continued advances in hardware design will undoubtedly require ever more sophisticated analyses and optimizations.
- New fixpoint algorithms that make more use of the flow grammar structure. One approach is the use of *grammar flow analyses* to find efficient evaluation orders.
- Incremental algorithms to support the repetitive nature of the optimization process.
- A study of the practical aspects of implementing fixpoint algorithms. Several aspects of implementing fixpoint algorithms that are not obvious from the theory come into sharp focus when actually implemented and used to analyze programs.
- A more complete study of the relationship between continuation passing style semantics and flow grammars.

References

[1] AT&T Bell Laboratories. *The Standard ML of New Jersey Library Reference Manual*. Feb. 1993.

[2] Abramsky, S. and C. Hankin. "An Introduction to Abstract Interpretation," in *Abstract Interpretation of Declarative Languages*, Abramsky S. and C. Hankin (eds.), 1987, pp.9-31.

[3] Aho, A. V., R. Sethi and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Publishing, 1986.

[4] Alblas, H. "Attribute Evaluation Methods," *Attribute Grammars, Applications ans Systems, Intl. Summer School SAGA, Lecture Notes in Computer Science 545*, Springer-Verlag, June 1991, pp. 48-114.

[5] Alblas, H. "Introduction to Attribute Grammars," *Attribute Grammars, Applications ans Systems, International Summer School SAGA*, June 1991, pp. 1-15.

[6] Alpern, B., M. Wegman and F. Zadeck. "Detecting Equality of Values in Programs," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, January 1988, pp. 1-11.

[7] Appel, A. *Compiling with Continuations*, Cambridge University Press, 1992.

[8] Arbib, M. A., A. J. Kfoury and R. N. Moll. *A Basis for Theoretical Computer Science*, Springer-Verlag, 1981.

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

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

[11] Banning, J. P. "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables," *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, January 1979, pp. 29-41.

[12] Barth, J. M. "A Practical Interprocedural Data Flow Analysis Algorithm," *Communications of the Association for Computing Machinery* 21, 9, September 1978, pp. 724-736.

[13] Cleaveland, J. C. and R. C. Uzgalis. *Grammars for Programming Languages*. Elsevier, 1977.

[14] Clocksin, W. F. and C. S. Mellish. *Programming in Prolog*, Springer-Verlag, 3rd edition, 1987.

[15] Cooper, B. G. *Ambitious Data Flow Analysis for Procedural Programs*. MS Thesis. University of Minnesota, 1989.

[16] Cooper, K., K. Kennedy and L. Torczon. "The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment," *ACM Transactions on Programming Languages and Systems* 8, 4, October 1986, pp. 491-523.

[17] Cooper, K. and K. Kennedy. "Interprocedural Side-Effect Analysis in Linear Time," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23, 7, June 1988, pp. 57-66.

[18] Cousot, P. and R. Cousot. "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, January 1977, pp. 238-252.

[19] Deransart, P. et al. *Attribute Grammars: Definitions, Systems and Bibliography, Lecture Notes in Computer Science 323*, Springer-Verlag, 1988.

[20] Emami, M. *A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing C Compiler*. MSc Thesis. McGill University, 1993.

[21] Emami, M., R. Ghiya and L. J. Hendren. "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 29, 6, June 1994, pp. 242-256.

[22] Erosa, A. M. and L. J. Hendren. "Taming Control Flow: a Structured Approach to Eliminating Goto Statements," ACAPS Technical Memo 76, McGill University, September 29, 1993.

[23] Farnum, C. D. *Pattern-Based Languages for Prototyping of Compiler Optimizers.* PhD Dissertation. University of California at Berkeley, 1990.

[24] Farrow, R. "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars," *SIGPLAN'86 Symposium on Compiler Construction*, SIGPLAN Notices 21, 7, July 1986, pp. 85-98.

[25] Glanville, R. S. and S. L. Graham. "A New Method for Compiler Code Generation," *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, January 1978, pp. 231-240.

[26] Gomard, C. K. and N. D. Jones. "Compiler Generation by Partial Evaluation: A Case Study," *Structured Programming* 12, 3, 1991, pp. 123-144.

[27] Gray, R. W. et al. "Eli: a Complete Compiler Construction System," *Communications of the Association for Computing Machinery* 35, 2, February 1992, pp. 121-131.

[28] Grundman, D. R. *Graph transformations and program flow analysis.* PhD Dissertation. University of California at Berkeley, 1990.

[29] Hecht, M. S. *Flow analysis of computer programs.* Elsevier, 1977.

[30] Heindel, L. E. and J. T. Roberto. *LANG-PAK – An Interactive Language Design System.* Elsevier, 1975.

[31] Hendren, L. J. et al. "Designing the McCAT compiler based on a family of structured intermediate representations," *Fifth Workshop on Languages and Compilers for Parallel Computing*, Yale University, 1992, pp. 261-275.

[32] Hendren, L. J., et al. "A Practical Context-Sensitive Interprocedural Analysis Framework for C Compilers," ACAPS Technical Memo 72, July 24, 1993.

[33] Heuring, V. P. "The Automatic Generation of Fast Lexical Analyzers," *Software – Practice & Experience 16*, September 1986, pp. 801-808.

[34] Hoare, C. A. R. "Communicating Sequential Processes," *On the Construction of Programs*, McKeag R. M. and A. M. Macnaghten (eds.), Cambridge University Press, 1980, pp. 229-254.

[35] Horspool, R. N. S. and M. R. Levy. "Mkscan – An Interactive Scanner Generator," *Software – Practice and Experience* 17, 6, June 1987, pp. 369-378.

[36] Horspool, R. N. S. "Incremental Generation of LR Parsers," *Journal of Computer Languages* 15, 4, 1990, pp. 205-223.

[37] Hudak, P. et al. *Report on the Programming Language Haskell – A Non-strict Purely Functional Language.* Version 1.0, April 1990.

[38] Hughes, J. "Analysing Strictness by Abstract Interpretation of Continuations," *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987, pp. 63-102.

