UNIVERSITY OF CALIFORNIA
BERKELEY

Graph Transformations and Program Flow Analysis

by

Douglas Richard Grundman

Contents

List of Figures

Chapter 1

Flow Analysis and Graphs

1.1 Introduction

Program flow analysis is a necessary component of optimizing compilers and software development environments. Algorithms abound for computing useful attributes of features of programs. Unfortunately, this area of computing also teems with complexity. Many individual algorithms are themselves extremely complicated, and the interrelationships present in a collection of algorithms can easily become a programming nightmare. The opportunities for confusion are only compounded by the low level of programming abstraction that is prevalent in such algorithms, as evidenced by the ubiquity of that data structure known as the bit-vector.

The result of this state of affairs is that a great many valuable algorithms do not make their way into compilers. Despite over thirty years of algorithm development, most modern optimizing compilers make use of no more than the four classical analysis algorithms and their associated optimizations that were known in the early 1960's. Valuable ideas such as interprocedural analysis or advanced constant folding techniques are often seen in papers, occasionally in research laboratories, and almost never in useful compilers. The same holds for software development environments: program slicing, flow-analysis based test generation, and other valuable methods of helping programmers are rarely if ever implemented.

The present dissertation is an attempt to confront these problems. In it is developed a graph-based methodology and language for writing flow analysis algorithms in a generic and portable fashion and for combining them with ease. Its minimalist stand with respect to data types and operators provides a high degree of compatibility between algorithms without sacrificing performance – either asymptotic or experienced.

1.2 Thesis

The goal of this dissertation is to convince the reader of the following claim.

Thesis: A programming language based on graphs and transformations on graphs is sufficient for performing flow analysis on computer programs in the following three senses:

1. *Representational sufficiency*: No other data structures (such as sets, bit-vectors or integers) are necessary for performing flow analysis.

2. *Computational sufficiency*: There is a small set of functions mapping graphs to graphs that is sufficient for performing analysis.

3. *Efficiency*: The use of graphs for performing analysis to the exclusion of other data types does not degrade the asymptotic time bounds of any algorithm. This exclusiveness entails at most a small constant factor slowdown over hand-crafted algorithms.

This work describes the development of a suitable family of transformations and a programming language based on them, and shows how this language may be used to implement higher-level functions on graphs and useful flow analysis algorithms. It also describes our implementation of this language, and our success in performing flow analysis with the resulting system.

The remainder of this chapter reviews the state of the art, provides motivation for the rest of the presentation, and supplies necessary definitions. Section 1.3 gives an overview of program flow analysis concepts, covering basic notions of control-flow and data-flow analysis. Section 1.4 explains many of the difficulties inherent in the implementation of analyzers, especially when multiple flow analysis algorithms must be incorporated into a single program. Section 1.5 provides a historical perspective, exhibiting not only the classical analysis algorithms but also several more advanced algorithms that rarely see implementations. Section 1.6 motivates our graph-based approach, providing some understanding of why the above-stated goal should be attainable. Section 1.7 provides a complement to that section by describing several benefits that accrue from using such a language. Finally, Section 1.8 provides fundamental definitions that are necessary for the remainder of this dissertation.

Chapter 2 gives the graph-theoretic foundation of the present approach, developing well-defined notions of graph pattern matching and of graph transformations. While it may seem to be excessively abstract, such a treatment is worthwhile in that it ensures that the transformations upon which our entire approach is based are indeed well-defined functions. Chapter 3 complements Chapter 2 by presenting algorithms that implement our desired graph pattern matching operations and our graph transformations. We will see that each implementation of these operations yields a family of graph functions. Rather than choose one particular implementation, the chapter gives "generic" algorithms that may be *parameterized* in a meaningful way to emulate a large class of implementations, thus providing a great deal of flexibility and power to the end-user.

Chapter 4 describes our programming language based on these pattern matching operations and transformations. Chapter 5 uses this language to demonstrate that despite its small size, it can be used to implement higher-level operations on graphs, as well as some common graph algorithms. All of these functions are then considered to be extensions to the language, and are used freely in the following chapter.

Chapter 6 is the culmination of Chapters 2–5, presenting implementations of several advanced flow analysis algorithms. It is shown that these algorithms are easy to code, to read, and to inter-mix, and that they are as efficient (in the asymptotic sense) as the original versions. That exposition provides the primary evidence for the thesis of this dissertation. Finally, Chapter 7 discusses our implementation of the language and the installation of our system into an optimizing compiler. It demonstrates the ease with which our system may be used, and that graph-based analyzers can be efficient in a real sense as opposed to simply a theoretical sense. It closes with a summary of the major points of this dissertation, and gives several directions for future research.

1.3 Fundamentals of flow analysis

Program flow analysis is the compile-time determination of the run-time behavior of programs. While this problem is not decidable in the general case, it is possible to determine much information about a program's behavior at compile time. Examples might include facts such as "variable x is always equal to 67" or "statement 100 is never executed."

The determination of such information is important because it can enable a compiler or a programmer to improve a program substantially. Information about the flow of data in a program, such as the use or modification of particular values, can be used by an optimizer to make a program smaller and faster without compromising its semantics. For example, references to a variable whose value is determinable at compile-time can be replaced by the pre-computable value, and unexecuted sections of code can be removed.

The term "flow analysis" really covers two domains. *Control-flow analysis* deals with the determination of the possible flow of control through a program. That is, it concerns itself with the conversion of a computer program from a static representation such as text or parse trees to a directed graph representation that encodes the possible flow of control. Nodes in such a graph represent blocks of code such as statements or expressions, or sequences of statements, and (directed) edges represent possible transfers of control between those blocks. The most common example of control-flow graphs in optimizing compilers is that of the *procedure flow graph*, in which nodes represent basic blocks (straight-line sequences of statements containing no jumps) and edges represent conditional branches, unconditional branches, or sequencing from one block to the next. Another example is the *program call graph* in which the graph nodes represent entire procedures, and edges represent calls from one procedure to another.

The word *possible* as a modifier of "control flow" is important, since it is usually not possible to determine the exact flow of control through a program. Such a determination would be tantamount to solving the halting problem, which is well known to be undecidable. Instead, a control-flow analyzer creates a conservative approximation to the actual flow of the program: every path that the program may take exists in the graph, but not all paths through the graph represent valid program executions. Thus, the smaller the graph a control-flow analyzer can build, the better that graph approximates the real program flow.

The main purpose for doing control-flow analysis, as far as compiler-writers are concerned, is to produce a flow graph for use in *data-flow analysis*, the second domain of program flow analysis. Data-flow analysis is the compile-time determination of run-time behavior of values in the program. A data-flow analyzer collects information at each node in the graph, then propagates this information around the graph, simulating the action of the program on the collected information to the degree that it can. The result of this process is that each node in the graph has associated with it a set of facts that are known to be true whenever that node is executed. Examples of data-flow analysis algorithms are *constant propagation*, which attempts to determine constant values for variable references, and *MOD set computation*, which attempts to find for each statement in the program a minimal set of variables that the statement may modify.

Most data-flow analysis algorithms adhere to the following paradigm:

1. Construct some universal set of facts U whose truth or falsity we wish to prove about various blocks in the program.

2. Let G be a flowgraph constructed by some control-flow analysis algorithm. Associate with each node n in G sets in_n and out_n (initially empty). These sets will assume values in the power set of U, and at the termination of the algorithm will represent facts that are provably true at the entry and exit, respectively, of block n.

3. For each node n in G, construct a function $f_n : 2^U \rightarrow 2^U$. This function will model the action of block n on facts in U, and will depend only on the contents of that block. In the most frequently implemented algorithms, these functions operate by validating some facts, invalidating others, and ignoring yet others.

4. Choose a function C (usually $\cup$ or $\cap$) for combining subsets of U at flowgraph confluences. Whenever multiple edges enter a node, C is used to combine the sets of facts true along each edge into a set of facts that is necessarily true on entry to that node.

5. Finally, propagate information around the graph until no new information is discovered anywhere. Information is propagated through a node n by applying f_n to in_n to produce out_n; it is propagated along edges by pushing the out_n sets along those edges, using C when necessary to combine multiple propagated out_n sets to produce new in_m sets.

Control-flow analysis and data-flow analysis are generally considered separately, but they can interact, especially when flow of control depends partly on the flow of data. This interaction happens, for example, when a function call is made indirectly through a variable that may refer to one of several functions in a program. A notable case is that of SCHEME [21], wherein almost all flow of control depends on the values of function-valued variables. Writers of compiler optimizers have found it convenient, however, to regard the two separately as this simplifies the problem of analysis (although at the expense of potentially useful information). One historical reason for this dichotomy is that most optimizers have been written for FORTRAN, which has no function-valued variables. This argument is supported by the fact that we see few if any good optimizers in use for languages such as LISP that make heavy use of function-valued expressions.

1.4 Difficulties in analyzer construction

In spite of the apparent simplicity with which most data-flow analysis algorithms may be specified, there is a great deal of complexity in their manufacture. First, a flowgraph must be constructed; a non-trivial task if function-valued variables are a part of the language being compiled. Second, efficiency concerns may cause the algorithms that propagate information around flowgraphs to be quite involved, since loops and other features of the flowgraph can cause simpler algorithms to have longer running times. And third, the construction of the functions f_n requires the availability of myriad local facts that need to be gleaned from the program. Less standard algorithms may have other complexities as well. For example, the set U of facts may not be determinable prior to the propagation phase; an example is the set of constant values discovered during constant propagation.

Yet another complication is that not all analyzers operate on a procedure flow graph. A *program call graph* models possible function calls by representing functions as nodes and possible calls as edges, and is used for dealing with interprocedural data-flow analysis problems. An example of one of these problems is determining which variables in a program are modified by a call to a function. A *dependence graph* models temporal data dependencies rather than temporal control dependencies: "A's new value depends on B" rather than "The program counter reaches X after reaching Y." A *global value graph* models the flow of values through a program, and may be used to make constant propagation efficient.

Each of these problems, and there are others, requires its own special algorithm for efficiency, and adds its own special requirements to the list of complexities. With interprocedural analysis, for example, complications such as aliasing due to parameter passing arise. Some of the faster propagation algorithms require their input graphs to be *reducible*, that is, lacking forward jumps into loops, and do not guarantee their fast time bounds when handed a non-reducible graph. While this restriction isn't a problem when handling procedure flow graphs (that tend to be reducible), other graphs such as program call graphs are hardly ever reducible, making the choice of a single good propagation algorithm difficult.

It would seem that with this long list of difficulties, hardly any flow analysis would ever get done, since the programming task is so complicated. And indeed, this is the case. Most optimizing compilers perform only the simplest intraprocedural analysis, the algorithms for which date back to the 1960's [1]. There are a great many algorithms found in the literature that simply never make it into compilers that are available for people to use [2, 3, 4, 6, 7, 9, 11, 13, 16, 19, 24, 25]. The main reason for this non-use is not just the abovementioned complexity, but the added complexity of combining multiple algorithms having different data structures and different propagation methodologies. Many algorithms deal with information represented at the level of bits as an attempt to save memory and time, a practice that leads to immense difficulties when one attempts to merge two or more of them.

On the other hand, it is not always known whether the implementation of a new algorithm will be worthwhile. There are, for example, very few research results indicating the effectiveness of interprocedural summary information in optimization. But the main reason for this lack of information is the daunting programming task required of any researcher wishing to do the experiment.

One final difficulty is that facing anyone attempting to choose among several different algorithms for solving the same problem. For example, barring programming expense, one might wish to know the fastest propagation technique for computing available expression information. Such information is simply not available. It is known which algorithms are asymptotically fastest, but their constant factors are unknown and may outweigh their asymptotic behavior on typical analysis problems. Not only is this information unavailable, it is practically unobtainable due to difficulties controlling for the effects of differing implementations when comparing two analysis algorithms. The result of this state of affairs is that only the easiest-to-implement algorithms are ever used.

1.5 Historical perspective

This section gives a brief overview of known data-flow analysis algorithms, from which it can be seen that few algorithms ever see use, and those that do are typically chosen for implementational reasons rather than for the results or speed they afford.

1.5.1 The iterative algorithms

According to Aho and Ullman [1], the first iterative data-flow algorithm seems to have been developed by Vyssotsky around 1960 for an early Fortran compiler. The mid-to late 1960's saw the development of the four classical intraprocedural analysis algorithms – reaching definitions, available expressions, live variable analysis, and very busy expressions – based on an iterative approach. These same algorithms are the most commonly used in optimizing compilers. Several techniques were later developed to make iterative techniques more efficient (e.g. worklists, depth-first ordering), but there were no advances made in making these techniques generally useful outside of the "classical" intraprocedural optimization problem domain.

In 1973, Kildall [18] formalized the ideas behind data-flow analysis by setting the problem in a lattice-theoretic framework. He also devised new algorithms, including the first published constant propagation algorithm. His generalizations helped explain the data-flow propagation algorithms that existed, but did not aid in managing the complexity of the propagation or information collection tasks.

Morel and Renvoise [20] unified redundant code elimination with loop-invariant code motion under a single data-flow umbrella with their work on partial redundancy elimination. Their iterative algorithm combined upward flow analysis, *anticipability*, with a downward analysis, *partial availability*, to locate good insertion and deletion points for expressions. This algorithm was implemented in Chow's portable optimizer, UOPT [5] to good advantage. It is a powerful and useful algorithm, but its computational complexity is so far unknown, due to the intermixing of upward and downward information flows.

1.5.2 The elimination algorithms

In 1970, Allen [2] and Cocke [6] developed a different method of performing the propagation step of data-flow analysis that they called *interval analysis*. Their idea was to convert a problem on a (reducible) flow graph to a problem on a smaller flow graph by collapsing *intervals* in the original flow graph to single nodes. Here, an *interval* is a single-entry region of a flowgraph corresponding roughly to a program loop. The interval-analysis algorithm can be used recursively on the reduced graph as many times as needed until all loops cease to exist. At this point, the problem of propagating information along the graph becomes trivial. Finally, information at each collapsed node is used to deduce information about the nodes in the corresponding interval. This algorithm's worst-case performance is $O(n^2)$, but in practice, where loop depth is generally bounded by a constant, performance is linear.

The original algorithm had a limitation in that it could be used only on reducible flow graphs. Since most procedure flow graphs are reducible, this restriction does not usually cause difficulties to a flow analyzer used for solving one of the classical problems. The restriction did pose a limitation on the algorithm's generality, though, as it could not be used on non-reducible graphs such as program call graphs. Allen and Cocke later extended the algorithm to handle irreducible graphs, although at the cost of pessimizing of its worst-case time bounds on those graphs.

Hecht and Ullman [16], Tarjan [22], and Graham and Wegman [13, 24] provided improvements to the basic interval algorithm, each modifying the definition of "interval" slightly, and introducing auxiliary data structures or interval reduction orders to lower the computational complexity of the worst-case performance. They did not, however, change the basic model of the algorithm.

Graham and Wegman also helped to provide a measure of abstraction, separating the notion of data-flow analysis from that of procedure flow graphs by restating data-flow analysis problems as "Information Propagation Problems". They viewed these problems as the search for a maximal set of assignments of assertions about the program to nodes in the program's flow graph, given, for each node in the flow graph, a function relating that node's input to its output. This characterization exposes some measure of generality in that it divorces the information propagation aspect from the fact that the input graphs are typically program flow graphs.

Elimination (interval-based) data-flow techniques are generally believed to be more efficient than their iterative counterparts on typical inputs when both algorithms apply, but there seems to be no hard data supporting this claim. Several authors (e.g. [15]) openly consider the added programming complexity to decide the issue in favor of the iterative approach in the face of this lack of data. This argument probably explains why elimination algorithms are rarely used in practice.

1.5.3 Other algorithms

Banning [3], and later, Cooper and Kennedy [7, 8] showed that interprocedural analysis can be accomplished by examining the program's call graph instead of the flow graph. The most important interprocedural problems are to determine the set of variables that may be modified by a function call and the set of variables that may be referenced during a function call so that optimization at call sites can take place. This information is also useful in determining whether or not two functions can run simultaneously on parallel machines. Cooper and Kennedy's solution works in almost linear time by breaking the problem into two sub-problems each of which may be solved efficiently by the application of a carefully crafted algorithm.

Wegman and Zadeck [25] developed a novel algorithm for constant propagation that is interesting in two respects. First, it propagates information along a graph called the GlobalValue graph instead of the usual procedure flow graph. This graph is a modification of a DefUseChain graph that can in turn be extracted from a flowgraph by means of a different data-flow analysis procedure. This is an example of a flow analysis algorithm that works on something other than a control-flow graph. The second interesting point is that the algorithm evaluates conditional expressions once these are known to be constant, and marks branches of the flowgraph as being non-executable if it determines that they indeed are. Thus, it "prunes" the flowgraph while analyzing, producing a flowgraph that represents a better approximation to the actual program flow.

Several researchers (e.g. [11, 19]) have experimented with various types of dependency graphs to take the place of control-flow graphs in flow analysis. These graphs represent both control-flow and data-flow in one graph, and make some types of analysis (especially that required for automatic parallelization of code) much easier.

It is unfortunate that only the iterative algorithms of subsection 1.5.1 are frequently used. Many more algorithms have been developed in the last two decades, but few if any of them have been incorporated into real optimizing compilers. One main reason for this is that these algorithms require a great deal more effort for their implementation than do the simple iterative algorithms. A second reason is that any optimizing compiler is likely to require the four classical analyzers; and that adding a fifth analyzer to the fray, one that does not fit the mold of the other four, multiplies the effort needed nontrivially. To sum up, the complexity of implementing and maintaining multiple analyzers inhibits the use of all but the most straightforward.

1.6 Motivation for a graphical approach

Flow analysis deals with a great many problems with a great many different algorithms. One evident commonality, however, is that all flow analysis algorithms operate on some sort of graph, be it a procedure flow graph, some derivative thereof, or another graph such as a call graph or dependence graph. In view of the fact that flow analysis seeks to uncover facts about programs that have to do with a network representation of a program, it seems natural to speak of flow analysis as the analysis of program properties that can be expressed naturally with graphs. This statement would seem to imply not only that graphs should appear in our analysis algorithms, but also, since we are not interested in properties that cannot be expressed graphically, that it should be possible to express those algorithms using nothing but operations on graphs.

To illustrate, the input to a data-flow analysis algorithm is typically a description of the input program in the form of a graph that has been built to whatever level of detail is relevant for the analysis to be performed. The output of this algorithm is usually a list of the program's nodes, each having associated with it a set of related facts. This output list may also be considered to be a directed graph each of whose nodes represents either a program node or a fact, and each of whose edges leaves a program node to enter an associated fact node. Viewing data-flow in this way, it thus appears that our data-flow algorithm is a program that can transform a given graph or set of graphs into some other graph or set of graphs.

For example, Def-Use information is a mapping from variable definitions (that is, assignments to variables) in a program to possible uses of those definitions. Given a flow graph G, we can represent Def-Use information by a directed graph. First, we let each instance of each variable in G be represented by a unique node of the Def-Use graph. Then, for each variable v in the program, we let each path in G from a node defining v to a node referring to v be represented by an edge in the DefUse graph if that path contains no other definitions of v. The edges in the DefUse graph can then be seen to represent exactly the mapping we require.