[39] Jeuring, J. and D. Swierstra. "Bottom-up Grammar Analysis – a Functional Approach," Technical Report UU-CS-1994-01, Utrecht University, January, 1994.

[40] Johnson, S. C. "Yacc–Yet Another Compiler Compiler," Computer Science Technical Report 32, AT&T Bell Laboratories, 1975.

[41] Jones, N. D. "Flow Analysis of Lazy Higher-Order Functional Programs," *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987, pp. 103-122.

[42] Joshi, S. M. and D. M. Dhamdhere. "A Composite Hoisting-Strength Reduction Transformation for Global Program Optimization – Part I," *International Journal of Computer Mathematics.* 11, 1982, pp. 21-41.

[43] Jourdan, M. and D. Parigot. "Techniques for Improving Grammar Flow Analysis," *European Symposium on Programming '90, Lecture Notes in Computer Science 432*, Springer-Verlag, pp. 240-255.

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

[45] Kastens, U. "Attribute Grammars in a Compiler Construction Environment," *Attribute Grammars, Applications and Systems, International Summer School SAGA*, June 1991, pp. 380-400.

[46] Kastens, U. *LIDO – A Language for Attribute Grammar Specification.* University of Paderborn, 1993.

[47] Kastens, U. "Ordered Attribute Grammars," *Acta Informatica* 13, 1980, pp. 229-256.

[48] Kastens, U., B. Hutt, and E. Zimmerman. *GAG – A Practical Compiler Generator, Lecture Notes in Computer Science 141*, Springer-Verlag 1982.

[49] Kennedy, K. "A Global Flow Analysis Algorithm," *International Journal of Computer Mathematics*, Section A, Volume 3, 1971, pp. 5-15.

[50] Kennedy, K. "A Survey of Data Flow Analysis Techniques," in *Program Flow Analysis: Theory and Applications*, Muchnick S. and Jones N. (eds.), 1981, pp. 5-54.

[51] Kildall, G. "A Unified Approach to Global Program Optimization," *ACM Symposium on Principles of Programming Languages*, October 1973, pp. 194-206.

[52] Knoop, J. and B. Steffen. "The Interprocedural Coincidence Theorem," *4th International Conference, Compiler Construction '92*, October 1992, pp. 125-140.

[53] Knoop, J., B. Steffen and O. Rüthing. "A Tool Kit for Constructing Optimal Interprocedural Data Flow Analyses," MIP-9413, Universität Passau, November 1994.

[54] Knuth, D. E. "Semantics of context-free languages," *Mathematical Systems Theory*, 2(2), 1968, pp. 127-145; correction: *Mathematical Systems Theory*, 5(1), 1971, pp. 95-96.

[55] Landi, W. and B. G. Ryder. "Pointer-Induced Aliasing: A Problem Classification," *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, January 1991, pp. 93-103.

[56] Landi, W. and B. G. Ryder. "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing," *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 27, 7, June 1992, pp. 235-248.

[57] Lee, P. *Realistic Compiler Generation*. PhD Dissertation. MIT Press, 1989.

[58] Lesk, M. E. "Lex - a Lexical Analyzer Generator," Computing Science Technical Report 39, AT&T Bell Laboratories, 1975.

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

[60] Michaelson, G. "Interpreter Prototypes from Language Definitions Style Specifications," *Information and Software Technology*, 30(1), 1988, pp.23-31.

[61] Möncke, U. and R. Wilhelm. "Grammar Flow Analysis," *Attribute Grammars, Applications ans Systems, International Summer School SAGA, Lecture Notes in Computer Science 545*, Springer-Verlag, June 1991, pp. 151-186.

[62] Morel, E. and C. Renvoise. "Global Optimization by Suppression of Partial Redundancies," *Communications of the Association of Computing Machinery* 22, 2, February 1979, pp. 96-103.

[63] Morel, E. and C. Renvoise. "Interprocedural Elimination of Partial Redundancies," in *Program Flow Analysis: Theory and Applications*, Muchnick S. and Jones N. (eds.), 1981, pp.160-188.

[64] Myers, E. W. "A Precise Inter-Procedural Data Flow Algorithm," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, January 1981, pp. 219-230.

[65] Nielson, F. "A Denotational Framework for Data Flow Analysis," *Acta Informatica* 18, 1982, pp. 265-287.

[66] Nielson, F. "Program Transformations in a Denotational Setting," *ACM Transaction on Programming Languages and Systems* 7, 3, July 1985, pp. 359-379.

[67] Parr, T., H. G. Dietz and W. E. Cohen. "PCCTS Reference Manual – Version 1.00," *ACM SIGPLAN Notices* 27, 2, February 1992, pp. 88-165.

[68] Parr, T. and R. W. Quong. "Adding Semantic and Syntactic Predicates to LL(k): pred-LL(k)," *5th International Conference on Compiler Cconstruction'94*, April 1994, pp. 263-277.

[69] Pelegri-Llopart, E. and S. L. Graham. "Optimal Code Generation for Expression Trees: an Application of BURS Theory," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, 1988, pp. 294-308.

[70] Proebsting, T. A. "Simple and Efficient BURS Table Generation," *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 27, 7, July 1992, pp. 331-340.

[71] Rees, J. and W. Clinger. "Revised^4 Report on the Algorithmic Language Scheme," MIT AI memo 848b, November 1992.

[72] Rosen, B. "High-Level Data Flow Analysis," *Communications of the Association for Computer Machinery* 20, 10, October 1977, pp. 712-724.

[73] Rosendahl, Mads. *Abstract interpretation and attribute grammars*. PhD Dissertation. University of Cambridge, 1991.

[74] Ryder, B. "Constructing the Call Graph of a Program," *IEEE Transactions on Software Engineering* SE-5, 3, May 1979, pp. 216-226.

[75] Sagiv S., et al. "Resolving Circularity in Attribute Grammars with Applications to Data Flow Analysis," *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, January 1989, pp. 36-48.

[76] Schröer, F. W. "Districts: A Foundation for the Suppression of Partial Redundancies," GMD-Bericht Nr. 304 (Arbeitspapiere der GMD), May 1988.