Another example is the DMOD sets of Cooper and Kennedy [7]. DMOD assigns to each call site in the program the set of variables that might be changed by execution of that call. One can view the solution as a directed graph whose set of nodes is $S \cup V$ where S corresponds to the set of call sites and V corresponds to the set of variables. An edge $s \rightarrow v$ is in our graph if and only if $s \in S$ corresponds to a call site, and $v \in V$ corresponds to a variable that is possibly changed by a call from that call site.

We see that the output of a program flow analyzer may be represented as a graph or set of graphs even as its input may. It makes sense, therefore, to view a flow analyzer as an algorithm that transforms its input graphs to its output graphs. Viewing an algorithm as a set of sub-algorithms leads one to believe that any flow analysis algorithm may be built-up from primitive graph transformations. This avenue is the direction explored by this work.

1.7 Benefits of a graphical approach

The practical importance of a language based on a graphs and a small family of primitive transformations is that such a language can provide a uniform base upon which to build analyzers, in the sense that all data formats would ultimately be the same, and that all algorithms would ultimately be expressed with respect to the same underlying formalism. The uniformity of data formats and implementation techniques that would result from such a language would provide the following benefits:

1. Flow analyzers would become much easier to build and use. For many common applications, an existing algorithm could be used from a library. This language extension mechanism is feasible because all algorithms would work on the same data formats.

2. Such a programming language would provide a higher degree of data abstraction than is currently used for flow analysis algorithms. In other words, programmers would work with graphs instead of bits.

3. The interface to the application program for which the analysis is being done would be simple and standard, allowing for easy use. The reason for this is that the directed graph is a simple and easily understood data structure. Since no other data structures would be involved, the system would require a user to learn no other data formats.

4. Since all data would be in a standardized format and there would be a clean break between the analyzer and the application, analysis algorithms could be changed, modi-

fied, or augmented without disturbing the application. In particular, other algorithms could be added, sharing data with preexisting algorithms without need for elaborate data format conversion coding.

5. A common implementation base would allow algorithms to be compared with one another in a meaningful way, opening up avenues of research.

6. Finally, since a static representation of a program such as a parse tree or abstract syntax tree can also be represented by a graph, the language could aid in performing control-flow analysis and the construction of the functions f_n that characterize flow-graph nodes. Dealing with this drudgery would help make flow analysis a standard part of compilers and programming environments, rather than an option.

These benefits will be exhibited in chapters 6 and 7.

1.8 Basic definitions

A *directed graph* $G = (N, E)$ will be a finite collection of *nodes* N and *edges* $E \subseteq N \times N$ such that for any proper subset N' of N, $E \not\subseteq N' \times N'$. In this work, the term *graph* will always mean a directed graph. We will write $a \rightarrow b$ for the edge from node a to node b.

All graphs will be over some large universal set of nodes U in the sense that if $G = (N, E)$, then $N \subset U$. Thus, it is possible to have more than one graph on the same set of nodes. Accordingly, one may view a graph as simply a finite set of (directed) edges on this set of nodes. As a notational convention, we will use lower-case letters near the beginning of the alphabet to represent graph nodes.

This definition allows graphs to contain edges of the form $a \rightarrow a$, termed *looping edges*. This is somewhat non-standard in graph theory, but is needed for our treatment of flow analysis of computer programs, where one-statement loops are commonplace.

A *subgraph* of a graph $G = (N, E)$ is a graph $G' = (N', E')$ such that $N' \subseteq N$ and $E' \subseteq E$. A *region* of a graph $G = (N, E)$ is a subgraph $G' = (N', E')$ of G such that for a and b in N', if $a \rightarrow b$ is in E then $a \rightarrow b$ is in E'.

We say two nodes a and b are *connected in G* if either $a = b$ or there is some sequence of nodes $a = n_0, n_1, \ldots, n_k = b$ such that for each $1 \leq i \leq k$, at least one of the edges $n_{i-1} \to n_i$, $n_i \to n_{i-1}$ lies in G. A graph G is *connected* if every pair of its nodes are connected in G. A *connected component* of a graph $G = (N, E)$ is a maximal connected region in G.

The *predecessors* of a node a in graph G are the nodes b such that $b \to a$ is an edge in G. Similarly, the *successors* of a are those nodes b such that $a \to b$ is an edge in G. A *path* in a graph G is a sequence of nodes $a_0, a_1, \ldots, a_n$ such that a_i is a predecessor of a_{i+1} for $i \in \{0, \ldots, n-1\}$. A *cycle* is a path $a_0, a_1, \ldots, a_n$ where $a_0 = a_n$. A graph is *acyclic* if it contains no cycles.

A *flow graph* is a graph G having a distinguished node s_0 called the *start node* such that for any node b in G there is a path $p = a_0, a_1, \ldots, a_n$ where $a_0 = s_0$ and $a_n = b$. In a flowgraph G, node *a dominates* node b if a appears in every path from s_0 to b.

A *spanning tree* for a flowgraph G is a connected acyclic sub-flowgraph $G' = (N, E')$ such that $|E'| = |N| - 1$. A *depth-first spanning tree* for a flowgraph is a spanning tree for that flowgraph such that some depth-first ordering of the nodes of the spanning tree is a depth-first ordering of the nodes of the flowgraph. A *frond* in a flowgraph G with respect to a spanning tree $G' = (N, E')$ is an edge $a \to b$ in G such that b dominates a in G'. An *s-numbering* of the nodes in G with respect to a depth-first spanning tree $G' = (N, E')$ is a bijection from the integers $\{1, \ldots, |N|\}$ to the nodes of G such that for any edge $a \to b$ in G, *s-number*$(b) <$ *s-number*(a) if and only if $a \to b$ is a frond with respect to G'.

An *adjacency matrix* for graph G is a square matrix $[G]$ with entries in $\{0, 1\}$ whose sequence of rows and sequence of columns are both considered to be labeled by a sequence of graph nodes. If a and b are graph nodes in this labeling, then the notation $[G]_{a,b}$ refers to the entry in the row and column of $[G]$ labeled by a and b, respectively. The value of $[G]_{a,b}$ tells whether or not there is an edge in the graph leaving node a and entering node b: a 1 entry means that this edge exists, while a 0 means that it does not.

Chapter 2

Patterns and Transformations

This chapter lays a graph-theoretic foundation for the graph transformations used in Chapters 5 and 6 for performing flow analysis. The paradigm used for transformations is that of pattern-matching and replacement. Section 2.1 explains the underlying graph pattern-matching scheme, while Section 2.2 defines graph transformations. The latter are developed by merging pattern matching with a family of graph functions that perform graph modifications.

2.1 Graph Pattern Matching

We begin with a very simple method of matching patterns in a graph, then conclude with a generalization that makes patterns simpler to specify and more powerful.

2.1.1 Matching with templates

Definition 1 A *template*, T, is a square matrix with entries in the set $\{0, 1, *\}$, along with a set L of labels, a labeling function bijectively mapping the columns of the matrix to L, and a labeling function bijectively mapping the rows of the matrix to L. (As an abuse of notation, T will also denote the matrix of the template T.)

A template is a very primitive specification for a set of structural graph characteristics. It is nothing more than an adjacency matrix with two minor extensions: first, the entries in the matrix may take on values in the set $\{0, 1, *\}$ rather than just the two values 0 and 1. Second, the rows and columns of the matrix are labeled so that the matrix has an easily referred-to "basis". We let $T_{p,q}$ denote the entry of T in the row labeled by p and column labeled by q. This parallels the definition of adjacency matrix entries given in Chapter 1.

Definition 2 An *acceptable template match* is a pair (G, T) where $G = (N, E)$ is a graph and T is a template whose matrix is of size $|N| \times |N|$.

Definition 3 A *successful template match* is a triple (G, T, β) where (G, T) is an acceptable template match and β is a bijection from L, the labels of T, to the nodes of G such that for any two labels x and y in L and for nodes $a = \beta(x)$ and $b = \beta(y)$ in G,

$$T_{x,y} = 1 \quad \Rightarrow \quad [G]_{a,b} = 1, \text{ and}$$

$$T_{x,y} = 0 \quad \Rightarrow \quad [G]_{a,b} = 0.$$

The condition $T_{x,y} = *$ specifies nothing about the state of $[G]_{a,b}$. If no such bijection exists, then the acceptable template match is said to be *unsuccessful.*

We say that a template *successfully matches a graph* if and only if there exists a β such that (G, T, β) is a successful template match. In other words, a template successfully matches a graph if and only if there exists a one-to-one correspondence between the template labels and the graph nodes such that a 1 (respectively 0) in the (x, y) position of the template matrix implies the existence (respectively non-existence) of the edge $\beta(x) \rightarrow \beta(y)$ in G. If a $*$ is in the (x, y) position of the template matrix, then neither the existence nor the non-existence of the edge $\beta(x) \rightarrow \beta(y)$ in G can cause the template to fail to match G.

Templates and template matching do not appear to be very useful because of the constraint that the size of the template must be equal to the size of the graph. One obvious objection to their use is that the specification of a template to match a typical flow graph is much too big to be practical. Further, since we will be interested mostly in small localized features of graphs, most useful template matrices would consist mainly of $*$'s. We also have the problem of not knowing at the time of template construction the size of the graph to be matched. Below, we present a formalism from graph theory that allows us to ignore a great many extraneous template entries in a mathematically structured way, and also to match graphs of varying sizes.

Definition 4 A *simple graph homomorphism* is a mapping ψ from a graph G onto a graph H such that for some collection of nodes K in G,

(i) For every pair of nodes $a, b \in K$, $\psi(a) = \psi(b)$,

(ii) For every node $a \notin K$ and for every node $b \neq a$, $\psi(a) \neq \psi(b)$, and

(iii) For each edge $a \rightarrow b$ in G, the edge $\psi(a \rightarrow b) = \psi(a) \rightarrow \psi(b)$.

The set K is called the *kernel* of the homomorphism.

Hence, a simple homomorphism maps some set K of nodes to a single "supernode" while acting bijectively on all others.[1] It should be noted that any simple homomorphism, if restricted to any subset of nodes not intersecting its kernel, acts as an isomorphism on those nodes and on edges among them. In particular, the case kernel(ψ) $= \emptyset$ implies that ψ is an isomorphism on G; thus, the identity map itself is a simple homomorphism. Note also that as a consequence of (iii), if there is an edge $a \rightarrow b$ in G with both a and b in K, then the image of ψ will contain a looping edge at the supernode.

The concept of simple graph homomorphism can be used along with that of successful template match to define a useful pattern matching operation on graphs. Simple homomorphisms allow the examination of portions of a graph (that is, those portions of the graph upon which the homomorphism acts like the identity function), while providing a convenient mechanism for ignoring the rest.

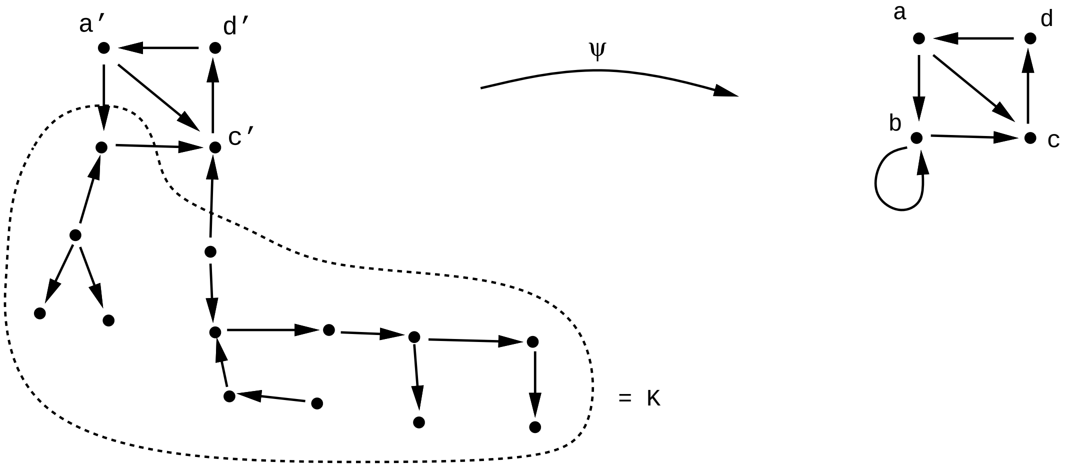
Definition 5 A *successful simple match* is a quadruple (G, T, β, ψ) where ψ is a simple graph homomorphism such that $(\psi(G), T, \beta)$ is a successful template match.

Example 1 The template

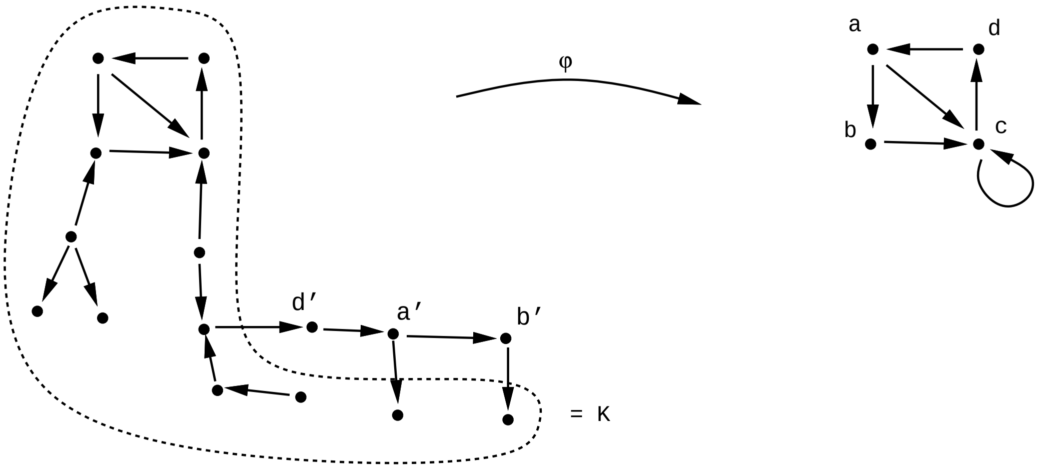
	w	x	y	z
w	*	1	1	*
x	*	*	1	*
y	*	*	*	1
z	1	*	*	*

[1] This definition extends the usual definition of graph homomorphism. The usual definition imposes the restriction that adjacent nodes can never be merged together (see [14]). The definition given here is in keeping with the relaxed definition of graph given in Chapter 1, that allows the existence of edges of the form $x \rightarrow x$.

matches the graph G below via the simple homomorphism ψ shown, with $\beta(w) = a$, $\beta(x) = b$, $\beta(y) = c$ and $\beta(z) = d$. The node b is the supernode formed by collapsing the kernel K. Notice that each edge in K leaves some node that maps to the supernode and enters some node that maps to the supernode. Thus, the image of ψ has a looping edge at node b.



Example 2 The template in the previous example also matches the same graph G via the simple homomorphism φ shown here, with $\beta(w) = a$, $\beta(x) = b$, $\beta(y) = c$ and $\beta(z) = d$. This is unfortunate, because we would expect and like the template shown to find loops in G, not construct them.



2.1.2 Patterns

We add an additional constraint to the concepts of template and simple match in order to construct a more versatile kind of matching that will help us avoid the problems of the preceding example.

Definition 6 A *simple pattern* is a template with a distinguished row and column label, denoted by the symbol $*$. All simple pattern matrix entries in any row or column labeled with $*$ must be $*$'s.[2]

The restriction that matrix entries in $*$ rows and columns must be $*$'s will be lifted in Definition 11 (see Subsection 2.1.3).

Definition 7 A *successful simple pattern match* is a successful simple match (G, P, β, ψ) where P is a simple pattern, and $\beta(*) = \psi(kernel(\psi))$. A *simple pattern matching algorithm* is an algorithm that, given a graph G and simple pattern P as input, finds a suitable label-to-node bijection β and simple graph homomorphism ψ (if such exist) so that (G, P, β, ψ) is a successful simple pattern match.

The constraint we have just added is that the supernode constructed by coalescing the graph nodes in the kernel K must necessarily correspond to the distinguished label $*$ in our simple pattern. The set of simple homomorphisms ψ for which (G, P, β, ψ) is a successful simple pattern match is thus seen to be a subset of the set of homomorphisms for which it is merely a successful simple match.

Example 3 The simple pattern

```
      x   y   z   *
     ----------------
x |   *   1   0   *
y |   *   *   1   *
z |   *   *   *   *
* |   *   *   *   *
```

matches a graph G if and only if G contains nodes a, b, and c and edges $a \rightarrow b$ and $b \rightarrow c$, but not the edge $a \rightarrow c$. Note that the row labeled z and the column labeled x contain only $*$'s and therefore specify nothing. As a convention, we allow such rows and columns (with the exception of the $*$ row and column) to be elided from the specification. The $*$

[2] One should not find these two meanings for $*$ confusing, since the contexts in which they are used, row/column label versus matrix entry, are disjoint.

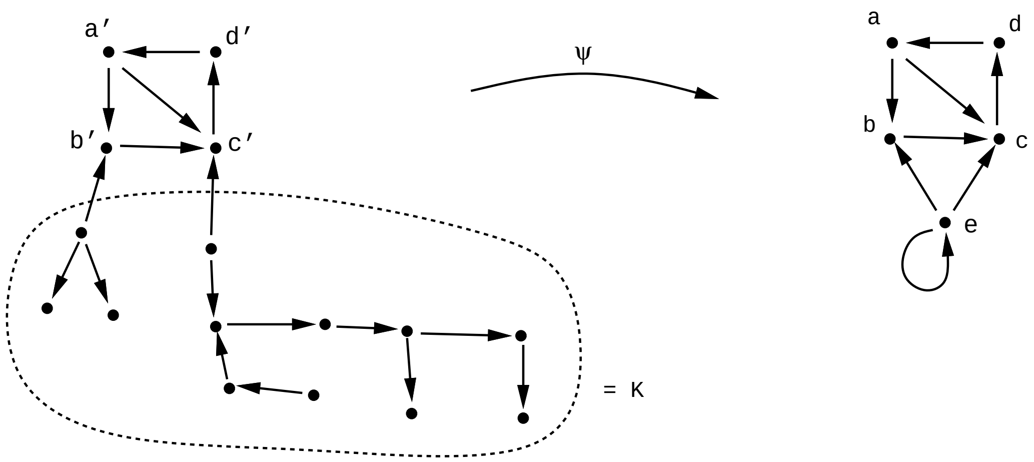
row and column must appear in order to distinguish a simple pattern from a template. An abbreviated version of this simple pattern is

```
      y   z   *
    ------------
x | 1   0   *
y | *   1   *
* | *   *   *
```

Example 4 The simple pattern

```
      w   x   y   z   *
    --------------------
w | *   1   1   *   *
x | *   *   1   *   *
y | *   *   *   1   *
z | 1   *   *   *   *
* | *   *   *   *   *
```

matches graphs having four-element loops with a diagonal edge. It matches the graph in Example 1 in exactly one way, using the simple homomorphism ψ shown here. Notice that none of the nodes a, b, c or d in $\psi(G)$ may be the supernode.