[77] Sethi, R. "Control Flow Aspects of Semantics-Directed Compiling," *ACM Transactions on Programming Languages and Systems* 5, 4, October 1983, pp. 554-595.

[78] 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.

[79] Shivers, O. "Control Flow Analysis in Scheme," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23, 7, June 1988, pp. 164-174.

[80] Silverberg, B. A. *Using a Grammatical Formalism as a Programming Language*. MSc Thesis, Technical Report CSRG–88, University of Toronto, 1978.

[81] Sridharan, B. *An Analysis Framework for the McCAT Compiler*. MSc Thesis. McGill University, 1992.

[82] Stoy, J. E. *The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[83] Tarjan, R. E. "A Unified Approach to Path Problems," *Journal of the Association for Computing Machinery* 28, 3, July 1981, pp. 577-593.

[84] Tarjan, R. E. "Fast Algorithms for Solving Path Problems," *Journal of the Association for Computing Machinery* 28, 3, July 1981, pp. 594-614.

[85] Tennent, R. D. *Semantics of Programming Languages*. Prentice Hall, 1991.

[86] Tjiang, W. K. T. and J. L. Hennessy. "Sharlit – A Tool for Building Optimizers," *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 27, 7, June 1992, pp. 82-93.

[87] Van Hentenryck, P., O. Degimbe, B. Le Charlier and L. Michel. "The Impact of Granularity in Abstract Interpretation of Prolog," *Workshop on Static Analysis'93*, September 1993, pp. 1-14.

[88] Venkatesh, G. A. "A Framework for Construction and Evaluation of High-Level Specifications for Program Analysis Techniques," *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, SIGPLAN Notices, SIGPLAN Notices* 24, 7, July 1989, pp. 1-12.

[89] Venkatesh, G. A. *A Framework for Specification and Implementation of Program Analysis Algorithms*. PhD Dissertation. Computer Science Technical Report #875, University of Wisconsin – Madison, 1989.

[90] Waite, W. M. and L. R. Carter. *An Introduction to Compiler Construction*. Harper-Collins, 1993.

[91] Wegman, M. and F. Zadeck. "Constant Propagation with Conditional Branches," *ACM Transactions on Programming Languages and Systems* 13, 2, April, 1991, pp. 181-210.

[92] Weihl, W. "Interprocedural Data Flow Analysis in the Presence of Pointer, Procedure Variables, and Label Variables," *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, January 1980, pp. 83-94.

[93] Whitfield, D. and M. L. Soffa. "Automatic Generation of Global Optimizers," *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 26, 6, June 1991, pp. 120-129.

[94] Wilhelm, R. "OPTRAN – a Language/System for the Specification of Program Transformations: System Overview and Experiences," *Attribute Grammars, Applications and Systems, International Summer School SAGA*, June 1991, pp. 133-147.

[95] Wilhelm, R. "Computation and Use of Data Flow Information in Optimizing Compilers," *Acta Informatica* 12, 1979, pp. 209-225.

[96] Wilhelm, R. "Global Flow Analysis and Optimization in the MUG2 Compiler Generating System," in *Program Flow Analysis: Theory and Applications*, Muchnick S. and Jones N. (eds.), 1981, pp. 132-159.

[97] Yi, K. *Automatic Generation and Management of Program Analyses*. PhD Dissertation. University of Illinois at Urbana-Champaign, 1993.

[98] Yi, K. and W. L. Harrison III. "Automatic Generation and Management of Interprocedural Program Analyses," *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 246-259.

A Experience with the FACT prototype

A prototype of the FACT tool has been implemented in ML and used to construct the analyses described in Chapter VI. This appendix describes this effort and discusses a number of issues that have arisen as a result. The prototype has proven invaluable for testing the utility of the FG model and (several of) the algorithms of Chapter V.

A.1 Overview

In order to test many of the ideas presented in the body of this dissertation, a prototype of the FACT tool has been implemented. The main result has been the validation of the FG methodology for specifying and implementing flow analyses. In addition, the structure of the system permits different fixpoint algorithms to be evaluated by simply replacing a single module of the system. Several variants of the fixpoint algorithms described in Section V.5 have been written and compared when solving various analyses.

The following sections detail various aspects of the prototype effort:

- a brief description of the structure of the prototype, which makes heavy use of the ML module facility;
- an illustration of the intra- and interprocedrual live variable analysis implementations;
- a discussion of several fixpoint algorithms that have been implemented, along with the results of analyzing a real program using each of the algorithms; and
- an example illustrating the procedure variable implementation is described, again with the results of analyzing the program using the various algorithms.

A.2 Structure of the FACT prototype

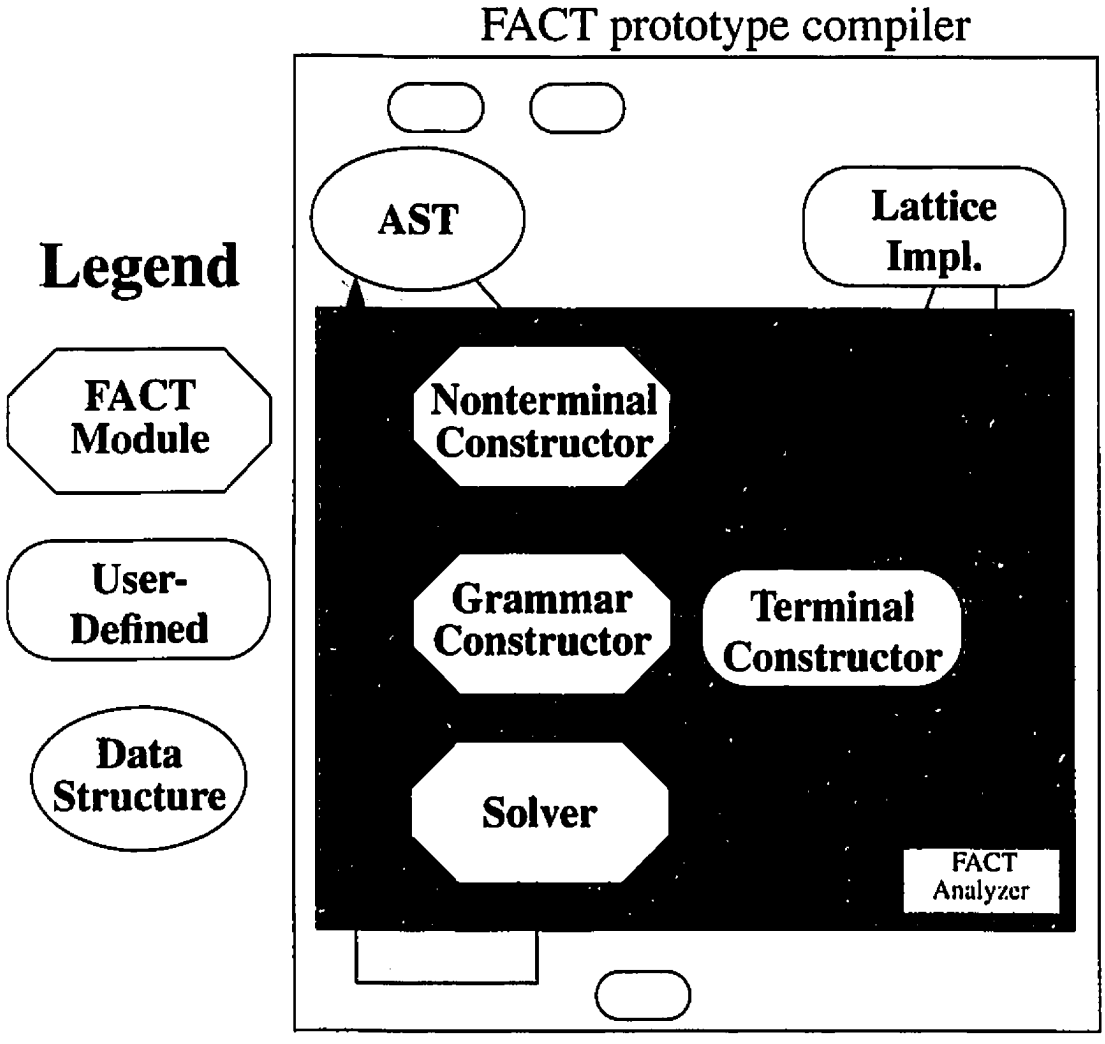
Currently, FACT consists of a single compiler for a Pascal-like language. It is implemented in ML in a style similar to that created by a compiler generation system such as Eli [27]. The scanner and parser were, in fact, constructed using ML versions of the LEX [58] and YACC [40] compiler generation tools. An abstract syntax tree is built as the source is parsed and further analysis is performed using left-to-right passes as might be found in an attribute grammar system.

Support for flow analysis is in the form of a set of modules for creating a FG from an AST, as well as several fixpoint modules. In a full implementation of the FACT tool, the FG modules would be generated from the AST-to-FG description. The fixpoint algorithms are implementations of two of the algorithms described in Section V.5.

Because the prototype was intended more as a test of the FG ideas rather than a production version for distribution, the interface to FACT is somewhat baroque. In FACT's current state, input modules to perform analyses look more like what a complete tool would generate (see Figure 49), rather than the abstract interface envisioned in the body of the dissertation. New analyses are created by writing a lattice implementation and then writing a terminal constructor. Several lattices have already been constructed, including a powerset lattice and a mapping lattice. Terminal constructors are relatively striaght-

forward to write by starting with an existing analysis and modifying it to create appropriate terminal function mappings. In the case of district analysis [76] (cf. Section VI.4), for example, there are three almost identical analyses that were all created from the live variable analysis specification. With experience, it is possible to create an intraprocedural analysis in an afternoon. Additions to allow interprocedural analysis take somewhat more time, as abstract parameter mapping requires considerable care. It should be noted that the various data structure implementations provided by the SML/NJ Library [1] proved invaluable for testing the algorithms.

Figure 49 Structure of a FACT flow analyzer



A.3 Example analysis: live variables

This section describes the FACT live variable implementation. First the intraprocedural variant is discussed by considering the FACT analysis of the example program in Section IV.2. Then the interprocedural variant is described using the recursive program from Section V.4.1 as the example.

A.3.1 Intraprocedural analysis

Consider once again the intraprocedural factorial program, reproduced in Figure 50. Initial analysis yields the AST shown in Figure 52, where distinct objects have been assigned unique integer values, as shown in Figure 51. As an example, the assignment "`i:=i-1`" is represented textually as:

```
stmtASSIGN
  {lvalue=exprIDENT 6,
   rvalue=exprMINUS
      exprIDENT 6,
      exprINT 1}
```

which is equivalent to the following more intuitive graphical representation:

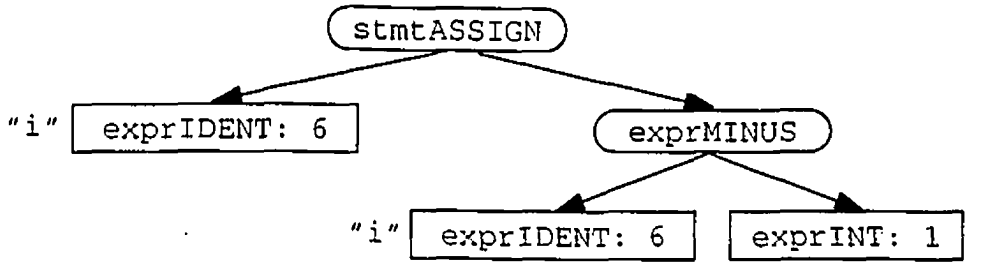


Figure 50 Factorial program

```
program test;
var    i:integer;
       f:integer;
       out:integer;
begin
   i := 3;
   f := 1;
   while i > 1 do begin
      f := f * i;
      i := i - 1
   end;
   out := f
end.
```