Bindings

For any successful simple pattern match (G, P, β, ψ), β and ψ determine a unique assignment of graph nodes to each non-$*$ simple pattern label. This assignment may be thought of as a binding of values to names. Accordingly, we shall use the term "variable" (or "pattern variable") to refer to non-$*$ pattern labels in what follows, with the understanding that the *value* bound to a variable v in simple pattern P after the successful simple match (G, P, β, ψ) is that graph node a such that $\psi(a) = \beta(v)$. This binding information is important, as it is necessary in order to combine patterns with functions to produce transformations. This point is discussed further in Section 2.2. Variable bindings will also be used later to transmit information from one pattern match to subsequent ones. Chapter 4 discusses this issue, and presents some mechanisms for controlling the lifetimes of bindings. For the purposes of this chapter, we will assume that variables are globally scoped, and that bindings, once established, are permanent.

Necessarily-connected components

We now define a structural feature of simple patterns that is echoed in all graphs that a simple pattern matches. It is useful for discussing several aspects of pattern matching.

Definition 8 A *necessarily connected component (NCC)* of a simple pattern P is a maximal set of pattern variables having the property that any set of graph nodes to which they may become bound in a successful simple pattern match (G, P, β, ψ) must necessarily lie in a single component of G.

Notice that nodes bound to variables in two different NCC's are not constrained to lie in different components of a graph, just that nodes bound to variables within the same NCC **must** lie in the same graph component.

Each pattern variable is a member of some NCC. The NCC containing variable p in simple pattern P may be computed by constructing an adjacency relation R defined as follows: let x and y be variables of P and let $x\,R\,y$ if and only if $P_{x,y} = 1$ or $P_{y,x} = 1$. Variable q is then in the same NCC as variable p if and only if pR^*q, where R^* is the reflexive transitive closure of R. Variables p and q are thus seen to be in the same NCC if they are in the same component of the graph formed by considering R as an adjacency matrix. Since R is determined solely by the 1's in the pattern matrix, this is exactly the matrix formed by replacing each of P's matrix's $*$ entries with a 0.

Figure 2.1 shows a simple pattern containing two NCC's. The first contains the variables v and w, while the second is made up of x, y, and z.

```
        v   w   x   y   z   *
      ------------------------
  v |   1   0   *   *   *   *
  w |   1   *   *   *   *   *
  x |   *   *   *   *   *   *
  y |   *   *   1   0   1   *
  z |   *   *   0   1   *   *
  * |   *   *   *   *   *   *
```

Figure 2.1: A simple pattern with two NCC's

Perhaps the most important feature of NCC's is that for the most part they may be considered separately by a simple pattern matching algorithm. Except for the requirement that the bindings of two NCC's may not intersect (since β is a bijection), a simple pattern with two or more NCC's may be considered to be two or more simple patterns, each containing only one NCC.

2.1.3 General matching

Simple pattern matching provides most of the power we need for performing flow analysis, but there is a generalization that gives us additional power at little extra cost. It can be used to streamline many flow analysis algorithms, both in terms of coding complexity and in terms of execution efficiency.

Definition 9 A *general graph homomorphism* is a mapping ψ from a graph G onto a graph H such that for some partition $K = \{K_1, \ldots, K_n\}$ of the nodes of G,

(i) For every pair of nodes $a \in K_i$ and $b \in K_j$, $\psi(a) = \psi(b)$ if and only if $i = j$, and

(ii) For each edge $a \to b$ in G, the edge $\psi(a \to b) = \psi(a) \to \psi(b)$.

The nonempty elements of K are thus in one to one correspondence with the nodes of H.

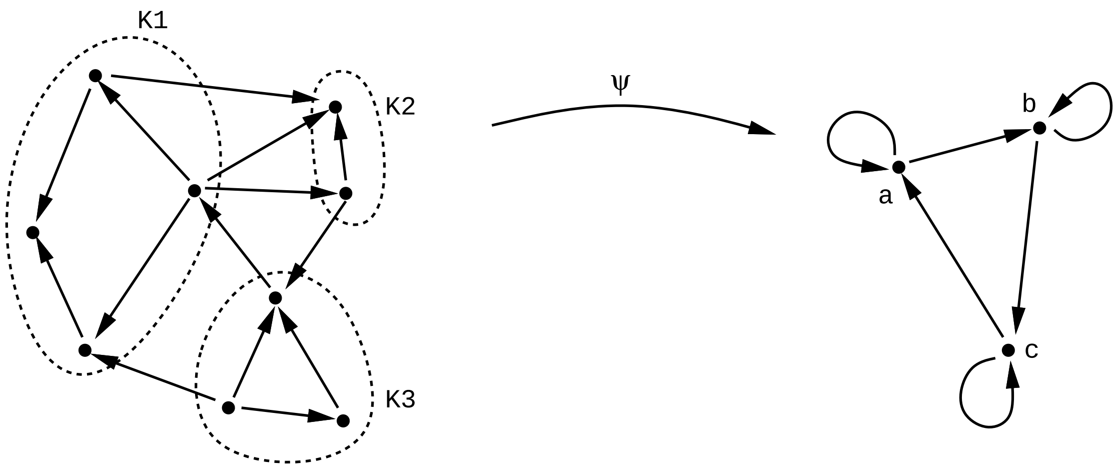
Where a simple homomorphism creates a single supernode, a general homomorphism creates several. We will use general homomorphisms to define the concept of *general pattern match* as an extension of simple pattern match.

Definition 10 A *weak general match* is a quadruple (G, T, β, ψ) where ψ is a general homomorphism such that $(\psi(G), T, \beta)$ is a successful template match.

Example 5 The template

```
      A   B   C
    ------------
A |   *   1   0
B |   0   *   1
C |   1   0   *
```

matches the graph G below via the simple homomorphism ψ shown, with $\beta(A) = a$, $\beta(B) = b$ and $\beta(C) = c$. The node a is the supernode formed by collapsing the set of nodes in $K1$. Similarly, b is formed from $K2$ and c is formed from $K3$. As usual, the homomorphism maps edges contained within $K1$, $K2$ and $K3$ to looping edges in the image.



Before defining general matching, we need a more generalized definition of pattern.

Definition 11 A *pattern* is a template whose labels are classified into three sets: *singleton variables*, *set-valued variables*, and $\{*\}$. A pattern must have exactly one $*$ label.

We denote singleton variables by lower-case names, set-valued variables by upper-case names, and, of course, $*$ by $*$. The usual pattern abbreviation convention applies to patterns as it did to simple patterns.

Definition 12 A *weak pattern match* is a weak general match (G, P, β, ψ) where, for each singleton variable v in P there is exactly one node a in G with $\beta(v) = \psi(a)$, and where, for each set-valued variable V in P, $\{a \mid \beta(V) = \psi(a)\} \neq \emptyset$. Further, no restrictions are placed on the size of the set $\{a \mid \beta(*) = \psi(a)\}$.

Variable binding is defined for weak pattern matching in an analogous fashion to the way it was defined for simple matching: the set of graph nodes that is bound to a variable χ after a weak pattern match is the set of nodes a such that $\psi(a) = \beta(\chi)$, whether χ is a singleton or a set-variable. While singleton variables are necessarily bound to exactly one graph node, set-variables may be bound to more than one. We will henceforth treat $*$ as a set-variable, except that as per the last sentence in the above definition, it may be bound to the empty set of graph nodes after a pattern match.

The definition of NCC is independent of that of binding, and so is extended to these more general patterns in the obvious way. The algorithm for their computation remains unchanged.

Definition 11 defines patterns more generally than simple patterns, removing the restriction that entries in the $*$ row and column be $*$'s. This is now possible because the $*$ label may be regarded as a set-valued variable (though a special one). But while we relax one restriction on the patterns we may write, we also impose one on the choices of ψ in order to be able to make use of both general and simple matching with the same pattern. By viewing this as a restriction on the functions ψ that can satisfy our matching criterion, we obtain extra utility without changing the semantics of our pattern matching process.

As might be surmised from its name, the weak pattern match is not sufficient for our needs. Of the many possible extensions of simple pattern matching that use general homomorphisms, we want one that fulfills two especially desirable criteria.

Requirement 1 If (G, P, β, ψ) is a successful simple pattern match then it is a successful general pattern match.

This requirement will ensure that the collection of general pattern matches is a true extension of the collection of simple pattern matches in the mathematical sense. This guarantees that general patterns and simple patterns will act consistently, in that if a match by general pattern P results in the binding of each non-$*$ variable to a single graph node, then the simple pattern Q obtained by replacing each set-valued variable in P with its lower-case equivalent will match the same graph with corresponding bindings.

Before presenting the second requirement, we prove a property of simple pattern matches that we want to extend to general pattern matches. Define the *complement* of a graph G to be the graph $\bar{G}$ containing the same nodes as G and containing edge $a \rightarrow b$ if and only if G does not. Similarly, we define the template $\bar{T}$ to be identical to the template T, except that the matrix of $\bar{T}$ has 1 entries wherever T has 0 entries and has 0 entries wherever T has 1 entries. Both $\bar{T}$ and T have the same $*$ entries.

Proposition 1 (G, P, β, ψ) is a successful simple pattern match if and only if $(\bar{G}, \bar{P}, \beta, \psi)$ is a successful simple pattern match.

Proof: Suppose that (G, P, β, ψ) is not a successful simple pattern match. This means there exist pattern variables x and y in P and nodes a and b in G with $\psi(a) = \beta(x)$ and $\psi(b) = \beta(y)$ such that either

$$P_{x,y} = 1 \quad \text{and} \quad [G]_{a,b} = 0$$

$$\text{or}$$

$$P_{x,y} = 0 \quad \text{and} \quad [G]_{a,b} = 1.$$

But then, by definition of complementary patterns and graphs, either

$$\bar{P}_{x,y} = 0 \quad \text{and} \quad [\bar{G}]_{a,b} = 1$$

$$\text{or}$$

$$\bar{P}_{x,y} = 1 \quad \text{and} \quad [\bar{G}]_{a,b} = 0,$$

showing that $(\bar{G}, \bar{P}, \beta, \psi)$ is not a simple pattern match.

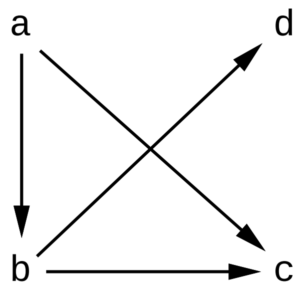
Conversely, suppose $(\bar{G}, \bar{P}, \beta, \psi)$ is not a simple pattern match. Then, by the same argument, $(\bar{\bar{G}}, \bar{\bar{P}}, \beta, \psi) = (G, P, \beta, \psi)$ is also not a simple pattern match. ∎

Our second requirement demands that general matches also have this property. While it is not terribly important for simple matches, we will see in Chapter 3 that it is crucial in order to ensure that patterns interact properly with the graph functions defined in Section 2.2.

Requirement 2 (Duality) (G, P, β, ψ) is a successful pattern match if and only if $(\bar{G}, \bar{P}, \beta, \psi)$ is a successful pattern match.

Weak pattern matches fail to satisfy the duality requirement. Let G be the graph and P the pattern shown in Figure 2.2. The $*$ row and column are filled with 0's, so every graph node must be bound to one of X, Y, or Z. Since no edges enter a and none leave c or d, then in any weak pattern match of G by P, X must bind to $\{a\}$ and Z must bind to $\{c, d\}$. It then follows that Y must bind to $\{b\}$. This information determines β and ψ.

	X	Y	Z	*
X	0	1	1	0
Y	0	0	1	0
Z	0	0	0	0
*	0	0	0	0



	X	Y	Z	*
X	1	0	0	1
Y	1	1	0	1
Z	1	1	1	1
*	1	1	1	1

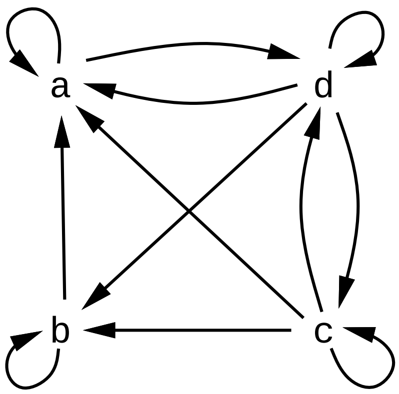


Figure 2.2: A pattern, graph, and their duals

Now the complement of G contains an edge from a to d. The complement of the pattern, however, contains a 0 in $\bar{P}_{X,Z}$, ensuring that $(\bar{G}, \bar{P}, \beta, \psi)$, which necessarily has the same bindings as above, will not be a weak pattern match.

Why does the duality condition hold for successful simple pattern matches but not for weak pattern matches? The problem is with 1's in the template. Where, in a weak pattern match, a 0 in position $P_{x,y}$ means that **no** edges $a \rightarrow b$ exist in G with $\psi(a) = \beta(x)$ and $\psi(b) = \beta(y)$, a 1 means that **some** such edges exist. The proper dual condition to the "0" situation would instead be that **all** such edges exist. The remainder of this section will develop a pattern match that enforces this stronger condition.

The homomorphism ψ collapses several edges into one, but provides no information as to how many it collapsed. In the case of successful simple pattern matches, we know that the number of graph edges in the inverse image of a single edge is exactly one. In the more general case, we may know the number of nodes that collapse to form each supernode, but we can have no idea of the number of edges that collapse to form a "superedge". We can know only that there is at least one. This indeterminacy means that there are a great many graphs differing in the portion being tested by the pattern that we cannot distinguish from one another. A more mathematical way of saying this is that the homomorphism ψ does not have a well-defined inverse, and so we do not have a function that can determine relevant features of G from features of $\psi(G)$. In particular, we cannot determine how $\bar{P}$ will relate to $\bar{G}$.

Let (G, P, β, ψ) be a weak pattern match, and let H be the graph $\psi(G)$. We define an auxiliary graph-valued function ψ_G^* termed the *G-dilation* of ψ as follows: let f and g be nodes in H, and let J be the subgraph of H containing just the nodes f and g and the edge $f \rightarrow g$. Then $\psi_G^*(J)$ is the graph whose set of edges is $\{a \rightarrow b \mid \psi(a) = f \text{ and } \psi(b) = g\}$ and whose nodes are all endpoints of such edges. We extend the domain of ψ_G^* to all of H by defining $\psi_G^*(J \cup K) \stackrel{def}{=} \psi_G^*(J) \cup \psi_G^*(K)$ where the union of two graphs is that graph formed by the union of their edge sets. ψ_G^* is not defined on individual nodes and edges as is ψ, but is instead defined as a map from graphs to graphs. This function ψ_G^* will serve as an inverse to ψ in the category of graphs (as opposed to that of nodes and edges), allowing us to define a suitable general pattern match.

Now let (G, P, β, ψ) be a weak pattern match. We define P_1^G to be the subgraph of $\psi(G)$ that contains the edge $a \rightarrow b$ if and only if $P_{\beta^{-1}(b),\beta^{-1}(a)} = 1$, or in other words, contains only those edges matched by 1's in the pattern. P_0^G is similarly defined to be the graph containing the edge $a \rightarrow b$ if and only if $P_{\beta^{-1}(a),\beta^{-1}(b)} = 0$. P_0^G is not a subgraph of $\psi(G)$, but its nodes are a subset of $\psi(G)$'s node set. P_0^G may alternatively be defined as $\bar{P}_1^{\bar{G}}$.

Definition 13 A *successful pattern match* (or simply *match*) is a weak pattern match (G, P, β, ψ) where $\psi_G^*(P_1^G) \subseteq G$. A *pattern matching algorithm* is an algorithm that, given a graph G and pattern P as input, finds a suitable β and ψ (if such exist) so that (G, P, β, ψ) is a successful pattern match.

Proposition 2 If (G, P, β, ψ) is a successful simple pattern match then it is a successful pattern match.

Proof: Since every simple homomorphism is a general homomorphism, every successful simple pattern match is a weak pattern match. To show that every successful simple pattern match (G, P, β, ψ) is also a successful pattern match, we need to exhibit $\psi^*_G(P^G_1)$ and show it to be a subgraph of G.

Let $p \to q$ be an edge in P^G_1. Since (G, P, β, ψ) is a successful simple pattern match, there is some 1 entry in some position $P_{x,y}$ in the pattern where neither x nor y is $*$, but where $\beta(y) = q$ and $\beta(x) = p$. Thus, there are unique nodes a and b in G such that the edge $a \to b$ is in G, and such that $p = \psi(a)$ and $q = \psi(b)$. This holds true for each edge in P^G_1, and so $\psi^*_G(P^G_1)$ is the union of these edges, each of which is in G. ∎

Proposition 3 (G, P, β, ψ) is a successful pattern match if and only if $(\bar{G}, \bar{P}, \beta, \psi)$ is a successful pattern match.

Proof: Let P be a pattern and G a graph. Suppose that (G, P, β, ψ) is a successful pattern match. Then for all pattern variables x and y in P bound to node sets A and B in G (via ψ and β), and for all $a \in A$ and $b \in B$, either $P_{x,y} = 1$ and $[G]_{a,b} = 1$, since $\psi^*_G(P^G_1) \subseteq G$, or $P_{x,y} = 0$ and $[G]_{a,b} = 0$ or $P_{x,y} = *$. The reason that $\psi^*_G(P^G_1) \subseteq G$ forces $P_{x,y} = 1$ to imply that $[G]_{a,b} = 1$ is as follows: $P_{x,y} = 1$, so the edge $\beta(x) \to \beta(y) \in P^G_1$. From the bindings, we know that $\psi(a) = \beta(x)$ and $\psi(b) = \beta(y)$. Consequently, the edge $a \to b$ lies in $\psi^*_G(P^G_1)$, and so in G.

Now, by definition of complementary graph, $\psi^*_G(P^G_1) \cap \bar{G} = \emptyset$. Thus, for all pattern variables x and y in P bound to node sets A and B in G (via ψ and β), and for all $a \in A$ and $b \in B$, either $\bar{P}_{x,y} = 0$ and $[\bar{G}]_{a,b} = 0$ or $\bar{P}_{x,y} = 1$ and $[\bar{G}]_{a,b} = 1$ or $\bar{P}_{x,y} = *$, showing that $(\bar{G}, \bar{P}, \beta, \psi)$ is a successful pattern match.

Now suppose $(\bar{G}, \bar{P}, \beta, \psi)$ is a successful pattern match. Then, by the same argument, $(\bar{\bar{G}}, \bar{\bar{P}}, \beta, \psi) = (G, P, \beta, \psi)$ is also a successful pattern match. ∎

Successful pattern matches rigorously define the semantics of 1's and 0's in rows and columns whose labels can bind to sets of nodes instead of just to singleton nodes. Let (G, P, β, ψ) be a successful pattern match. If the entry in $P_{X,Y}$ is a 1, then the edge $\beta(X) \to \beta(Y)$ is in $\psi(G)$. Considering this edge and its endpoints to be a graph H, all edges in $\psi^*_G(H)$ are necessarily in G. If, on the other hand, $P_{X,Y}$ is a 0, then none of these edges are in G.