Figure 51 Identifier mapping

Identifier	Value
'integer'	4
'i'	6
'f'	8
'out'	10

Figure 52 AST of factorial program shown in Figure 50

```
(progOF
     (body=stmtBLOCK
              [stmtASSIGN (lvalue=exprIDENT 6,rvalue=exprINT 3),
               stmtASSIGN (lvalue=exprIDENT 8,rvalue=exprINT 1),
               stmtWHILE
                  (body=stmtBLOCK
                           [stmtASSIGN
                               (lvalue=exprIDENT 8,
                                rvalue=exprSTAR
                                         (exprIDENT 8,
                                          exprIDENT 6)),
                            stmtASSIGN
                               (lvalue=exprIDENT 6,
                                rvalue=exprMINUS
                                         (exprIDENT 6,
                                          exprINT 1))],
                     condition=exprGT (exprIDENT 6,exprINT 1)),
               stmtASSIGN(lvalue=exprIDENT 10,rvalue=exprIDENT 8)],
      declarations=[declVAR (6,typeNAME 4),
                    declVAR (8,typeNAME 4),
                    declVAR (10,typeNAME 4)])
```

The first step in computing the FG is the creation of a set of nonterminals for inclusion in the FG. Two nonterminals are associated with each node in the AST for this purpose, along with two additional nonterminals for each procedure declaration. In an attribute grammar system, attribution rules to compute these sets are straightforward, though tedious, to construct. Decorating the example program's AST with this information yields the tree shown in Figure 53: the pair of integers shown immediately after each node tag correspond the the "in" and "out" nonterminals. For example, "`stmtASSIGN ((48,49),...`" indicates that nonterminals S_{48} and S_{49} have been assigned to the above assignment statement. S_{48} represents the "in" point and S_{49} represents the "out" point.

Currently, the FACT prototype defines a canonical AST to FG transformation upon which analyses are based. Each individual analysis defines precisely where terminals need to be placed, permitting "projection" of the canonical FG. That is, various parts of the canonical FG may be replaced by terminals expressing the effect of the deleted canonical productions. In the case of live variables, for example, the entire assignment statement shown above is represented by a single terminal even though the canonical transformation has nonterminals and productions representing the evaluation of the entire subtree. The FG generated for the program is shown in Figure 54. Productions are represented in a fairly obvious manner. For example, the assignment statement ("`i:=i-1`") discussed above is represented as "`48-->[tm_fn,ntm_49]`" and corresponds to the production "$S_{48} ::= t_{48/49}\ S_{49}$" (i.e., "`tm_fn`" denotes a terminal, "`ntm_49`" denotes a nonterminal and, because FACT FGs are context-free, the lhs is always a nonterminal and need not be further distinguished). In this case, the sub-grammar representing the individual evaluation of the subcomponents of the of the assignment have been collapsed into the terminal $t_{48/49}$, the function itself ($\lambda x.(x-\{i\})\cup\{i\}$) computed by attribution. Finer granularity is, of course, possible at the expense of a larger flow grammar. The mechanism is quite general, permitting an arbitrary tradeoff between attribution and FG solution.

Figure 53 AST decorated with nonterminals

```
progOF ((62,63),
   {body=stmtBLOCK ((60,61),
      [stmtASSIGN ((16,17),
         {lvalue=exprIDENT ((12,13),6),
          rvalue=exprINT ((14,15),3)}),
       stmtASSIGN ((22,23),
         {lvalue=exprIDENT ((18,19),8),
          rvalue=exprINT ((20,21),1)}),
       stmtWHILE ((52,53),
         {body=stmtBLOCK ((50,51),
            [stmtASSIGN ((38,39),
               {lvalue=exprIDENT ((30,31),8),
                rvalue=exprSTAR ((36,37),
                  exprIDENT ((32,33),8),
                  exprIDENT ((34,35),6)))),
             stmtASSIGN ((48,49),
               {lvalue=exprIDENT ((40,41),6),
                rvalue=exprMINUS ((46,47),
                  exprIDENT ((42,43),6),
                  exprINT ((44,45),1)))))]),
          condition=exprGT ((28,29),
            exprIDENT ((24,25),6),
            exprINT ((26,27),1)))}),
       stmtASSIGN ((58,59),
         {lvalue=exprIDENT ((54,55),10),
          rvalue=exprIDENT ((56,57),8)})]),
    declarations=
      [declVAR ((2,3),6,typeNAME ((0,1),4)),
       declVAR ((6,7),8,typeNAME ((4,5),4)),
       declVAR ((10,11),10,typeNAME ((8,9),4))]})
```

Figure 54 FACT FG for factorial program in Figure 50

```
{rules=
[
 60 --> [ntm_16],
 16 --> [tm_fn,ntm_17],
 17 --> [ntm_22],
 22 --> [tm_fn,ntm_23],
 23 --> [ntm_52],
 52 --> [ntm_28],
 28 --> [tm_fn,ntm_29],
 29 --> [ntm_50],
 50 --> [ntm_38],
 38 --> [tm_fn,ntm_39],
 39 --> [ntm_48],
 48 --> [tm_fn,ntm_49],
 49 --> [ntm_51],
 51 --> [ntm_28],
 29 --> [ntm_53],
 53 --> [ntm_58],
 58 --> [tm_fn,ntm_59],
 59 --> [ntm_61],
 61 --> [ntm_63],
 63 --> []
],
start=60}
```

After construction, the FG along with an initial abstract state is passed to a fixpoint finder. The output is a mapping from nonterminals to abstract states. Attribution rules then distribute the results throughout the AST. Treating the example program as an intraprocedural problem yields the decoration shown in Figure 55. At the assignment "`f:=1`" the solution is represented as:

```
stmtASSIGN (([6],[6,8]),...)
```

indicating that `i` (denoted 6) is live before the statement while both `f` (denoted 8) and `i` are live after the statement.