Figure 2.3 illustrates a pattern that makes use of the new definition by having (useful) non-$*$ entries in a $*$ column. This pattern matches a graph if there exists an edge $a \rightarrow b$ in the graph where $\beta(y) = \psi(b)$ and no other edges enter b.

```
      y   *
     --------
  x |  1   *
  y |  0   *
  * |  0   *
```

Figure 2.3: Example use of the $*$ row

2.2 Graph Transformations

Definition 14 A *primitive graph function* is a function $f_{A,B}$ specified by two graph constants $A = (N_A, E_A)$ and $B = (N_B, E_B)$ whose edge sets E_A and E_B are disjoint. If $G = (N, E)$ is a graph, then $f_{A,B}(G)$ is the graph whose edge set is $E \cup E_A - E_B$. Its nodes are the endpoints of its edges.

An example primitive graph function is depicted in Figure 2.4. It is clear that this family of functions contains, for any graphs G and H, a function that maps G to H. This is not to be construed as meaning that the family contains all graph functions, since given arbitrary graphs G, H, J and K, there are primitive graph functions mapping G to H and others mapping J to K, but there may be no single primitive graph function that does both. The reason for this is that the edges being added and removed are not a function of the graph being operated on. Nonetheless, this family of functions is large enough to suit our purposes.

We now define a related set of functions that are more useful in what follows:

Definition 15 A *primitive h-function* is a function $f_{A,B,h}$ specified by two graph constants $A = (N_A, E_A)$ and $B = (N_B, E_B)$ whose edge sets E_A and E_B are disjoint, and an arbitrary function h from graphs to graphs. If $G = (N, E)$ is a graph, then

$$f_{A,B,h}(G) \stackrel{\text{def}}{=} f_{h(A),h(B)}(G).$$

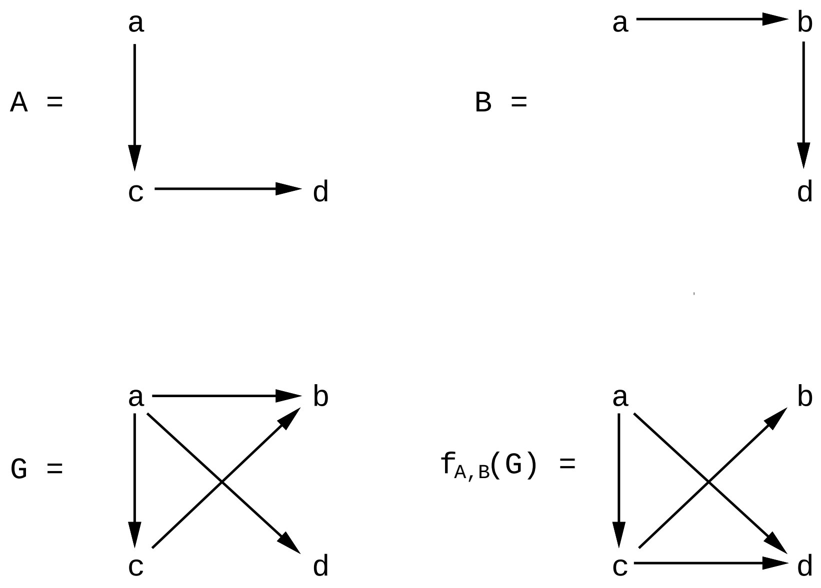


Figure 2.4: A primitive graph function

The set of primitive h-functions is actually the same as the set of primitive graph functions. As specified by its definition, any primitive h-function $f_{A,B,h}$ may be written as the primitive graph function $f_{h(A),h(B)}$. Further, if I is the identity function on graphs, then any primitive graph function $f_{A,B}$ may be written as the primitive h-function $f_{A,B,I}$. The former definition gives a more easily understood definition of the set of functions being considered, while the second definition allows us to write the functions in a more useful way.

We now make use of the second definition, coupling patterns and primitive h-functions to produce a class of functions called *S-maps*. Let Γ be the set of all graphs.

Definition 16 An *S-map* $S_{P,A,B}$ is a function from Γ to 2^Γ specified by a pattern P and two disjoint sets A and B of pairs of variables of P. It is defined as

$$S_{P,A,B}(G) \stackrel{\text{def}}{=} \{f_{\beta(A),\beta(B),\psi^*_G}(G) \mid (G,P,\beta,\psi) \text{ is a successful pattern match } \}$$

where $\beta(X)$ is here defined as the graph consisting of all edges $\beta(x) \to \beta(y)$ such that the pair (x, y) is in the set X of pairs of variables.

Informally, the image of a graph under an S-map is the set of all graphs that could result from applying some primitive h-function to G with h being the ψ^*_G derived from a pattern match of G. This need for an explicit ψ^*_G is the primary reason that weak general matches are insufficient for our purposes. We will see that S-maps specialize to the kinds of functions we want. It is clear, however, that their definition makes no sense without a graph-functional inverse to the homomorphism that arises from a pattern match.

Were the images of S-maps single graphs rather than sets of graphs, we would have the transformations we seek. But they are not. Our tack then is to turn S-maps into transformations by constructing a function that unambiguously chooses a single representative from each S-map image of a graph. This "disambiguating function" D, when applied to some S-map $S_{P,A,B}$, produces the well-defined graph transformation $DS_{P,A,B}$. In other words, $S_{P,A,B}$ maps $\Gamma \rightarrow 2^{\Gamma}$, but $DS_{P,A,B}$ maps $\Gamma \rightarrow \Gamma$. The following chapter develops a class of disambiguating functions in detail.

Chapter 3

Disambiguation

Chapter 2 introduced S-maps and motivated the idea of disambiguators. This chapter describes a parametrizable disambiguator that may be used to convert S-maps to well-defined graph transformations. The word "parameterizable" here stems from the fact that this disambiguator can be configured dynamically to emulate a large class of useful disambiguators.

3.1 Transformations

Definition 17 *Let* Γ *be the set of all graphs,* $[\Gamma \to \Gamma]$ *be the set of all functions from* Γ *to* Γ*, and let* $[\Gamma \to 2^\Gamma]$ *be the set of all functions from* Γ *to* 2^Γ*. A* disambiguator D *is a function from* $[\Gamma \to 2^\Gamma]$ *to* $[\Gamma \to \Gamma]$ *such that for all* $f \in [\Gamma \to 2^\Gamma]$ *and all* G *in* Γ*,* $Df(G) \in f(G)$*.*

The functions f to which we will be applying disambiguators are, of course, the S-maps of Chapter 2. We may think of an S-map as being a non-deterministic function on graphs by considering its image to be a set of possible image graphs. The use of a disambiguator allows us to convert such a non-deterministic function into a deterministic one by forcing the choice of a single representative from each S-map image set.

We can now define the graph transformations that we later use for flow analysis:

Definition 18 A *graph transformation* (or more simply, *transformation*) $T_{P,A,B,D}$ is a function from graphs to graphs defined by

$$T_{P,A,B,D}(G) \stackrel{\text{def}}{=} (DS_{P,A,B})(G)$$

for some disambiguating function D. Should the pattern P not match G, causing $S_{P,A,B}(G)$ to be the empty set, we define $T_{P,A,B,D}(G)$ to be the identity function.

Assuming a particular disambiguator D, we write $T_{P,A,B,D}$ using a tabular notation consisting of a pattern followed by a *replacement* whose format is strikingly similar to that of the pattern. A replacement consists of a matrix of the same size as the pattern matrix with row and column labels identical to those adorning the pattern matrix. The only difference between a transformation's pattern and replacement is in the replacement's matrix entries. We denote the entry in the position of the replacement matrix whose row and column are labeled by the variables x and y, respectively, as $R_{x,y}$. Every replacement matrix entry is 1, 0, or "=". A 1 in position $R_{x,y}$ means that the pair of variables (x, y) is in the set A and describes edges to be added to G, while a 0 means that (x, y) is in B and describes edges to be removed from G. An = entry is understood to be a placeholder, and the variable pair it identifies is neither in A nor in B. Hence, it may be thought of as describing edges that are to be left alone. Since the sets A and B are disjoint, this notation can express any graph transformation. Accordingly, we allow ourselves to write $T_{P,A,B,D}$ as $T_{P,R,D}$ when the replacement R requires explicit mention.

Example 6 An important transformation taken from data-flow analysis is the T_2' transformation due to Graham and Wegman [13, 24]. This transformation is used in their efficient but complex information propagation algorithm. Figure 3.1 gives a rough pictorial idea of what this transformation does.

Informally, the pattern portion of the transformation looks for a structure in the graph like that shown on the left, wherein node a points to node b, and b points to several other nodes, among them node c. The dotted arrow indicates that no other edges may be incident to node b if the pattern is to match. Should these conditions be fulfilled, the edge $b \rightarrow c$ is deleted and the edge $a \rightarrow c$ inserted. A transformation that accomplishes this task is shown in Figure 3.2.

Figure 3.2 may be read as follows: the left-hand matrix is a pattern that matches the situation described above: the matcher searches for graph nodes a, b, c and d to bind to the pattern variables x, y, z and w, respectively, so that

1. There is an edge from a to b (the 1 in row x),

2. No other edges enter b (the 0 entries in column y),

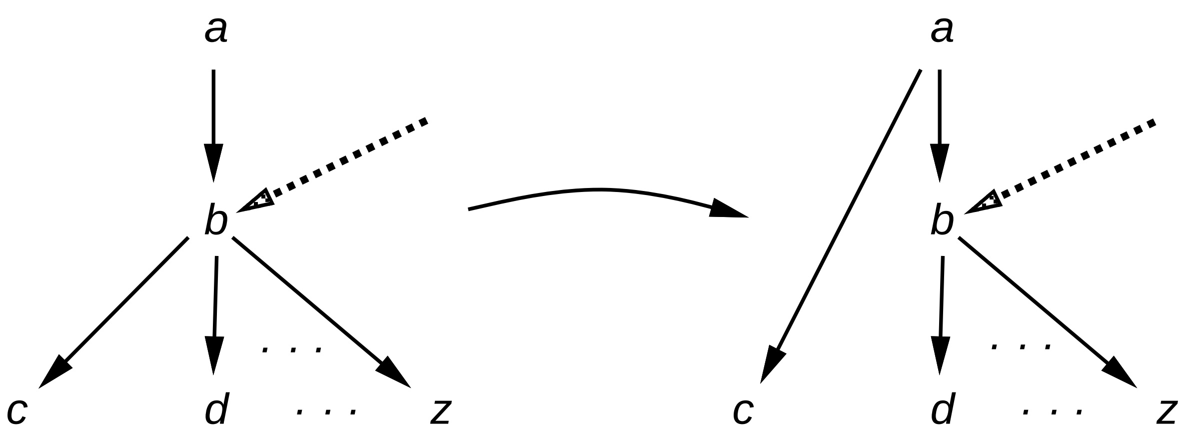


Figure 3.1: The T_2' transformation of Graham and Wegman

	x	y	z	w	*
x	*	1	*	*	*
y	*	0	1	1	*
z	*	0	*	*	*
w	*	0	*	*	*
*	*	0	*	*	*

	x	y	z	w	*
x	=	=	1	=	=
y	=	=	0	=	=
z	=	=	=	=	=
w	=	=	=	=	=
*	=	=	=	=	=

Figure 3.2: A T_2' transformation

3. b has at least two edges leaving it (the 1's in row y).

Should the pattern match succeed, finding such a, b, c and d, the h-function indicated by the replacement matrix is performed on the graph. The edge $a \rightarrow c$ is added to the graph (the 1 entry denotes that $(x, z) \in A$) and the edge $b \rightarrow c$ is deleted (the 0 entry denotes that $(y, z) \in B$). The function ψ_G^*, in conjunction with the binding function β, is used to identify the variables x, y, z and w with the nodes a, b, c and d in order to determine the edges to be added or deleted.

In this example, if the variables x, y and z were set-valued variables (and hence written X, Y and Z) instead of singleton variables, then the 1 in the replacement would mean that for all graph nodes a bound to X and for all graph nodes c bound to Z, an edge

$a \rightarrow c$ would be added to the graph. Similarly, for all graph nodes a and b bound to X and Y, respectively, the edge $a \rightarrow b$ would be removed from G were it present.

It should be evident that the following requirement is satisfied:

Requirement 3 The transformation $T_{P,R,D}$, where the replacement R is constructed by replacing P's $*$ matrix entries with $=$ entries, leaves a graph unchanged for every pattern P.

This requirement says that if a pattern matches some edge (or the lack thereof), then adding it (respectively, deleting it) has no effect. This may seem an innocuous requirement, but it is equivalent to the duality requirement introduced in the previous chapter. This equivalence is most easily seen by noticing the fact that a 1 in position $R_{x,y}$ of this replacement means to add all edges of $\psi^*_G(H)$ to G, where H is the graph containing the two points $\beta(x)$ and $\beta(y)$ and the edge $\beta(x) \rightarrow \beta(y)$. For this action to leave the graph unchanged, the 1 in position $P_{x,y}$ of the pattern would have to match only this same set of edges. But this is exactly what the duality condition of Chapter 2 says.

3.2 A Generic Disambiguator

Example 6 of the previous section was informal in the sense that the function $T_{P,A,B,D}$ was not exhibited explicitly. In fact, since no disambiguator D was given, the example did not actually describe a function. The problem, as was mentioned at the end of Chapter 2, is that there may be more than one set of nodes a, b, c and d satisfying the conditions stated, and so it is not clear where in the graph the modification is to take place. An S-map models this by having as its image the set of all possible results. The way we choose to disambiguate this indeterminacy, and thus make the transformation a well-defined function, is to specify a particular pattern matching algorithm to be used to find the functions β and ψ (and thereby ψ^*_G) in a deterministic fashion. This section describes a particular disambiguator by presenting such an algorithm. The following section modifies this algorithm so that it can emulate many other useful disambiguators. While it is not necessarily the case that the algorithm presented here is the best or most efficient for performing this task, Chapter 6 shows it to be sufficient for performing the flow analysis algorithms found in the literature with no degradation of asymptotic time bounds.

Rather than find the functions β and ψ, our pattern matcher will attempt to find a function $\beta^{-1} \circ \psi$. If (G, P, β, ψ) is a successful pattern match, this function maps graph nodes to pattern variables. For v a pattern variable in P, the inverse image of $\beta^{-1} \circ \psi$ at v is the set of graph nodes bound to v.

The reason for finding this function rather than its constituent components is that β and ψ fail to be unique in a rather trivial way: let $H \neq G$ be a graph isomorphic to G via a map σ, and let α be a bijection mapping the variables of P to the nodes of H given by $\alpha = \sigma \circ \beta$. Then note that $\beta^{-1} \circ \psi = (\alpha^{-1} \circ \alpha) \circ \beta^{-1} \circ \psi = \alpha^{-1} \circ ((\alpha \circ \beta^{-1}) \circ \psi) = \alpha^{-1} \circ (\sigma \circ \psi) = \alpha'^{-1} \circ \psi'$ for $\psi' = (\sigma \circ \psi)$. Thus, for a given set of variable bindings, there are many choices of β and ψ that describe it. Finding a mapping describing the variable bindings instead of finding β and ψ allows us to circumvent this problem.

It is more convenient for the purpose of performing a transformation to have available the function which maps each pattern variable to its inverse image under $\beta^{-1} \circ \psi$. This function gives a straightforward mapping from the specification of a replacement to the sets of edges that must be added to or removed from the graph being transformed, thus easing the task of the replacer. Thus, we will compute, for each pattern variable, the set of graph nodes that map to it so that the set of constraints imposed by the pattern (the existence or non-existence of certain edges, and the fact that singleton variables must have exactly one pre-image node) are satisfied. Computing these sets is equivalent to finding a partition of the graph's nodes into the "buckets" provided by the pattern variables subject to those constraints. An algorithm for deterministically solving this partitioning problem will thus determine, for each graph G and pattern P that matches it, a unique set of variable bindings. Since each member of $S_{P,A,B}(G)$ results from a different binding function, this will cause $|S_{P,A,B}(G)| = 1$. The single graph represented there will then be the image of our transformation $DS_{P,A,B}(G)$.

The problem of pattern matching as it has been outlined here includes the *subgraph isomorphism problem* which is well-known to be NP-complete [12]. A search for any subgraph H in the graph G can be coded by writing a simple pattern whose matrix, ignoring the $*$ row and column, is the adjacency matrix for H with 0 entries replaced with $*$'s. If the resulting pattern matches part of G, then the edges corresponding to H have been identified: H is easily seen to be a subgraph of a simple homomorphic image of G, and in fact a subgraph of that portion of the simple homomorphic image that does not include the kernel. It is thus exposed as isomorphic to some subgraph of G.

It follows that our disambiguator must necessarily include a general (that is, worst-case exponential) search algorithm.[1] Nevertheless, it is possible to do somewhat better in the average case than testing each partition of the node variables until finding one that satisfies the specified constraints. Matters may be significantly improved by using structural information found in the pattern to narrow the scope of the search. This is the approach taken here.

The exposition that follows assumes for simplicity that the patterns with which the algorithm deals have exactly one necessarily-connected component. Since necessarily-connected components are, for the most part, processed independently (subject to the constraint that the regions in the graph that they match may not overlap), this is little loss of generality. Where interactions among NCC's become significant, they are so noted.

3.2.1 A simple pattern matching algorithm

Matching a simple pattern P against a graph G is done with a recursive procedure *bind* that searches for a complete set of bindings for the pattern variables satisfying the constraints given in the pattern. We construct initial data structures for the algorithm as follows: let N be the NCC that we wish to match. Then,

1. Choose a variable $v \in N$. We call v the *seed variable* for this NCC.

2. Order the nodes in G according to some well-order $\prec$.

3. Construct a set $\bar{v}$ containing every node in G. This set holds the possible values of v. Let $\dot{v}$ be the smallest element of $\bar{v}$ with respect to the ordering $\prec$.

4. Construct a set B and enter into it the single triple $(v, \bar{v}, \dot{v})$. This triple is called a *possible binding vector (PBV)*. It holds all possible legal values for the variable v in its second component. The third component indicates a particular element in the set $\bar{v}$ to which v is considered to be provisionally bound while this triple is in B. Whenever a variable appears in a PBV in B, we say that the variable is *represented* in B.

By construction, we will ensure that any set of provisional bindings expressed by the PBV's in B always satisfy all of the constraints expressed by the pattern P.

[1] Or provide a proof that P=NP.

	x	y	z	*
x	*	1	*	*
y	*	*	1	*
z	1	*	*	*
*	*	*	*	*

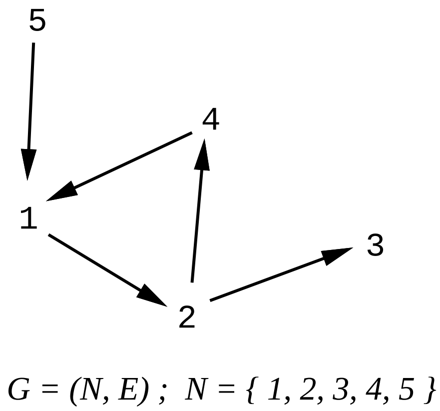


Figure 3.3: Matching a cycle

Example 7 Figure 3.3 shows a small graph containing a cycle and a pattern which attempts to match such cycles. The above four steps performed on this input are:

1. Choose our seed variable to be x.

2. Order the graph nodes. We will let them be ordered by their node numbers.

3. The set $\bar{x} = \{1, 2, 3, 4, 5\}$ and $\dot{x} = 1$.

4. $B = \{\ (x, \{1, 2, 3, 4, 5\}, 1)\ \}$.

We define variables x and y to be *adjacent* if and only if $P_{x,y} = 1$ or $P_{y,x} = 1$. Recall that N is determined by the reflexive transitive closure of the adjacency relation. The remainder of the algorithm is an application of the recursive algorithm *bind*, which reads as follows:

1. Choose a pattern variable x not yet represented in B that is adjacent in N to at least one variable represented in B. If more than one such x exists, choose whichever is alphabetically least. If no such x exists, then the algorithm has processed all of N and the provisional bindings in B represent a successful match. Return *success*.

2. The variable x is, by its choice, adjacent to at least one variable in N. Let $\overrightarrow{x}$ $\overset{\not\leftrightarrow}{x}$ $\overleftarrow{x}$ and $\overset{\not\leftrightarrow}{x}$ be sets of variables in N defined by

$$\overrightarrow{x} = \{y \text{ represented in } B \mid P_{x,y} = 1\}$$

$$
\begin{aligned}
\overset{\not\rightarrow}{x} &= \{y \text{ represented in } B \mid P_{x,y} = 0\} \\
\overleftarrow{x} &= \{y \text{ represented in } B \mid P_{y,x} = 1\} \\
\overset{\not\leftarrow}{x} &= \{y \text{ represented in } B \mid P_{y,x} = 0\}.
\end{aligned}
$$

The constraints on the values that x may assume in order to be consistent with the bindings in B are all encoded by the variables in $\overrightarrow{x}$ $\overset{\not\rightarrow}{x}$ $\overleftarrow{x}$ and $\overset{\not\leftarrow}{x}$ and their provisional bindings. Let the provisional binding function, which associates a value with a variable, be called ν, and let the set of legal values for x be called $\bar{x}$. The value of $\bar{x}$ is then given by:

$$
\begin{aligned}
\bar{x} &= \{a \mid \ a \text{ is not } \dot{x} \text{ for any variable } x \text{ represented in } B\ \} \\
&\cap \left(\bigcap_{\forall y \in \overrightarrow{x}} \{a \mid a \rightarrow \nu(y) \text{ is an edge in } G\ \} \right) \\
&\cap \left(\bigcap_{\forall y \in \overset{\not\rightarrow}{x}} \{a \mid a \rightarrow \nu(y) \text{ is not an edge in } G\ \} \right) \\
&\cap \left(\bigcap_{\forall y \in \overleftarrow{x}} \{a \mid \nu(y) \rightarrow a \text{ is an edge in } G\ \} \right) \\
&\cap \left(\bigcap_{\forall y \in \overset{\not\leftarrow}{x}} \{a \mid \nu(y) \rightarrow a \text{ is not an edge in } G\ \} \right).
\end{aligned}
$$

If $\bar{x}$ is empty, then this invocation of *bind* returns *failure*; otherwise it adds the PBV $T = (x, \bar{x}, \dot{x})$ to B where $\dot{x}$ is the smallest node in $\bar{x}$ under the ordering $\prec$.

3. Call *bind* recursively. If it succeeds, return *success*. Otherwise, remove $\dot{x}$ from $\bar{x}$. If $\bar{x}$ is non-empty, let $\dot{x}$ be the smallest node in $\bar{x}$ under the ordering $\prec$ and perform this step again. If, on the other hand, $\bar{x} = \emptyset$, then remove T from B and return *failure*.

Example 8 We will continue the previous example, stepping through *bind*.

1. Both variables y and z are adjacent to x; we choose the variable y for consideration.

2. Examining the pattern matrix, we see that

$$\overrightarrow{y} \ = \ \emptyset$$

$$\begin{aligned}
\overset{\not\rightarrow}{y} &= \emptyset \\
\overleftarrow{y} &= \{x\} \\
\overset{\not\leftarrow}{y} &= \emptyset.
\end{aligned}$$

Recalling that N is the set of all nodes in the graph, We obtain $\bar{y} = \{2,3,4,5\} \cap N \cap N \cap \{2\} \cap N = \{2\}$, since empty intersections are taken to be the universal set. This in turn gives $B = \{\ (x, \{1,2,3,4,5\}, 1),\ (y, \{2\}, 2)\ \}$.

3. We make a recursive call to *bind*:

   1 We choose variable z for consideration.

   2 $\overrightarrow{z} = \{x\}$, $\overset{\not\rightarrow}{z} = \emptyset$, $\overleftarrow{z} = \{y\}$, and $\overset{\not\leftarrow}{z} = \emptyset$. $\bar{z} = \{3,4,5\} \cap \{4,5\} \cap N \cap \{3,4\} \cap N = \{4\}$. $B = \{\ (x, \{1,2,3,4,5\}, 1),\ (y, \{2\}, 2),\ (z, \{4\}, 4)\ \}$.

   3 We make a recursive call to *bind*. It discovers there are no more variables to process and returns `success`. Accordingly, this invocation returns `success`.

   Since the recursive call succeeded, this invocation returns success.

If the algorithm terminates successfully, we obtain a *binding function* ν mapping graph nodes to pattern variables. $\nu(a)$ is defined to be the variable v such that $\dot{v} = a$ if such a v exists, otherwise $\nu(a)$ is defined to be $*$.

We can then choose an arbitrary bijection β from pattern variables to elements of some set of graph nodes and construct a simple homomorphism $\psi = \beta \circ \nu$. ψ is a simple homomorphism because ν is one-to-one on the non-$*$ pattern variables.

Example 9 Continuing the previous example, we see that *bind* has succeeded. The variable x is (provisionally) bound to 1, y to 2, and z to 4. Thus, $\nu(1) = x$, $\nu(2) = y$, $\nu(4) = z$, and $\nu(3) = \nu(5) = *$. Let β be the function $\{(x,10),(y,11),(z,12),(*,13)\}$. Then ψ is, by construction, $\{(1,10),(2,11),(3,13),(4,12),(5,13)\}$.

Theorem 1 The above algorithm constructs a β and ψ making (G, P, β, ψ) a simple match if such a match exists.

Proof: Suppose at least one match exists, and let μ be a function mapping (non-*) pattern variables to nodes that expresses this match (that is, $\mu = \nu^{-1}\ |_{\{\text{non-}*\text{ variables}\}}$ for that match). Let the seed variable be v_1, and enumerate the rest of the variables in the order

that they are considered by *bind*, calling them $v_2, \ldots, v_n$. Note that $\bar{v}_1$ contains every node in G, so v_1 will at some time take on the value $\mu(v_1)$ unless some other match is found first. Now suppose $v_1, \ldots, v_k$ are represented in B with provisional bindings $\mu(v_1), \ldots, \mu(v_k)$ for some $k < n$. Then $\mu(v_{k+1})$ is a legal value for v_{k+1}, and so is an element in $\bar{v}_{k+1}$. Accordingly, v_{k+1} will take on this value before any provisional binding of the variables $v_1, \ldots, v_k$ is changed, unless some other match is found first. Since this holds for all k, the provisional bindings $\mu(v_1), \ldots, \mu(v_n)$ will be found at some point unless another match is found first. If no other match is found, *bind* terminates with $\nu = \mu$, finding the match.
On the other hand, suppose that no such match exists. Then there is no set of values for $v_1, \ldots, v_n$ satisfying the constraints expressed by P. Since all of the provisional bindings in B always satisfy this set of constraints, we see that not all of the v_k can be represented in B. But then step 1 of *bind* can never terminate with a success status, and so the algorithm can find no match. ∎

Two observations may be made here. The first is that the order in which variables are considered by *bind* is only a function of the pattern, and not of the graph to which the pattern is being applied. Thus, this order may be computed prior to any actual matching attempt. This being the case, the sets $\overrightarrow{x}$ $\overset{\not\rightarrow}{x}$ $\overleftarrow{x}$ and $\overset{\not\leftarrow}{x}$ can also be pre-computed.

A second observation is that NCC's cannot be processed completely independently, as a match of one may cause a second one to go unmatched even when some match accommodating both exists. The proper way to proceed is either to place one seed from each NCC into the set B at initialization time or to make the success of step 1 of the *bind* algorithm contingent upon the successful matching of all other NCC's in the pattern via a recursive application of the entire algorithm.

3.2.2 An algorithm for general matching

The presence of a set-valued variable in an NCC creates special problems for a pattern matcher. Even if some mechanism creates the set of nodes that may be bound to such a variable, there is no good way to determine which subset of this node set should be taken as the variable's value. Since searching through all possible subsets is patently a bad idea, we take another tack.

The approach is to bind all singleton variables in a pattern before binding any set variables. We then use adjacency information given in the pattern matrix to determine the set-variable values, taking the sets to be as large as possible subject to the constraint that each may not contain any node bound to any other variable. We make the success of the singleton binding portion of the algorithm contingent on the success of the set-variable binding algorithm so that failures in finding set-variable values cause *bind* to consider other possibilities.

This dissertation does not contain an algorithm for matching NCC's containing no singleton variables. In such patterns, there are no seeds to help constrain the values of the set variables. In that case, the structural information in the pattern cannot be used effectively, and the search process degenerates into an exponential search. The omission of such an algorithm is an insignificant loss to us, as one of the criteria for deciding the quality of a flow analysis algorithm is its speed.

The first half of our general matching problem, binding all singleton variables in a pattern, could be done by re-defining NCC's to be comprised solely of singleton variables and then by binding them using the algorithm above. While this would undoubtedly work, it would be inefficient, since structural information that connects these restricted NCC's through set-valued variables would be lost. Consider, for example, the pattern shown in Figure 3.4, remembering that upper-case variables denote set-variables. Under this algorithm, nodes x and z would comprise their own NCC's, being separated from each other by the set-variable Y. A search based on this idea would ignore the important fact that z must be a distance of at most two away from x, and would treat each of x and z as independent seeds, making $\bar{x}$ and $\bar{z}$ both contain all nodes in G. This would result in an O(n^2) search.

```
      Y   z   *
    ------------
x |   1   *   *
Y |   *   1   *
* |   *   *   *
```

Figure 3.4: A general pattern with 1 NCC

Instead of redefining NCC, our algorithm processes singleton variables as usual, ignoring all set-valued variables. There comes a time when no unprocessed singletons in the NCC are adjacent to any variable represented in B. The algorithm then estimates the values of the set-valued variables which are adjacent to variables in PBV's in B so that it can use adjacency information to limit the region it needs to examine to find values of any remaining singleton variables. It does this estimation by provisionally binding each set variable X to its entire $\bar{X}$ set, calculated as being adjacent to provisionally-bound singletons in the usual way. It does not, however, install these provisional bindings in B, and so the $\bar{X}$ sets may overlap. It then repeatedly uses adjacency to these variables as a means of constructing $\bar{v}$ sets for any variables in the NCC which are adjacent to these set variables (again without considering intersections of these sets with the $\bar{X}$ sets) until it has done so for at least one singleton variable. The resulting $\bar{v}$'s may be larger than necessary, but they consist only of graph nodes that lie within the proper distance from provisionally bound nodes in B. The singleton matching process then proceeds recursively as per this paragraph.

Finally, after all singletons are matched, new $\bar{X}$ sets are computed for each set-valued variable X. These may intersect, so some procedure is necessary for whittling down the overlaps without leaving any of the $\bar{X}$ sets empty, if this is possible. This problem is solved by noticing that the relation between the variables and the graph nodes bound to them can be modeled by a bipartite graph. A straightforward bipartite graph matching algorithm may then be used to find a graph node for each set, ensuring that none will be empty, if possible. Each "unallocated" graph node may then be distributed to any set-variable's value set in which it would be a legal member.

3.2.3 The $*$ variable

By convention, the $*$ variable is used to match all nodes in the graph that do not take part in the graphical structure being searched for by a pattern; that is, it stands for "all the rest" of the graph nodes. It follows that all entries in the $*$ row and column are usually $*$'s. It is therefore unnecessary to compute explicitly the set of graph nodes that bind to $*$ when this condition holds. Thus, the algorithm may be modified to avoid this computation whenever it can do so, simply by neglecting to consider $*$ to be a variable. This means that considerable time savings are possible.

Example 10 The pattern

```
        y   *
      --------
  x |   1   *
  * |   *   *
```

simply looks for any edge in the graph. It should find a successful match after examining exactly one edge, and thus the matching process should complete in O(1) time.

We can add additional special checks in our algorithm to avoid calculating the value of $*$ in other cases as well.

Example 11 Consider the pattern

```
        y   *
      --------
  x |   1   *
  y |   0   *
  * |   0   *
```

A successful match of this pattern binds nodes a and b to variables x and y such that the edge $a \rightarrow b$ enters node b and no others do. Rather than calculate the value of $*$, the algorithm can examine the set of edges entering the node b, making sure that for all edges $c \rightarrow b$ in G, $c = a$. This examination completes in time O(1) rather than in time O($|G|$).

More formally, let P be a pattern and v be a singleton variable provisionally bound to graph node b. Suppose that $P_{*,v} = 0$, and that every variable w labeling a row with $P_{w,v}$ either 1 or $*$ is provisionally bound. The constraint encoded by the 0 in position $P_{*,v}$ may then be checked by verifying that for all edges $a \rightarrow b$ in G, the node a is provisionally bound to some variable in the pattern. If graphs are represented as an adjacency list coupled with a reverse adjacency list, this check is made by examining all graph nodes in b's reverse adjacency list, making sure that they are provisionally bound. The check takes time O(in-degree(b)). Zero entries in the $*$ row are treated *mutatis mutandis.*

A similar check cannot be made efficiently if the $*$ row or column contains a 1 entry unless information has been maintained giving the size of the graph and the in- and out-degrees of each of the graph nodes. If this information is available, then a simple subtraction and comparison of edge counts is sufficient to decide the validity of the provisional bindings.

Should the $*$ variable's node set actually be needed, it may be computed after all other variables are bound simply by binding $*$ to all remaining unbound variables.

3.3 Parameterizing the Disambiguator

This section modifies the disambiguating pattern matching algorithm given above so that it may emulate a variety of other disambiguators. Parameterizations may be provided at pattern matching time. This ability to parameterize the disambiguator at pattern-matching time results in a very useful tool that can make use of a programmer's knowledge of the graph being dealt with to make the matching process fast or to match graph features in some particular order.

The primary difference that distinguishes two pattern matching algorithms based on a search mechanism like that in the previous section is the order in which they consider graph nodes when searching for the value of a singleton. This order in turn is determined by the ordering $\prec$ on graph nodes. The disambiguating algorithm presented in the previous section imposes its own default node ordering, and represents $\bar{v}$ sets as lists sorted with respect to this order.

3.3.1 Node orderings

It follows that an effective way of modifying the behavior of the matching algorithm is to allow it to take other graph node orderings into consideration. We will allow graph node orderings to be computed by the user and provided to the pattern matcher in the form of a graph organized as a list of nodes. This graph is suitable for controlling search order, since it, or a processed form of it, may be used to order the $\bar{v}$ lists. In particular, if v is a seed variable, then $\bar{v}$ is merely a copy of this list. Should such an ordering graph not include all nodes in the graph being searched, the search will only occur over the subgraph it describes. Thus, the mechanism also provides a simple way to restrict the scope of matches and transformations.

Many algorithms on graphs make use of node orderings to distinguish various types of edges. For example, given a reverse depth-first search ordering *dfs* on the nodes in a flowgraph, looping edges (also called back edges or fronds) are those having the property that $dfs(\text{tail}) < dfs(\text{head})$. Since this kind of node ordering information is useful, and since it is already incorporated into our generic matcher by virtue of the preceding paragraph, we make it available as a pattern matching parameter by allowing relational annotations within patterns of the form $v_1 \prec v_2$ for pattern variables v_1 and v_2. Such an annotation means that for a match to succeed, the value bound to v_1 must be less than the value bound to v_2 under the prevailing node ordering. If, say, the variable v_2 is considered after v_1 in the *bind* algorithm, then this annotation is used when constructing $\bar{v_2}$ to restrict the legal choices for v_2. This restriction of possibilities means that the use of such node ordering constraints may actually speed-up the searching process.

3.3.2 Based matching

So far, we have only been concerned with pattern matches wherein all information necessary for determining the success of a matching operation is contained in the template. It may frequently be the case, however, that information in the form of variable bindings produced by a previous match can be used to good advantage to speed-up the matching process. If we were to allow pattern variables to be bound to graph nodes at the outset of the matching process, it would be desirable for the pattern matcher to be able to use this additional information to constrain the search. Such a mechanism would give very fine-grained control over the actions and speed of the pattern matcher. This section describes how the above algorithm may be modified to take this kind of information into account.

Definition 19 A *based* pattern-match is one in which every NCC of the pattern contains at least one previously-bound variable.

The term "based" comes from the fact that the bound variables serve as basepoints, anchoring the pattern match operations to particular sections of the graph. Conversely, we define a *free* pattern-match to be one in which no pattern variables are bound. Based and free pattern matches are two ends of a continuum; we say a match is *partially based* if some but not all of the pattern's NCC's contain bound variables.

The behavior of the pattern matcher when presented with a based pattern should be governed by the following consistency requirement:

Requirement 4 Let P be a pattern, and let φ be a function mapping P's variables to the sets of graph nodes to which those variables become bound in a successful free match of P to graph G. Any based match of P to G having each based pattern variable v pre-bound to $\varphi(v)$ must also succeed.

Requirement 4 ensures that based matching and free matching act consistently, that is, that a based matching will never need to bind a previously-bound variable to a different value, and that the binding of a variable to a value resulting from a successful matching will not invalidate that match. This point is important, since the success of a match should only depend on the structure of the graph, the structure of the pattern, and constraints imposed by variable bindings. It should not depend on the time sequence of the same set of bindings.

We formalize this concept by declaring that the variable binding constraint information modifies the behavior of the pattern matcher only by restricting the acceptable choices of β and ψ. The additional information thus only serves to eliminate possible matches from consideration; it never introduces new possibilities.

The algorithm for performing a based match is a simple modification of the algorithm for free matching. First, one needs to perform a verification that the edges that the pattern demands exist between bound nodes do in fact exist, and that the edges that the pattern forbids do not. The verification algorithm checks the submatrix of the pattern matrix determined by the bound variables in the pattern against the corresponding edges in the graph to make sure they agree. The correspondence is given by the bindings of the variables. If position $P_{x,y}$ in the matrix is 1, then for every pair of nodes a and b in G such that $\psi(a) = \beta(x)$ and $\psi(b) = \beta(y)$, the edge $a \rightarrow b$ is in G. If $P_{x,y}$ is 0, no such edge may be present in G. And if $P_{x,y}$ is a $*$, it doesn't matter whether or not there is such an edge in G. The result of the algorithm is a single success or failure indication.

After such a verification has completed successfully, the previously-described search algorithm is used with the bound variables serving as seeds. This is done by initializing B to contain a special PBV for each such v, with the associated $\bar{v}$ containing only what v is bound to. The rest of the matching algorithm remains unchanged.

3.3.3 Informal analysis

The worst-case running-time of the matching algorithm is, of course, exponential. We can, however, say a bit more about its running time with the kinds of patterns that are typically used in flow analysis.