Figure 55 AST decorated with Live Variable solution

```
progOF
  (([],[]),
   {body=stmtBLOCK (([],[]),
      [stmtASSIGN (([],[6]),
         {lvalue=exprIDENT (([],[]),6),
          rvalue=exprINT (([],[]),3)}),
       stmtASSIGN (([6],[6,8]),
         {lvalue=exprIDENT (([],[]),8),
          rvalue=exprINT (([],[]),1)}),
       stmtWHILE ([6,8],[8]),
         {body=stmtBLOCK (([6,8],[6,8]),
            [stmtASSIGN (([6,8],[6,8]),
               {lvalue=exprIDENT (([],[]),8),
                rvalue=exprSTAR (([],[]),
                  exprIDENT (([],[]),8),
                  exprIDENT (([],[]),6)})}),
             stmtASSIGN (([6,8],[6,8]),
               {lvalue=exprIDENT (([],[]),6),
                rvalue=exprMINUS (([],[]),
                  exprIDENT (([],[]),6),
                  exprINT (([],[]),1)})})]),
          condition=exprGT (([6,8],[6,8]),
                exprIDENT (([],[]),6),
                exprINT (([],[]),1)})}),
       stmtASSIGN (([8],[]),
         {lvalue=exprIDENT (([],[]),10),
          rvalue=exprIDENT (([],[]),8)})]),
     declarations=
       {declVAR (([],[]),6,typeNAME (([],[]),4)),
        declVAR (([],[]),8,typeNAME (([],[]),4)),
        declVAR (([],[]),10,typeNAME (([],[]),4))}})
```

A.3.2 Compressing the FG

The size of a FG can have a major impact on the amount of computation and storage required for its solution. It is clear from inspection of the sample grammar in Figure 54, however, that there are considerably more points than necessary (an artifact of deriving the control flow model from the AST). For example, the following list of productions forms a "chain" in which each of the non-terminals S_{50}, S_{38}, S_{39}, S_{48}, and S_{49} appear on the rhs of a single production.

```
29-->[ntm_50],

50-->[ntm_38],

38-->[tm_fn,ntm_39],

39-->[ntm_48],

48-->[tm_fn,ntm_49],

49-->[ntm_51],

51-->[ntm_28]
```

In a unidirectional problem,[1] this can be compressed to a single production:

```
29-->[tm_fn,tm_fn,ntm_28]
```

where the two terminal function are as in the productions above.

1.Section V.3.5 shows how this transformation may be inappropriate for bidirectional problem.

Applying this transformation to an entire FG is equivalent to forming basic blocks, as is traditionally done in data flow analysis [3]. It is straightforward to perform this algorithmically and can yield significantly smaller FGs, as shown in Figure 56 for the example program. The original FG has 19 nonterminals, while the compressed FG has only 4 nonterminals. Similarly, the original FG has 20 productions, compared with 5 for the compressed FG. In terms of finding a fixpoint to the flow problem, the compressed FG requires roughly 20% of the storage and considerably less computation, though the latter depends on the particular fixpoint algorithm used. As with control flow graphs, once the solution has been found at the points between basic blocks, computing flow values at points within a basic block is easy.

Figure 56 Compressed FG for example program

```
{rules=
 [
  60 --> [tm_fn,tm_fn,ntm_28]
  28 --> [tm_fn,ntm_29],
  29 --> [tm_fn,tm_fn,ntm_28],
  29 --> [tm_fn,ntm_63],
  63 --> [],
 ]
 start=60
}
```

A.3.3 Interprocedural live variables

Section VI.2 discusses the extension of intraprocedural live variable analysis to an interprocedural problem. An example of a program illustrating these ideas is shown in Figure 57, the corresponding identifier mapping appears in Figure 58. The live variable decorated AST for this program appears in Figure 59. As in the previous example, assign-

ment statements are treated as single entities, rather than separate evaluations of the r-value, l-value and subsequent assignment, for the purposes of flow analysis.

Figure 57 Program with interprocedural control flow

```
program example(output);
var f,j,out:integer;
procedure fact(i:integer);
begin
  if i<=1 then
    f := 1
  else begin
    i := i-1;
    fact(i);
    f := f*i
  end
end;

begin
  j:=5;
  fact(j);
  out := f
end.
```

Figure 58 Identifer mapping for program in Figure 57

Identifier	Value	Shadow
f	6	
j	8	
out	10	
fact	12	
i	14	15

Figure 59 Intraprocedural live variable AST for program in Figure 57

```
progOF
  (([],[([],[])]),
   (body=stmtBLOCK (([([],[])],[([],[])]),
      [stmtASSIGN (([([],[])],[([],[8])]),
        (lvalue=exprIDENT (([],[]),8),rvalue=exprINT (([],[]),5))),
       stmtCALL (([([],[8])],[([],[6])]),
        (arguments=[exprIDENT (([],[]),8)],
         function=exprIDENT (([],[]),12))),
       stmtASSIGN (([([],[6])],[([],[])]),
        (lvalue=exprIDENT (([],[]),10),
         rvalue=exprIDENT (([],[]),6)))]),
    declarations=
      [...
       declPROC(([],[]),12,
        (body=stmtBLOCK ((([([6],[14])],([6,15],[14,15]))],
                          [([6],[6])],([6,15],[6,15]))]),
           [stmtIFTE ((([([6],[14])],([6,15],[14,15]))],
                       [([6],[6])],([6,15],[6,15]))]),
              (condition=exprEQ ((([([6],[14])],([6,15],[14,15]))],
                                 [([6],[14])],([6,15],[14,15]))]),...),
               elsepart=stmtBLOCK ((([([6],[14])],([6,15],[14])],
                                    [([6],[6])],([6,15],[6,15]))]),
                 [stmtASSIGN ((([([6],[14])],([6,15],[14])],
                               [([6],[14])],([6,15],[14]))]),(...)),
                  stmtCALL ((([([6],[14])],([6,15],[14])],
                             [([6],[6,14])],([6,15],[6,14,15]))]),(...)),
                  stmtASSIGN ((([([6],[6,14])],([6,15],[6,14,15]))],
                               [([6],[6])],([6,15],[6,15]))]),(...))]),
               thenpart=stmtASSIGN
                 ((([([6],[])],([6,15],[15]))],
                   [([6],[6])],([6,15],[6,15]))]),(...))))]),
        declarations=[],
        proceduretype=typePROC(...))]))
```