A free pattern is used primarily for finding an edge in a graph, perhaps one satisfying a node-ordering constraint. Such a pattern with no such constraints always matches in constant time, since the first edge it examines satisfies it and since the value of the $*$ variable need not be computed explicitly. The same pattern with a node ordering constraint at worst works in time O(|E|), where E is the set of edges in the graph. The reason for this bound is that the matching algorithm has to examine every outgoing edge from every graph node exactly once in the worst case.

Simple based patterns are amenable to analysis. Suppose a simple based pattern has exactly one NCC. Let λ be the number of unbound variables in the NCC. Then all of the matching activity takes place within a distance of λ from the set of bound nodes at which the pattern is being applied. If d is the maximum node degree (in- or out-) within that area, then the complexity of the pattern match is at worst $\mathrm{O}(d^{\lambda})$, in that $|\bar{v}| \leq d$ for every $\bar{v}$ set encountered during the search. λ is, of course, bounded by a constant for each pattern. Should it be the case that all edges indicated by the pattern either flow away from or toward the bound nodes, d can be restricted to either the maximum out-degree or the maximum in-degree in the search area. For example, a T_2' transformation's pattern has four variables. Suppose that the "top" one (x in Figure 3.2) is bound. Since the usual maximum node (out-) degree for a flow graph is two, a based T_2' match can be expected to consider at most 8 sets of PBV's in B. If another node is bound instead, then the number of comparisons may be greater, since there is no corresponding maximum in-degree of a node in a flow graph.

Finally, general patterns are usually used for finding certain node sets, as in the "demultiplexing" example in Figure 3.5. In this case, where the set variables are all at the periphery of the pair of singletons v and w, the complexity is no more than if the pattern were a simple pattern, since the $\bar{X}$ $\bar{Y}$ and $\bar{Z}$ sets need to be computed anyway just as if X Y and Z were singletons. The only additional cost would be in making sure that the set values were pairwise disjoint, which is a moot question in the example given anyway.

```
      X   Y   Z   *
    ----------------
 v |  0   1   1   0
 w |  1   0   1   0
 * |  *   *   *   *
```

Figure 3.5: A "demultiplexing" general pattern (v and w are bound)

Chapter 4

A Graph Programming Language

This chapter describes a programming language based on the foregoing graph transformation techniques. This language is quite small, easy to read and use, and sufficiently powerful to allow for the concise implementation of flow analysis algorithms. The linguistic base underlying the graph-transformational computational model provides flow-of-control primitives, a scoping discipline for variables, and basic facilities for constructing and handling graphs. These language features are described in this chapter. Chapter 5 will exhibit several useful functions on graphs written in this language, and Chapter 6 will make use of these functions to construct flow analysis algorithms.

4.1 A Common Lisp framework

The language described here was designed as a set of extensions to Common Lisp (hereafter referred-to simply as Lisp). It should not, however, be viewed as a language containing or requiring all the power of Lisp, but rather as a small language unto itself that simply borrows from Lisp a few underlying features such as variables and control-flow constructs. It does not rely on the existence of an underlying interpreter, and could just as well be implemented as a compiled Algol-style language. Its power comes from its computational model of transformations on graphs rather than from any linguistic innovations. The choice of Lisp as a base was motivated by concerns of compatibility with other software being developed at Berkeley, and because this choice afforded easy prototyping of the graph transformation language.

This chapter does not attempt to describe the Common Lisp language or the workings of the LISP facilities used to implement the graph-based extensions; rather, it aims at describing those extensions under the assumption that the reader is already familiar with LISP.

4.2 The graph datatype

The first necessary extension to LISP is the addition of a *(directed) graph* data type. Graphs are, of course, collections of nodes and edges. Nodes will be represented as simple integers, so as to allow their easy manipulation by programmers. This choice introduces no confusion, since none of our graph transformation algorithms, including our flow analysis algorithms, will have need of an integer data type. Each graph is treated as a set of edges on this universal set of nodes. Thus, there may be more than one graph on the same set of nodes, and two such graphs may or may not be completely disjoint. The set of nodes that are considered to lie in a graph contains exactly those nodes that are endpoints of the edges in the graph.

A graph may thus be viewed as nothing more than a set of edges. The idea is that a programmer will construct a graph on a set of nodes that represent some other computational objects in order to represent a relationship that holds among those objects. The integer representation of nodes is then convenient in that it allows for simple mappings between the graph nodes and the objects they represent. A built-in function, `new-node`, is provided that, when called, returns a node that is guaranteed to be unused.

Edges, on the other hand, are not entities in and of themselves, and have no explicit representation that the programmer can obtain or manipulate. Edges may only be added to or removed from graphs, and only by applying transformations. An extension presented in Section 4.5 will, however, give us a bit more flexibility in their use.

The manipulation of graphs other than that afforded by pattern matches and transformations is done with the following two primitive built-in functions: `make-graph` creates and returns a new (empty) graph, while `graph-changed` returns a Boolean value indicating whether or not the graph was changed by a transformation. The Boolean flag is actually a field in the *graph* data structure, and can be set back to `nil` via `setf` in preparation for any subsequent transformation attempt.

4.3 Patterns, transformations and variables

The extension of LISP by the incorporation of patterns and transformations is fairly straightforward, except that we need to define an interface between LISP variables and pattern variables and a scoping mechanism so that the entire system functions in a self-consistent manner. While LISP provides a scoping mechanism for variables, this mechanism does not apply to pattern variables, which are manipulated by the graph transformation system rather than by the LISP interpreter, and hence are not recognized by LISP.

Our language implements such a scoping mechanism by breaking pattern matches into two parts: a *pattern declaration* and a *pattern application*. These parts correspond roughly to a function declaration and a function call, and provide an easily-understood mechanism to control the lifetimes of bindings of pattern variables.

4.3.1 Pattern declarations

A pattern declaration contains, along with a labeled pattern matrix, declarations of the pattern's variables. Each pattern variable must be declared to be an *in* variable, an *out* variable, or a *local* variable. *In* variables are those that are considered to be bound prior to pattern-matching; that is, their bindings get passed in to the pattern matcher from outside. They serve as the base-points in based matches. *Out* variables are those that will propagate their bindings out of the pattern application after a successful match. They are unbound immediately prior to the matching attempt. *Local* variables, which are also unbound at the beginning of a match, do not communicate their values out of the pattern or transformation.

Patterns are declared and constructed by the macro `gpattern`, which takes four arguments: *in-list*, *out-list*, *local-list*, and *labeled-matrix*. Each of *in-list*, *out-list*, and *local-list* is a list of pattern variables (e.g. `(x y z)`), and declares the variables in its list to be of the designated type. The *labeled-matrix* is a list of lists that represents the pattern itself. The first sublist of this list contains the column labels of the pattern. Each subsequent sublist represents a row of the pattern, its first element being the label of that row. Figure 4.1 shows an example pattern. In that figure, the pattern variable `b` is an *in* variable, `a` and `c` are *out* variables, and `x`, `y` and `z` are *local* variables. The fourth argument, the labeled matrix, is arranged matrix-wise to allow for easy reading. Its top row, the first sublist, contains the column labels. Each of these labels, along with each row label, must be a pattern variable declared as an *in*, *out* or *local*.

```
(gpattern (b)(a c)(x y z) ((   c a z x *)
                           (a  1 0 1 1 *)
                           (b  1 1 0 0 *)
                           (x  1 0 * * *)
                           (y  0 0 1 * *)
                           (*  * * * * *)))
```

Figure 4.1: Example pattern definition

As was mentioned in an earlier chapter, a naming convention distinguishes singleton variables from set variables in patterns. If the first letter of a variable's name is upper-case, the variable is a set variable, otherwise it is a singleton. It should be noted that this convention necessitates a case distinction within COMMON LISP. Fortunately, most LISP implementations allow this distinction to be made.

4.3.2 Pattern applications

A pattern application contains a pattern, a graph to attempt to match with it, a list of values to be bound to *in* variables prior to the matching attempt, and a list of LISP variables to receive the bindings of *out* variables after the match, should it succeed. *In* parameters are passed by value by being copied to the *in* pattern variables just prior to the match. *Out* values are passed by result, being copied from the *out* pattern variables to the *out* LISP variables after a successful match.

This application of a pattern to a graph is done by the macro `pmatch` that takes four arguments: a *pattern*, a *graph*, a list of *in-values*, and a list of *out-variables*. The pattern argument may either be the result of a previous `gpattern` macro invocation or a list containing arguments to such a call. `pmatch` returns a Boolean value indicating whether or not the match succeeded, and only binds the *out* LISP variables after a successful match. Figure 4.2 shows an example pattern application. The *in* values held by the LISP variables `m` and `n` are copied to the *in* pattern variables `p` and `q` prior to the match. Should the match succeed, that is, if the edge `m` $\rightarrow$ `n` is in G, the `pmatch` call will return `t`. Since there are no *out* variables, none will be bound.

```
(pmatch ((p q)()() ((  q *)
                    (p 1 *)
                    (* * *))) G (m n) ())
```

Figure 4.2: Example pattern application

4.3.3 Transformations

Transformations are similarly considered in two parts, a *transformation declaration* and a *transformation application*. Transformation declarations contain variable declarations and pattern matrices identical in form to those in pattern declarations, along with *replacement matrices*.

A transformation is declared and constructed by the macro **pxform**, which takes four (not five) arguments: *in-list*, *out-list*, *local-list*, and *labeled-matrix-pair*. Each of *in-list*, *out-list*, and *local-list* is, as before, a list of pattern variables, and declares them to be of the designated type. The *labeled-matrix-pair* is a list of lists that represents both the pattern and the replacement. Even-numbered sublists form the pattern, while odd-numbered sublists form the replacement. The first two sublists in the list contain the column labels of the pattern and of the replacement, respectively. They must therefore be the same. Each subsequent pair of sublists represents a row of the pattern and a row of the replacement. The first element of each sublist is the label of that row. Figure 4.3 shows an example transformation. In that figure, the pattern variable **x** is an *in* variable, and **y** is an *out* variable. The fourth argument, the labeled matrix-pair, is arranged to appear as a pair of matrices. The three sublists on the left form the pattern, while the three sublists to the right form the replacement. It should now be apparent why the pattern and replacement are expressed as an interleaved pair of matrices, rather than as two separate lists.

```
(pxform (x)(y)() ((  y *) (  y *)
                  (x 1 *) (x 0 =)
                  (* * *) (* = =)))
```

Figure 4.3: Example transformation definition

Paralleling pattern applications, transformation applications contain a transformation, a graph, a list of *in* values, and a list of *out* variables. The function of each of these parts is almost identical to that of its counterpart in pattern applications. The two exceptions are that the transformation may alter the graph if the pattern match incorporated into it succeeds, and that local pattern variables maintain their bindings until any such alteration is completed. Local pattern variables may thus be used to communicate between the pattern and the replacement of a transformation.

The macro that performs transformation applications is `pxapply`. Its first argument, the transformation, may either be a transformation returned from a previous call to `pxform`, or may be a list containing the arguments that would be given to such a call. It returns a Boolean value indicating whether or not the pattern-match portion of the transformation succeeded. The replacement action and the binding of *out* variables only happens if the pattern match succeeds. Figure 4.4 shows a pattern application that adds an edge from node `m` to every other node in G. The value of `m` is bound to the pattern variable `z` prior to the pattern match. This match necessarily succeeds, since it describes no constraints. The 1 in the replacement matrix says to add an edge from the node bound to `z` to each node bound to $*$, or in other words, to every other node in the graph.

```
(pxapply ((z)()() ((  *) (  *)
                   (z *) (z 1)
                   (* *) (* =)))  G (m) ())
```

Figure 4.4: Example transformation application

In both pattern applications and transformation applications, the graph is passed to the pattern matcher or transformer by reference. Also in both cases, the application returns (LISP-function-style) a Boolean value indicating the success or failure of the pattern match. This value may then be tested by the usual LISP control-flow constructs.

4.4 Node orders

Node orders, the necessity for which was explained in Chapter 3, are incorporated into the language as a second data type. They are incorporated this way rather than as graphs because a simple list of nodes (such as a linear directed graph) is not a suitable form for efficiently checking that a pair of nodes satisfies certain semantic constraints (such as $x < y$). That problem is rectified by the introduction of this new datatype, each instance of which expresses a read-only mapping between nodes and their positions in an ordered list. We provide one constructor function, `node-order`, that takes a linearly-organized graph as input and returns an object of type *node-order*. This object may be given to the forms `pmatch` and `pxapply` as an optional last argument; doing so causes them to use the specified order instead of the default order in free searches and in semantic constraint checks. Constructing the desired linear graph is left to programmers who can write graph traversal programs based on pattern matching and transformations. An example will be shown in Chapter 5 after several other auxiliary graph functions are developed.

The default ordering of nodes is the order in which they are created with `new-node`. This order is typically not very useful unless one is extremely careful about the way in which one originally enters graphs into the system. It is highly recommended that the customizable node order facility be used if one wishes to represent syntax trees or other data structures in which order of visitation is important.

4.5 Edges

We have seen how the pattern-matcher can bind graph nodes to variables and can reference those bindings in subsequent matches. It is sometimes desirable to be able to refer to edges in similar ways. An extension of the notation used for pattern matrix entries allows us to do so.

4.5.1 Edge bindings

Let x and y be (singleton) pattern variables in a pattern P such that $P_{x,y}$ is either 1 or $*$. After a successful match, there is a correspondence between the position $P_{x,y}$ and the edge in the matched graph that leaves the node bound to x and enters the node bound to y. We cause this edge to be bound to a variable z by writing the matrix entry in position $P_{x,y}$ as $1.z$ or $*.z$, instead of 1 or $*$. It of course makes no sense to write any entry as $0.z$, since the edge in question then necessarily doesn't exist and cannot be bound to z.

If, say, x should instead be a set-valued variable (that is, X), the set of edges corresponding to the pattern position $P_{X,y}$ could be captured with the predictable notation

$1.Z$ or $*.Z$. This is just an extension of the convention that lower-case variable names are used to capture single values, while upper-case names are used to capture sets of values.

This "dot notation" allows us to name an edge, given its endpoints. Of course, the inverse operation – naming an edge's endpoints given the edge – is also well-defined, since all edges are directed, and so the edge's two endpoints are distinguishable from one another. We introduce a second notation to allow us to perform this inverse operation. Again, we will annotate the pattern matrix position $P_{x,y}$, this time with a variable that is already bound to an edge. The pattern matcher will then be able to identify x and y uniquely as that edge's tail and head, respectively, and bind them to these values prior to beginning the normal matching process. Let **e** be the entry (1, 0, or $*$) in position $P_{x,y}$ in the pattern matrix, and let z be the variable that is already bound to some edge (or set of edges) in the graph. We annotate the matrix entry **e** by rewriting it as $\mathbf{e}@z$. The entry **e** and the variable z do not interact directly; z is merely used as a strong "hint", and is notationally glued to **e** only so that they may share one position in the pattern matrix.

In the case that **e** is a 1 or $*$, we may combine the above two notations, writing a matrix entry as $\mathbf{e}@x.y$. Such a matrix entry means that x should be used as a hint to give bindings for the variables labeling the entry's row and column, the pattern-match should proceed, and if it is successful, the graph edges that correspond to **e**'s position in the pattern matrix should be bound to y. Interestingly enough, the value of y after this operation need not be equal to the value of x, especially if x and y are set-valued variables (i.e. X and Y). X may have received its value from a successful match of another pattern to a different graph, and the edges to which it refers may not all exist in the current graph. An entry of the form $*@X.Y$ can match a different set of edges from those in X. The guarantee is only that the endpoints of the edges in Y will be a subset of the endpoints of the edges in X.

Note that the matrix entry $0@z$ is a legal notation, and a pattern containing it matches a graph only if the edge described by the value of z is not in that graph.

Edge variables interact with the variable scoping mechanisms the same way that node variables do, and so each must be declared in a pattern or transformation as an *in*, *out* or *local* variable.

4.5.2 Edges as nodes

It is sometimes useful to be able to create graphs whose nodes are the edges of other graphs (for example, the line graph dual [14]), or to refer to edges as if they were objects instead of relations between objects. Such a facility is especially important for some data-flow analysis algorithms that prefer to attach local information propagation functions to flowgraph edges instead of to nodes. To facilitate these kinds of edge manipulations, we declare that for each possible edge there exists a unique graph node associated with that edge. In practice, we allocate this node when the "to-be-corresponding" edge is bound to a variable for the first time. The node chosen to correspond to that edge will thus never have appeared previously in any graph.

The correspondence between edges and their associated nodes is maintained internally by the run-time system. It can thus be the case (and is) that variables only bind to nodes, never to edges. That is, whenever an edge is to be bound to a variable, its corresponding node is bound to that variable instead. Should a bound variable occur in a context requiring an edge binding, the coercion back is performed implicitly if, of course, there is some edge (or set of edges) corresponding to the node (or set of nodes) bound to the variable. However, if there is no such corresponding edge or edge set, the variable is considered to be bound to the empty set of edges, and the pattern match in which it appears fails.

4.6 A library of transformations

The preceding part of this chapter has described a programming language for operating on graphs. This language appears to be rather low-level, having as its operators only the graph transformations developed in Chapter 2. Nonetheless, the language may be extended to include higher-level operations by means of a library of useful transformations and flow analysis procedures. Such a library has been developed, and is considered to form part of the language. The contents of this library is dealt with in the next two chapters.

The idea of extending a language by means of a subroutine library is not a new one. This particular library is only remarkable in that it provides high-level operations that can be used without modification to solve a wide variety of flow analysis problems. This in turn would not be unusual, except that it has been accomplished by the design of the language instead of by a great deal of discipline of a programming team. Every computation expressed in this language depends ultimately upon one data type, graphs, and one way of manipulating objects of that type, graph transformations. Any two correct algorithms for computing, say, available expressions must necessarily work on the same flow graph and produce exactly the same result graph. The difference between two such algorithms can be only in the way in which the *gen* and *kill* graphs are specified, or in the order of their arguments. Thus, ensuring that two such algorithms are interchangeable is only a matter of checking that they have been written to accept the same argument lists.

Chapter 5

Graph Manipulations

The graph transformation techniques described in Chapters 2 and 3 are quite powerful, but rather low-level to program with. This chapter shows how, with the aid of COMMON LISP functions and macros, these techniques may be used to build higher-level graph manipulators that are more amenable to use, and begins to show the utility of the transformational paradigm. The following chapter will use these aids to implement flow analysis algorithms.

The set of functions presented here purports neither to be the only nor the best possible set of language extensions for the task of performing flow analysis. Further experimentation with the system will doubtless reveal better ways of doing things. This set is easily understood, however, and is sufficient for the task, as is shown in the following chapter.

5.1 Adding and deleting edges

Perhaps the most fundamental operation on a graph is the addition or removal of a particular edge. Since these operations will occur often, we provide, for the sake of brevity, a pair of functions for doing so. These are shown in Figure 5.1.

The pattern portion of each of these transformations necessarily matches any graph successfully.[1] This way of writing the transformations causes them to be somewhat more efficient than does the approach of checking whether the edge exists and only conditionally

[1] Since `x` and `y` are already bound to nodes, the graph does not have to be nonempty for these patterns to match.

```
(defun +!edge (v w G)
    (pxapply ((x y)()() ((  y *) (  y *)
                         (x * *) (x 1 =)
                         (* * *) (* = =))) G (v w) ()))
```

```
(defun -!edge (v w G)
    (pxapply ((x y)()() ((  y *) (  y *)
                         (x * *) (x 0 =)
                         (* * *) (* = =))) G (v w) ()))
```

Figure 5.1: Functions for adding and deleting edges

modifying the graph. The reason for this difference in efficiency is that the graph transformer has to do this check internally in any case. Both of these functions thus return `t`.

5.2 Combs

We define a *comb graph* to be a directed graph all of whose edges emanate from a single vertex. For each node x in a graph G, define the *comb at x in G* to be the subgraph of G consisting of all arcs emanating from x (along with their endpoints). Given a graph G and a node x, we can construct a graph equal to the comb at x in G with the algorithm shown in Figure 5.2.

```
(defun comb (G x)
   (let ((Y (make-graph)) (A nil))
      (when (pmatch ((x)(A)() ((  A *)
                                (x 1 0)
                                (* * *))) G (x) (A))
            (pxapply ((x A)()() ((  A *) (  A *)
                                  (x * *) (x 1 =)
                                  (* * *) (* = =))) Y (x A) ()))
      (when (pmatch ((x)()() ((  x *)
                               (x 1 *)
                               (* * *))) G (x) ())
            (pxapply ((x)()() ((  x *) (  x *)
                                (x * *) (x 1 =)
                                (* * *) (* = =))) Y (x) ()))
      Y))
```

Figure 5.2: Algorithm for computing combs

The algorithm first binds `A` to the set of all graph nodes not equal to `x` to which arcs from `x` point, and enters these same arcs into the initially empty graph `Y`. The second half of the algorithm then checks for an arc from `x` to itself, entering it into `Y` if it exists. The graph `Y` is returned. This is a good example of how arcs may be copied from one graph to another: once the relevant nodes are bound to variables, arcs connecting those nodes may be added to the second graph.

`(comb G x)` creates a comb graph in which each edge points to an immediate successor of `x`. It is sometimes useful to create a comb graph wherein each edge points to a predecessor of `x` instead. The algorithm in Figure 5.3 does exactly this. The call `(rcomb G x)` is equivalent to `(comb (reversegraph G) x)`, but is much more efficient, as it does not need to compute the reverse of all of `G`.

```
(defun rcomb (G x)
   (let ((Y (make-graph)) (A nil))
      (when (pmatch ((x)(A)() ((  x *)
                                (A 1 *)
                                (* 0 *))) G (x) (A))
            (pxapply ((x A)()() ((  A *) (  A *)
                                  (x * *) (x 1 =)
                                  (* * *) (* = =))) Y (x A) ()))
      (when (pmatch ((x)()() ((  x *)
                               (x 1 *)
                               (* * *))) G (x) ())
            (pxapply ((x)()() ((  x *) (  x *)
                                (x * *) (x 1 =)
                                (* * *) (* = =))) Y (x) ()))
      Y))
```

Figure 5.3: (rcomb G x) = (comb (reversegraph G) x)

5.3 Iterators

Iterators over graph nodes and edges are important and powerful tools. The first iterator we will implement, and probably the most straightforward, is `foreachnode`, the algorithm for which is shown in Figure 5.4. Iterators are implemented using LISP macros so that the body of code that they iteratively execute lies within the scope of the iteration variables.

The algorithm works by first creating a new node `z` and a new graph `C`.[2] These

[2] For notational convenience, `gensym`'d symbols in LISP macros (such as `z` and `C` in `foreachnode`) will be referred to throughout this dissertation by the names of the LISP variables binding to those symbols.

```
(defmacro foreachnode ((x G) &body body)
  (let ((C (gensym)) (z (gensym)))
    `(let ((,z (new-node)) (,x nil))
        (pxapply ((z)()() ((  *) (  *)
                           (z *) (z 1)
                           (* *) (* =)))  ,G (,z) ())
        (let ((,C (comb ,G ,z)))
           (pxapply ((z)()() ((  *) (  *)
                              (z *) (z 0)
                              (* *) (* =)))  ,G (,z) ())
           (while (pxapply ((z)(x)() ((   x *) (   x *)
                                      (z  1 *) (z  0 =)
                                      (*  * *) (*  = =))) ,C (,z) (,x))
              ,@body)))))
```

Figure 5.4: Iterator over graph nodes

are used to keep track of which nodes still need to be visited. `C` is initialized to be a comb graph, the teeth[3] of which are all the nodes in the graph `G`. This initialization is done by adding to `C` an edge from `z` to each of its nodes, then taking the comb of the resulting graph at `z`. The while loop test uses a `pxapply` to remove one of the edges in `C`, simultaneously binding the node this edge enters. The body of the iteration is then executed with this node bound to the iteration variable. As long as this process is able to remove edges leaving `C` (that is, the algorithm can find un-visited nodes), the while loop repeats.

Our second iterator iterates over all graph edges. It is shown in Figure 5.5.

```
(defmacro foreachedge (((x y) G) &body body)
   `(foreachnode (,x ,G)
       (let ((F (comb ,G ,x)))
            (while (or
                      (pxapply ((x)(y)() ((   y *)(   y *)
                                          (x  1 *)(x  0 =)
                                          (*  * *)(*  = =))) ,F (,x) (,y))
                      (and
                          (pxapply ((x)()() ((   x *)(   x *)
                                             (x  1 *)(x  0 =)
                                             (*  * *)(*  = =))) ,F (,x) ())
                          (setf ,y ,x)))
                 ,@body))))
```

Figure 5.5: Iterator over graph edges

This iterator uses `foreachnode` to visit each node in the graph, using `comb` to

[3] "leaves", but using a less arboreal metaphor.

construct a copy of the comb at that vertex. Each edge in the original graph thus ends up being copied into exactly one of these combs. The iterator then executes the macro's body once for each edge in the comb with the endpoints of this (directed) edge bound to the iteration variables. As in the algorithm for `comb`, a special pattern is needed to check for looping edges.

A variant of this iterator may be defined that also binds each edge to a variable as explained in Subsection 4.5.2. This modification results in the algorithm shown in Figure 5.6. Of course, this version is less efficient than the simpler version because each `pxapply` may cause a new mapping to be created between the edge being bound and the node that is to represent it. Sometimes, however, this behavior is exactly what is wanted. An application of this feature is shown at the end of the next section.

```
(defmacro foreachedge-with-edge-binding (((x e y) G) &body body)
   `(foreachnode (,x ,G)
       (let ((F (comb ,G ,x)))
            (while (or
                       (pxapply ((x)(y e)() ((    y  *)(   y *)
                                             (x  1.e *)(x  0 =)
                                             (*   *  *)(*  = =))) ,F (,x) (,y ,e))
                       (and
                           (pxapply ((x)(e)() ((    x  *)(   x *)
                                               (x  1.e *)(x  0 =)
                                               (*   *  *)(*  = =))) ,F (,x) (,e))
                           (setf ,y ,x)))
                   ,@body))))
```

Figure 5.6: An iterator that binds graph edges

Other iterators, such as `foreachchild`, which iterates over all nodes reachable via arcs from a particular node, are also straightforward to define. One simply combines the above iterators with particular subgraphs, such as combs. For examples, see Figures 5.7 and 5.8. More complex iterators will be developed in Chapter 6.

```
(defmacro foreachchild ((y x G) &body body)
   (let ((dummy (gensym)))
      `(foreachedge ((,dummy ,y) (comb ,G ,x)) ,@body)))
```

Figure 5.7: Iterator over child nodes

```
(defmacro foreachparent ((y x G) &body body)
   (let ((dummy (gensym)))
      `(foreachedge ((,dummy ,y) (rcomb ,G ,x)) ,@body)))
```

Figure 5.8: Iterator over parent nodes

5.4 Some global graph transformations

The iterators just described may be used to implement several global graph operations. The first and simplest such global graph function is a function that makes a copy of a graph. The version shown in Figure 5.9 uses `foreachedge` along with `+!edge` to copy each edge in the graph `G` into a new empty graph that it returns.

```
(defun copygraph (G)
  (let ((X (make-graph)))
     (foreachedge ((x y) G) (+!edge x y X))
     X))
```

Figure 5.9: Algorithm to copy a graph

The next simplest operations on entire graphs perform some logical operation on corresponding edges in two graphs.

Definition 20 The *union* of two graphs G and H is the graph whose edge set is the union of the edge-sets of G and H, and whose node-set is the union of the node sets of G and H.

An algorithm for computing graph unions is shown in Figure 5.10.

```
(defun union (G H)
  (let ((X (copy-graph G)))
     (foreachedge ((x y) H) (+!edge x y X))
     X))
```

Figure 5.10: Graph unions

Definition 21 The *difference* of two graphs G and H is the graph whose edge-set is the set difference $edges(G) - edges(H)$. Its nodes are exactly the endpoints of its edges.

An algorithm for computing graph differences is shown in Figure 5.11.

```
(defun difference (G H)
   (let ((X (copy-graph G)))
      (foreachedge ((x y) H) (-!edge x y X))
      X))
```

Figure 5.11: Graph differences

Definition 22 The *intersection* of two graphs G and H is the graph whose edge-set is the intersection of the edge sets of G and H. Its nodes are the endpoints of its edges.

An algorithm for computing graph intersections is shown in Figure 5.12. The pattern is used to test the graph H for the existence of edges found in graph G; if an edge exists in both graphs it is added to the new graph.

```
(defun intersection (G H)
   (let ((X (make-graph))
      (foreachedge ((x y) G)
         (when (pmatch ((x y)()() ((  y *)
                                   (x 1 *)
                                   (* * *))) H (x y) ())
               (+!edge x y X))))
      X))
```

Figure 5.12: Graph intersections

An operations such as union, difference, or intersection can also be done in-place, modifying one of its arguments. This obviates the need for making an explicit copy of that argument. An example of one of these in-place operations is shown in Figure 5.13.

```
(defun union! (G H)
     (foreachedge ((x y) H) (+!edge x y G))
     G))
```

Figure 5.13: In-place graph unions

Definition 23 The *reversal* G^R of a graph $G = (V, E)$ is the graph (V, E') such that the edge $x \to y \in E'$ if and only if the edge $y \to x \in E$.

An algorithm for computing graph reversals is shown in Figure 5.14.

```
(defun reversegraph (G)
   (let ((X (make-graph)))
      (foreachedge ((x y) G) (+!edge y x X))
      X))
```

Figure 5.14: Graph reversals

There are two other important global graph transformations that are somewhat more complicated. The first combines two graphs in the manner of relational composition, while the second un-does this process in a simple but useful way.

Definition 24 The *composition* $G \circ H$ *of two graphs* G *and* H *is the graph that contains an edge* $p \rightarrow q$ *if and only if there exists some node* x *such that the edge* $p \rightarrow x$ *is in* G *and the edge* $x \rightarrow q$ *is in* H.

If the graphs G and H are thought of as representing relations, then their graph composition as defined exactly represents the relational composition $G \cdot H$. The algorithm requires nothing more than what we already have developed, and is shown in Figure 5.15.

```
(defun compose (G H)
   (let ((X (make-graph)))
      (foreachedge ((w x) G)
         (let ((F (comb H x)))
            (when (not (empty-graph F))
                  (foreachedge ((y z) F)
                     (+!edge w z X)))))
      X))
```

Figure 5.15: Graph compositions

The last algorithm in this section attempts to un-do the effects of a composition, producing two graphs from its single graph argument in such a way so that the composition of the two is equal to the original graph. Of course there is not a unique way to create these graphs, since many pairs of graphs may compose to the same thing.[4] The approach taken here is to split each edge in the original graph into two edges that will then compose to form the original edge. The algorithm is shown in Figure 5.16.

The algorithm uses the iterator `foreachedge-with-edge-binding` described in the previous section to bind each edge `x` $\rightarrow$ `y` in the original graph to a unique node `e`. It

[4] As evidenced by $R \cdot R^* = R^* = R^* \cdot R^*$ for any finite relation $R \neq R^*$

```
(defun factor (G)
   (let ((G1 (make-graph)) (G2 (make-graph)))
      (foreachedge-with-edge-binding ((x e y) G)
         (+!edge x e G1)
         (+!edge e y G2))
      (values G1 G2)))
```

Figure 5.16: Graph factoring

then thinks of each such new node as being the midpoint of its associated edge, and creates edges `x` $\rightarrow$ `e` and `e` $\rightarrow$ `y`, the "first half" and "last half" of the edge `x` $\rightarrow$ `y`. Graph `G1` is used to collect the first halves, while `G2` collects the second halves. Accordingly, `G1` is a disjoint union of comb graphs, while `G2` is the reverse of such a union.

The resulting graphs `G1` and `G2` have an interesting and useful property. We already know that `(compose G1 G2)` is equal to the original graph `G`. `(compose G2 G1)`, on the other hand, is the so-called *line (di)graph*[5] of `G`, whose nodes correspond to the edges of `G`, and that has an edge from (node) e to (node) f if there exists a node x in `G` such that edge e enters x and edge f leaves x. Line graphs will be very important in the next chapter for certain data flow analysis algorithms.

5.5 Emulating other data structures

A programming language with only one real data type may seem somewhat restrictive. However, graphs are sufficiently general that they may be used to emulate other data structures successfully. Operations on these data structures can be coded in terms of our transformations. This section outlines a few examples.

5.5.1 Sets of nodes

Creating a graph representing a set of nodes is uncomplicated. One merely creates a special node n used nowhere else, and forms a comb graph all of whose edges leave this new node. A node x is entered into the set by adding the edge $n \rightarrow x$ to the comb graph. Figure 5.17 exhibits an implementation of sets.

[5] This term is taken from [14]. Other terms used in the literature for this graph include the "derivative" graph, "derived graph", "edge-to-vertex dual", "covering graph", and "adjoint".

```
(defconstant SetHead (new-node))


(defun add-to-set! (x S)
   (+!edge SetHead x S))

(defun delete-from-set! (x S)
   (-!edge SetHead x S))

(defun set-member? (x S)
   (pmatch ((s x)()() ((  x *)
                       (s 1 *)
                       (* * *))) S (SetHead x) ()))
```

Figure 5.17: A set package implemented with graphs

Set union, intersection, and difference all work via the graph union, intersection and difference functions given previously. One may iterate over all elements in the set by means of the iterator `foreachedge`. One iterates over all edges in the (comb) graph, and ignores the binding of the `SetHead` node to the first of `foreachedge`'s iteration variables.

5.5.2 Stacks

It is often useful, especially when attempting to traverse a graph in some particular order, to push graph nodes onto a stack for later consideration. Presented here are two functions for managing a graph as a stack. The graph in question has a distinguished node (the head), and has the form of a linked-list (that is, it is acyclic and every node but one has exactly one successor). The functions are shown in Figure 5.18.

`gpush` adds an item to the front of the linked list (underneath the header node), while `gpop` removes and returns the node under the header. The body of `gpush`'s "when" clause is only invoked if the stack is empty, while the body of `gpop`'s "when" clause is only invoked when the stack has zero or one element.

This style of stack cannot contain the same node more than once lest a loop occur in the graph. A more general stack push routine might create a new node, inserting it just after the header node, and add a new edge emanating from this newly-created node pointing to the node being pushed. In order that the nodes being pushed be distinguishable from the new nodes used for maintaining the list structure, each of the new "cons-cell-like" nodes would have to be marked, perhaps by having another distinguished node point to it.

```
(defun gpush (x head Stk)
   (when (not (pxapply ((x p)()(y) ((   y x *) (   y x *)
                                     (p  1 * *) (p  0 1 =)
                                     (x  * * *) (x  1 = =)
                                     (*  * * *) (*  = = =))) Stk (x head) ()))
         (pxapply ((x p)()() ((   x *) (   x *)
                              (p  * *) (p  1 =)
                              (x  * *) (x  = =)
                              (*  * *) (*  = =))) Stk (x head) ())))


(defun gpop (head Stk)
   (let ((x nil))
      (when (not (pxapply ((p)(x)(y) ((   x y *) (   x y *)
                                      (p  1 * *) (p  0 1 =)
                                      (x  * 1 *) (x  = 0 =)
                                      (*  * * *) (*  = = =))) Stk (head) (x)))
            (pxapply ((p)(x)() ((   x *) (   x *)
                                (p  1 *) (p  0 =)
                                (x  * *) (x  = =)
                                (*  * *) (*  = =))) Stk (head) (x)))
      x))
```

Figure 5.18: Functions to maintain a stack

Since the only use made of stacks in this dissertation will be for traversing a graph, the simpler version in Figure 5.18 will suffice.

5.5.3 Binary trees

The simplest way to implement a binary tree using graphs is to use two graphs named L and R. L contains those tree edges that point to left children, while R represents those edges that point to right children. This strategy adequately solves the problem of telling which edges point in which direction, but has the disadvantage that pattern-matches on the tree must be decomposed so as to be applied to the pair of graphs.

It is more satisfying to represent a tree with a single graph. Such a representation can be achieved by taking the two trees L and R, and forming their union, T. A node-order graph may be constructed by traversing the tree (using L and R) in, say, top-down left-to-right order. This node order, in conjunction with semantic order constraints, may then be used to match patterns against T.

5.5.4 Graphs with typed nodes

It is sometimes useful to be able to distinguish different classes of nodes from one another. For example, if we have a graph representing an expression tree, we may wish to know which nodes represent operators, which represent variables, and which represent constants. We may wish to create and maintain this kind of type information for each node in the graph. To "assign" types to nodes, we create one distinguished node per type, and add edges to our graph pointing from this special node to each graph node having that type. The type of any node may then be checked in a pattern by feeding the pattern the appropriate special node as an *in* variable, adding a row labeled by it, and placing 1's in this row in columns labeled by variables whose type we are checking. The pattern can then only match if the desired typing arcs are present.

This node-typing technique gives yet another way to implement binary trees: create two type nodes, say, l and r, and have them point to all the left and right children, respectively.

5.6 Computing node orders

We now have enough tools at our disposal to compute node orders. Recall that to build a node order for a graph G, we need to construct a linear graph from the nodes of G in the order we want, and hand this resulting graph to the primitive function `node-order`. This section describes an algorithm to compute one very useful order, that resulting from *s-numbering*. It is shown in Figure 5.19.

```
(defun s-number (G n0)
   (let ((Mark (make-graph)) (S (make-graph)) (y nil))
      (labels ((dfssub (n)
                  (when (not (set-member? n Mark))
                        (add-to-set! n Mark)
                        (foreachchild (c n G) (dfssub c))
                        (gpush n S))))
            (dfssub n0))
      (pxapply ((x)()(y) ((  y *) (  y *)
                          (x 1 *) (x 0 =)
                          (* * *) (* = =))) S (StackHead) ())
      S))
```

Figure 5.19: Graph traverser

This algorithm assumes that the input graph G is a flowgraph, and that `n0` is its unique entry node. The local function `dfssub` performs a depth-first search on the flowgraph, starting at `n0`. As it backs-up from each node, it pushes that node onto the stack `S`. It terminates when there are no more reachable nodes to visit. The `pxapply` is then executed; it removes the stack header node from the stack, leaving a graph in the form of a linear list. This may them be given as a parameter to the built-in function `node-order` to create the node-order object that may be used in subsequent matches.