Note that flow information in this AST is considerably more complicated than the information given in the previous tree (Figure 55). This is because a partial function has been computed for each point in the program, rather than a single live variable set. Consider the body of the main program:

```
body=stmtBLOCK ((([[],[])],[([],[])]),
  [stmtASSIGN ((([[],[])],[([],[8])]),{...}),
   stmtCALL ((([[],[8])],[([],[6])]),{...}),
   stmtASSIGN ((([[],[6])],[([],[])]),{...})])
```

The flow value for each point is a list of lattice pairs `(x,y)` where `y` is the flow value known to hold at the point when `x` is known to hold on exit from the main program. Because it has been assumed that there are no variables live at the end of the program and the main program is not recursive, there is a single calling context present: the empty set, represented as `[]`. At the point after the call statement the variable `f` is live (represented as `[([],[6])]`: "when nothing is live at the end of the main program, `f` is live at this point"). Similarly, `j` is live immediately before the call (`[([],[8])]`), and nothing is live at the beginning of the program (`[([],[])]`).

The procedure is more interesting: there are two calling contexts, one as a result of the call from the main program (`[6]`) and one from the recursive call (`[6,15]`). The latter indicates that the *shadow* of variable `i` is live at exit (along with `f`). At "entry" to the procedure the following two contexts hold:

```
([6],[14]) and ([6,15],[14,15])
```

That is, when `f` (6) is live on exit from the procedure, `i` (14) is live on entry. When `f` and the shadow of `i` (15) are live on entry, both `i` and its shadow are live on exit.

A.4 Algorithm implementations

While developing the FACT prototype, a variety of fixpoint algorithms were written. They may be classified broadly as two fundamentally different work pile fixpoint algorithms:

PF: "Partial Function Values" – each flow value is treated as an entire partial function.

IV: "Independent flow values" – each flow value is treated as a set of independent pairs.

For both of these algorithms, several variations have been written. Each of the following techniques are independent of each other and may be applied separately or in combination:

Comp: "Compression" – the FG may be compressed to basic blocks, as described in Section A.3.2.

Smart $\wedge$: "Smart meet" – the FG structure makes a simple optimization possible. Specifically, when recomputing a flow value for a point that depends on only a single value (that is, derived by only a single nonterminal in the forward case; on the lhs of a single production in the backward case), it is not necessary to take the meet with the old value. Instead of the meet step of Figure 29:

$$\zeta[X] \leftarrow \zeta[X][(newv \wedge_L oldv)/v]$$

it is occasionally possible to perform a straight replacement:

if #depends_on > 1 **then** $\zeta[X] \leftarrow \zeta[X][(newv \wedge_L oldv)/v]$

else $\zeta[X] \leftarrow \zeta[X][newv/v]$

{Queue}: "Queue sets" – the work pile is a queue. It can be naïvely implemented using the SML/NJ Library Queue module. Alternatively, it can be maintained with additional overhead in a data structure permitting access as both a queue and a set.

With the exception of linearly searching the queue for possible duplication, all possible combinations of the above techniques have been implemented, resulting in 16 different algorithms.

Determining the time complexity of the various algorithms is difficult, as the "shape" of the lattice and the set of possible functions can have a major impact on the time needed to solve a given FG analysis. In the following sections we content ourselves with the following measures:

$\wedge_L$: Number of "intraprocedural" lattice meets used to find fixpoint.

$=_L$: Number of "intraprocedural" lattice equality tests used to find fixpoint.

FTapps: Number of applications of flow terminal functions used to find fixpoint.

A.4.1 A "real" program

The FACT prototype, in spite of being implemented in ML, is capable of analyzing "real" programs. Naturally, performance varies with the particular problem being solved. The nature of the lattice and associated function space has a major impact on the sizes and kinds of programs that can be analyzed effectively.

To gain an appreciation of the abilities of the analyzer, Figure 61 shows a 45 line program (not counting blank lines), converted from a C program, that is analyzable with the FACT prototype. Figure 60 gives an indication of the size of the internal representation of this program within the prototype compiler. Run-times to perform the various interprocedural analyses on this program are on the order of seconds. It generally takes longer to generate the ASCII representation of the AST than it does to compute the solution and annotate the AST.

Figure 60 Characteristics of sample simulation program

Non-blank source lines	45
AST Nodes	141
Flow productions	
– uncompressed	136
– compressed	22

Figure 61 Program to simulate rolling of dice in board game

```
program risk(input,output);
  const SEED=1234;
        NTOSSES=100;
        M=65537;
        A=17129;
  label 10;
  var wins, nt, tosses:integer;
      i, j, d, e:integer;
      seed:integer;
      out:integer;
  procedure mysrand(initseed:integer);
  begin
    seed := initseed
  end;
  procedure myrand();
  begin
    seed := seed * A mod M
  end;
begin
  mysrand(SEED);
  wins := 0;
  nt := NTOSSES;
  tosses := 0;
  i := 0;
  while i < nt do begin
    myrand();
    d := seed mod 6;
    tosses := tosses + 1;
    j := 0;
    while j < 3 do begin
      myrand();
      e := seed mod 6;
      tosses := tosses + 1;
      if e > d then begin
        wins := wins + 1;
        goto 10;
      end;
      j := j + 1
    end;
10:
    i := i + 1
  end;
  out := wins;
  out := nt;
  out := tosses
end.
```

Table 2 shows the result of applying the FACT interprocedural live variable analysis to the example program using all the variants of the fixpoint algorithms. There are a number of features of note:

1. For the most part, performing grammar compression reduces the number of meets and equality tests. It does, however, increase the number of terminal function applications. The latter is due to the fact that on the last iteration entire "basic blocks" must be evaluated to determine when propagation is complete.
2. The "smart meet" technique proved especially beneficial in the partial function value case, and generally valuable overall by reducing the number of meets and equality tests required. Because it is straightforward to compute the "depends_on" set, this optimization should always be performed. Indeed, this technique alone made brought the partial function value algorithm in line with the results of the independent variable algorithm.
3. Maintaining the queue as a set proved beneficial for both algorithms. In the case of the PF algorithm, this is inexpensive, as sets of terminals may be represented using bit vectors. For the independent value algorithm, however, it is necessary to consider sets of FT×L pairs, which may entail some overhead to test the lattice value.
4. Overall, the IV algorithm appears to be the most appropriate for the interprocedural live variable analysis of this program. It should perhaps be noted that the PF algorithm performs fewer equality tests in certain instances, a reasonable lattice implementation should have an inexpensive equality test.

Table 2 Measurements for live variable analysis of simulation program

Algorithm				Statistics		
PF/IV	Comp	Smart $\wedge_L$	{Queue}	FTapps	$\wedge_L$	$=_L$
PF				93	246	385
PF			✓	91	222	327
PF		✓		93	79	54
PF		✓	✓	91	78	51
PF	✓			130	190	209
PF	✓		✓	93	122	115
PF	✓	✓		130	119	57
PF	✓	✓	✓	93	85	46
IV				86	157	146
IV			✓	72	136	114
IV		✓		86	70	155
IV		✓	✓	72	63	123
IV	✓			100	124	98
IV	✓		✓	79	96	66
IV	✓	✓		102	88	104
IV	✓	✓	✓	79	72	72

A.4.2 Analyzing procedure variables

Figure 62 shows a program used to test the use of shadow variables in combination with the "ignore" (i.e., ⊗) technique of Section VI.5. Careful examination of the program reveals that the procedure `ppp` calls each of the procedures `a`, `b`, `c` and `d`, in this order. Without the use of shadow variables (or something similar), it is not possible to discern this fact because the parameter `z` can take the value of any of these four procedures. Using shadow variables it is possible to compute this precisely. Table 3 shows the results of analyzing this program using the 16 algorithm variants.

Figure 62 Procedure variable example program

```
program p;

  var x:procedure; zzz:integer;

  procedure a; forward;

  procedure d; begin x := a end;

  procedure c; begin x := d end;

  procedure b; begin x := c end;

  procedure a; begin x := b end;

  procedure ppp(z:procedure);
  begin z(); if zzz then ppp(x) end;
begin
  ppp(a)
end.
```

Perhaps the first thing to notice is that the PF algorithms performed markedly worse than the IV algorithms for this problem on this program. Thus, considering only the IV algorithms, it is clear that maintaining the queue as a set was especially beneficial. In combination with smart meets, the results are quite startling: in the best case, only 63 meets and 298 equality tests are required. This compares favourably with the live variable analysis example of the preceding section: this particular algorithm seems the most suitable starting point for further investigations.

Table 3 Measurements for procedure variable analysis example

Algorithm				Statistics		
PF/IV	Comp	Smart	{Queue}	FTapps	$\wedge_L$	$=_L$
PF				3566	2570	13044
PF			✓	2774	1915	9955
PF		✓		3566	851	6880
PF		✓	✓	2774	710	5591
PF	✓			3658	2025	10397
PF	✓		✓	2673	1255	6805
PF	✓	✓		3658	789	6233
PF	✓	✓	✓	2673	595	4638
IV				1273	865	1766
IV			✓	257	161	298
IV		✓		1273	367	1766
IV		✓	✓	257	63	298
IV	✓			1644	918	1909
IV	✓		✓	320	154	309
IV	✓	✓		1655	393	1913
IV	✓	✓	✓	330	71	311

A.5 Summary

This appendix summarized and illustrated the FACT prototype, the existence of which validates the FG control flow model. The implementation has proven invaluable for exploring various analyses, as well as various fixpoint algorithms. Although more work is needed, initial results indicate that the "smart meet" and "queue set" techniques are worth implementing, and that, perhaps surprisingly, "grammar compression" may not be.

B Glossary

Term/Symbol	*Description*	*Page*
ANTLOC	local anticipability	166
AST	abstract syntax tree	2
AVAIL	availability	166
b	backward flow function	104
berFGP	backward interprocedural flow grammar problem	115
BerFGP	bidirectional interprocedural flow grammar problem	117
BraFGP	bidirectional intraprocedural flow grammar problem	104
CFG	context-free flow grammar	75
COMP	local availability	166
CONST	"fudge factor" for partial redundancy elimination	168
CP	constant propagation lattice	60
CPS	continuation passing style	38
DELETE	points to delete a computation (district analysis)	171
DISTRICT	district flow analysis system	170
dom	domain of a function	118
f	forward flow function	104
F	monotone function space associated with a lattice L	11
FACT	Flow Analysis Compiler Tool	2
ferFGP	forward interprocedural flow grammar problem	116
FG	flow grammar	65
F(L)	total function flow analysis lattice over L	72
FN	flow nonterminal	65
FP	flow productions	65
FT	flow terminal	65

Term/Symbol	*Description*	*Page*
GFP	greatest fixed point solution	79
G	grammar	9
$\mathcal{G}_{T,t}$	meaning of a flow grammar	71
INSERT	points to insert a computation (district analysis)	170
K_N	flow variable corresponding to N	75
L	lattice	11
LFP	least fixed point	62
LV	live variable lattice	59
MDFAF	monotone data flow analysis framework	24
MOP	meet over all paths solution	67
$\wedge_L$	meet operator on lattice L	10
N	arbitrary nonterminal	75
P	set of productions in a grammar G	9
PAVAIL	partial availability	166
PFL	partial function lattice	103
PV	procedure variable	172
RFG	regular flow grammar	75
R	right-reaches relation	84
S	start symbol in a grammar G	9
S_i	nonterminal in a flow grammar	10
SFN	start flow nonterminal	65
t	semantic functional of terminal symbols	54
T	set of terminal symbols in grammar G	9
t^*	semantic functional of terminal strings	56
$\mathcal{T}_{T,t}$	semantic function for sets of execution paths	61
TRANSP	transparency	166
υ	initial value in a flow problem	104
TFL	total function flow analysis lattice	72
V	set of nonterminal symbols in a grammar G	9
Var	set of variables in a program	59
ζ	flow solution	106