5.7 Transitive Closure

As a prelude to the program flow analysis algorithms discussed in the next chapter, we close this chapter with a discussion of a simpler graph problem, that of computing transitive closures.

5.7.1 Warshall's algorithm

Perhaps the most widely known and used algorithm for computing transitive closure is Warshall's algorithm, an $\mathrm{O(n^3)}$ algorithm that was designed for computing with relations represented as boolean matrices [23]. The biggest drawback to using Warshall's algorithm is that it is necessarily $\mathrm{O(n^3)}$, since it operates on a dense representation of a relation. One can do better with a sparse representation such as that afforded by graphs.

5.7.2 Hunt, Szymanski, and Ullman's algorithm

Hunt, Szymanski, and Ullman [17] developed an algorithm using a sparse representation that runs in $\mathrm{O}(n^2)$ time on sparse relations (those having $\mathrm{O}(|E|) = \mathrm{O}(|V|)$). Their algorithm, shown in Figure 5.20, works by using a simple node mark-and-stack path-finding algorithm (`hsu-search`) to compute, for a given node, all nodes reachable from that node. Whenever it finds a new reachable node, it adds an edge from the starting node to this new node into the graph representing the closure. Wrapped around this path-finder is a loop that iterates over each node in the graph. Thus, the result contains an edge from each node to every node reachable from that node via some path in the original graph. The algorithm's behavior is $\mathrm{O(|V| \cdot |E|^2)}$. It degenerates to $\mathrm{O(|V|^3)}$ in the dense case, but gives $\mathrm{O(|V|^2)}$ for sparse graphs.

```
(defun hsu (G)
   (let ((H (make-graph)))
      (foreachnode (p G)
         (hsu-search G p H))
      H))


(defun hsu-search (G p H)
   (let ((Stk (make-graph)) (MarkSet (make-graph)))
      (gpush p Stk)
      (while (not (empty-graph Stk))
         (let* ((k (gpop Stk))
                (C (comb G k)))
            (foreachedge ((kk q) C)
                (when (not (is-element? q MarkSet))
                   (add-to-set! q MarkSet)
                   (gpush q Stk)
                   (+!edge p q H)))))))
```

Figure 5.20: The Hunt Szymanski and Ullman algorithm

The version of the algorithm shown uses a stack represented by the graph `Stk`, and "marks" visited nodes by placing them into the `MarkSet` set. Between iterations, all nodes become unmarked by the simple expedient of assigning a new empty graph to `MarkSet`.

5.7.3 A new algorithm

A simple modification makes this algorithm even faster (though only by a constant factor). Examining Hunt, Szymanski and Ullman's algorithm, we note that, if n is some node in G, many searches may pass through n, even after H already encodes all nodes reachable from n. Accordingly, we can modify `hsu-search` to keep track of every node the closure from which has already been computed, and to copy that information when the search reaches such a node instead of searching through it. The modified algorithm is shown in Figure 5.21.

While this algorithm has the same asymptotic time bounds as does the Hunt, Szymanski and Ullman algorithm, experimentation has shown it to be 10% to 40% faster than that algorithm. The increase in algorithmic complexity seems small compared to the efficiency achieved, especially when computing closures of large relations.

```
(defun transitive-closure (G)
   (let ((H (make-graph)) (Known (make-graph)))
      (foreachnode (p G)
         (tc-search G p H Known)
         (add-to-set! p Known))
      H))


(defun tc-search (G p H Known)
   (let ((Stk (make-graph)) (MarkSet (make-graph)))
      (gpush p Stk)
      (while (not (empty-graph Stk))
           (let ((k (gpop Stk)))
                  (if (is-element k Known)
                      (let ((C (comb H k)))
                         (foreachedge ((zz q) C)
                            (+!edge p q H)
                            (add-to-set! q MarkSet)))
                      (let ((C (comb G k)))
                         (foreachedge ((zz q) C)
                            (when (not (is-element? q MarkSet))
                                  (+!edge p q H)
                                  (add-to-set! q MarkSet)
                                  (gpush q Stk))))))))
```

Figure 5.21: Faster transitive closure algorithm

Chapter 6

Data-flow Analysis Revisited

As was stated in Chapter 1, the goal of this dissertation is to show that a language based on graph transformations is sufficient for implementing most if not all known program flow analysis algorithms, and that its use reduces to a manageable level the complexity of combining multiple such algorithms. A family of graph transformations was developed in Chapters 2 and 3, and a language based on these transformations was exhibited in Chapter 4 and extended in Chapter 5. The claim is now made that this language is a suitable one for attaining our goal.

This chapter demonstrates this point by developing and presenting implementations in this language for several flow analysis algorithms, showing how easily they may be used and combined. The algorithms have been chosen to cover as wide a range of implementation techniques as possible in order to show the versatility of our graph-based approach.

6.1 A simple approach

This section exhibits a very simple algorithm that computes a solution to the "reaching definitions" problem. This problem is to compute, for each variable reference, the definitions of (assignments to) that variable that reach the reference along some path in the flowgraph. The standard approach is to associate with each graph node n a set called gen_n containing the definitions contained within n that reach the exit of n, and another set called $kill_n$ that contains definitions that become invalidated when the code in n is executed.

The standard algorithm then computes sets in_n and out_n for each n, satisfying

$$in_n = \bigcup_{p \in pred(n)} out_p$$

and

$$out_n = in_n - kill_n \cup gen_n.$$

This computation is done by taking the initial approximation $in_n = out_n = \emptyset$ and iteratively evaluating the right-hand-sides of the above equations until the in_n and out_n sets cease to grow. The resulting collection of in_n and out_n sets is called the *minimal fixed-point solution* of the data-flow problem.

To implement such an algorithm, we require a means of representing the *kill*, *gen*, *in* and *out* sets and some way of evaluating the data flow expressions.

Our approach uses graphs to represent *associations*. In general, a graph G will associate a node b with node a if and only if G contains an edge from a to b. If N is the set of nodes in a program's flowgraph and U is a set of nodes corresponding to the variable definitions in the program, then a graph G associates a subset W of U with a node n in N if and only if it contains, for each w in W, an edge leaving n and entering w.

Accordingly, we represent *in*, *out*, *gen* and *kill* sets with graphs that associate definitions with flowgraph nodes. We construct *gen* and *kill* graphs exactly as indicated by the previous paragraph: the graph *gen* will contain an edge $n \rightarrow d$ if and only if the definition corresponding to node d is considered to be in the *gen* set of n, and the graph *kill* will contain an edge $n \rightarrow d$ if and only if the definition corresponding to node d is considered to be in the *kill* set of n. These two graphs thus represent *all* of the usual *gen* and *kill* sets simultaneously, the individual definitions associated with each node being accessible in that graph via edges leaving that node.

The construction of the *gen* and *kill* graphs is easy. We start with several graphs that are directly derivable from some static representation of the program to be analyzed, such as its abstract syntax tree. Let N be a set of flowgraph nodes, let V be a set of nodes that correspond to the variables in the program, and let D be a set of nodes that correspond to the definitions (assignments) to be tracked. We create the following graphs at control-flow analysis time:

ND contains an arc from node $n \in N$ to node $d \in D$ if and only if the definition d occurs in node n.

VD contains an arc from node $v \in V$ to node $d \in D$ if and only if the variable v is the one being defined in definition d.

DV is the reverse of graph *VD*, and so contains an arc from each definition to the variable to which that definition assigns a value.

Each of these three graphs encodes purely local information about nodes in the program.[1]

The graph encoding *gen* information is then simply *ND*, which tells which definitions are generated at each flowgraph node. The definition of *kill* is a bit more involved, being the composition $ND \circ DV \circ VD$. Recall that *ND* has arcs from each node to all definitions in that node. $ND \circ DV$, therefore, has arcs from each node to all variables defined in that node. Then, $(ND \circ DV) \circ VD$ has arcs from each node to all definitions of variables defined in that node; this is the definition of *kill*.[2]

Once the *gen* and *kill* graphs have been constructed, the data flow analysis problem can be solved by the algorithm shown in Figure 6.1.

```
(defun Reach (G gen kill)
   (let ((in (make-graph)) (out (make-graph)) (Gr (reversegraph G)))
      (setf (graph-changed out) t)
      (while (graph-changed out)
         (setf (graph-changed out) nil)
         (setf in (compose Gr out))
         (union! out (union (difference in kill) gen)))
      (values out in)))
```

Figure 6.1: A simple algorithm for reaching definitions

Each iteration, this algorithm recalculates the entire *in* graph by composing the reverse of G with *out*. Consider: an edge $n \to m$ is in the reverse of G if and only if m is a predecessor of n in G. An edge $m \to d$ is in *out* if and only if the definition corresponding to node d has been found, on some previous iteration, to reach the exit of flowgraph node m. Therefore, an edge $n \to d$ is in their composition if and only if d represents a definition reaching the exit of some predecessor of n. But this set of definitions is composed of exactly those definitions reaching the entry of n.

[1] For ease in presentation, we assume that there is at most one definition in each flowgraph node. When more are present, the definition of *ND* changes slightly, and requires analysis of each basic block in the program in order that it encode only definitions that are live at the end of that block.

[2] *Kill* may also be defined as $(ND \circ DV \circ VD) - ND$, which has arcs from each node to all definitions of variables defined in that node that aren't themselves in that node. Since *gen* is the same as *ND*, this difference is a moot one, and so we have elected to define *kill* as the version that requires less computation.

The new value of the *out* graph is calculated by the expression $in - kill \cup gen$. Since the *out* graph can only grow, the result of this computation is added to the pre-existing *out* graph, causing the latter to change only if the new value is unequal to the old value. Since the looping condition depends on *out*'s changing, the algorithm will terminate once the *out* graph stabilizes.

Although this algorithm computes the correct result graphs, one would not want to use it in practice because of its extreme inefficiency. Each iteration, it re-computes all of the *in* and *out* graphs even if only one edge needs to be added. It would be far more efficient to compute only the parts of the *in* and *out* graphs that change each iteration rather than the entire graphs. The next section implements exactly this improvement.

6.2 The worklisting approach

The simplest, and undoubtedly the most frequently implemented data-flow analysis algorithms are based on a *worklist*. A worklist is a set of flowgraph nodes at which the *in* and *out* information is known to need (re-)computation. It is typically initialized to contain only the flowgraph's start node. The algorithm repeatedly removes a node from the worklist, evaluates that node's *in* set in terms of the *out* sets of its flowgraph predecessors, and computes a new *out* set using the formula $out := in - kill \cup gen$. Should the node's *out* information change, the algorithm puts all of its successors into the worklist since their *in* (and probably their *out*) sets are known to need updating. The algorithm continues in this fashion until the worklist is empty.

All four of the classical intraprocedural data-flow analysis algorithms, reaching definitions, available expressions, live variables and very busy expressions, may be solved in this way. The only differences lie in the way the *in* and *out* sets are initialized, the way *out* information is propagated to *in* sets, and whether the algorithm works on the flowgraph or its reverse.

6.2.1 A worklisting iterator

Rather than implement this standard algorithm each time we need it, we take an alternative approach made possible by our language. This approach is the construction of a general worklist-based iterator that can be wrapped around whatever special evaluation code is needed for the data-flow problem at hand. This iterator is shown in Figure 6.2.

```
(defmacro foreach-node-with-worklist ((x G n0 H) &body body)
    (let ((W (gensym)) (Visited (gensym)) (p (gensym)))
        `(let ((,x nil) (,W (make-graph)) (,Visited (make-graph)))
              (add-to-set! ,n0 ,W)
              (while (not (emptygraph ,W))
                     (pxapply ((w)(x)() ((   x *) (   x *)
                                         (w  1 *) (w  0 =)
                                         (*  * *) (*  = =))) ,W (SetHead) (,x))
                     (setf (graph-changed ,H) nil)
                     ,@body
                     (when (or (graph-changed ,H)
                               (not (set-member? ,x ,Visited)))
                           (add-to-set! ,x ,Visited)
                           (foreachchild (,q ,x ,G)
                                 (add-to-set! ,q ,W)))))))
```

Figure 6.2: Worklisting iterator

Here, `G` is the flowgraph with start node `n0` and `H` is a graph whose changing will indicate that the worklist needs updating. `H` will usually be the graph representing the *out* sets in the data-flow problem. The variable `x` is the iteration variable. It is successively bound to each node removed from the worklist until the worklist is empty. Since the algorithm adds nodes to the worklist, `x` may well be bound to the same node several times during the algorithm. At each execution of the body, however, `x` will be bound to a node at which it is known the *out* graph needs updating.

The graph `W` is the worklist, and is a set that is initialized to contain only the start node. An auxiliary set, `Visited`, is maintained to ensure that every node reachable from the start node is bound to `x` at least once. The use of this set makes sure that every reachable node has an opportunity to have its data-flow function evaluated at least once.

6.2.2 A worklisting example

An example of this iterator's use is the algorithm in Figure 6.3, that again computes reaching definitions.

Let `x` be some node taken from the worklist. Each edge in the graph `(rcomb G x)` leaves `x` and enters one of `x`'s predecessors in the flowgraph. When we compose this graph with *out*, we get a graph associating with `x` all of the definitions associated with each of `x`'s predecessors, that is, the propagation of `x`'s predecessors' *out*s to `x`. This graph is exactly the comb of the desired *in* graph at `x`. Accordingly, we use the graph union operator to

```
(defun Reach (G n0 gen kill)
   (let ((in (make-graph))(out (make-graph)))
       (foreach-node-with-worklist (x G n0 out)
          (let ((new-ins (compose (rcomb G x) out)))
             (union! in new-ins)
             (union! out (union (difference new-ins kill) (comb gen x)))))
       out))
```

Figure 6.3: Reaching definitions via worklist

add all of its edges to the *in* graph.

To calculate the new *out* information corresponding to this new *in* information, we need to apply *gen* and *kill*. We perform this application by subtracting those edges that exist in the *kill* graph, then adding those edges that lie in the comb of the *gen* graph at `x`. The result of this procedure is the comb of the desired *out* graph at `x`, which is then merged into *out*. Recall that the worklisting iterator monitors the *out* graph for changes. If this last union operation changes the *out* graph, then all of `x`'s successors are automatically added to the worklist.

We have thus shown the ability of our graph-based language to solve the classical data-flow analysis problems with the classical algorithm. Better still, we have done it with a single generic iterator that can be used to solve multiple problems. Notice also that the worklist iterator's use is not dependent on the use of *gen* and *kill* sets to represent local data-flow functions. The actual propagation of information is done in the body of the loop, which can be customized to propagate information using any method a programmer sees fit to use. Its use is thus not restricted to any particular data-flow framework (fast, distributive, monotone, etc.). Note also that the set of facts being propagated does not need to be determined before propagation begins, as new graph nodes may be created and added to a graph at any time. We shall see an example of this sort of behavior in Section 6.6.

An improved worklisting algorithm

The use of a graph to represent a worklist is a rather trivial use of the representational power of graphs. It is natural to wonder if the capabilities we have available can be used to improve the algorithm.

One obvious weakness in the usual worklisting algorithm is that it may perform

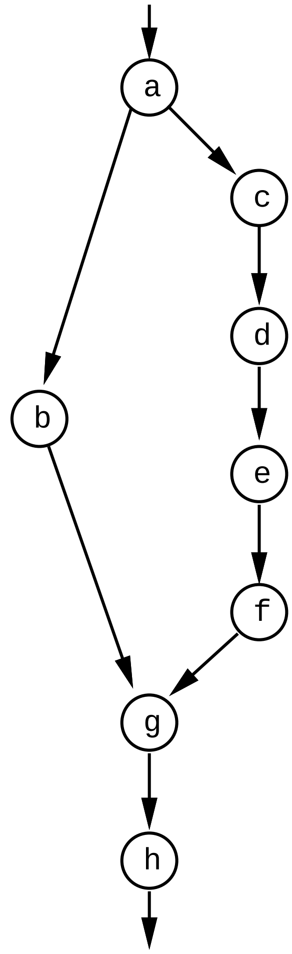


Figure 6.4: A section of a flowgraph

evaluations at many nodes more often than is necessary. Take, for example, the flow graph shown in Figure 6.4 and assume the worklist operates queue-fashion, except that the addition of a node already in the worklist has no effect. When a non-trivial evaluation happens at node a, nodes b and c are put into the worklist. We then see the following sequence of evaluations:

1. Node b is evaluated, putting node g into the worklist.
2. Node c is evaluated, putting node d into the worklist.
3. Node g is evaluated, putting node h into the worklist. $(*)$
4. Node d is evaluated, putting node e into the worklist.
5. Node h is evaluated, putting node i into the worklist. $(*)$

6. Node e is evaluated, putting node f into the worklist.
7. Node i is evaluated, putting node j into the worklist. $(*)$
8. Node f is evaluated, putting node g into the worklist.
9. Node j is evaluated, putting node k into the worklist. $(*)$
10. Node g is re-evaluated, putting node h into the worklist.

The sequence of evaluations continues, with the re-evaluation of h and its descendants interleaved with evaluations of nodes farther "down" in the graph. Notice that the starred evaluations, that is, the first evaluation at node g and the evaluations triggered thereby, are all unnecessary, since each of these nodes is re-evaluated after the evaluation at node f. This sort of sequence of redundant computations can be seen to occur whenever the two branches of an *if-then-else* construct are of different lengths. Our goal is to improve this situation by deferring the evaluation of each confluence node such as g until all of its predecessors have been evaluated. This deferral will result in a more efficient order of evaluation by eliminating the redundant evaluations at this node and its descendants.

We implement our improved algorithm by adding a bit more structure to the worklist, now termed a *workgraph*. The standard algorithm adds a node to the worklist graph by simply adding an edge to it from the designated set header node. The new algorithm does this too, but can add other edges to the workgraph. Let the node being added be n, and let e be the flowgraph edge the propagation along which is causing n to be added to the workgraph. Whenever n is not already in the workgraph, all flowgraph edges entering n other than e will be copied into the workgraph. These edges are added regardless of whether or not any of the new node's predecessors already appear in the workgraph. These additional edges are used to represent evaluation-order dependencies. We will say a node in the workgraph is *blocked* if more than one workgraph edge enters it, otherwise it is *unblocked.*

The addition of edges to the workgraph only represents half of the job of the workgraph iterator. The other half is the strategy for choosing that workgraph node that is the best candidate for evaluation. The strategy used here is simple: an unblocked node is chosen if any exist, otherwise the algorithm chooses any blocked node. All edges entering or leaving the chosen node are then removed from the workgraph.

In the example in Figure 6.4, a would be evaluated, placing b and c in the workgraph. When b was then evaluated, it would place g in the graph, blocked by the edge $f \rightarrow g$. Evaluation would then proceed along the path c, d, and e, until f was evaluated, unblocking g by the removal of the edge $f \rightarrow g$. Only then would the node h and its descendants be evaluated. The evaluation of g can be seen to have been deferred until all of its predecessors' *out* information became available. This deferral does away with the unnecessary first evaluation not only of g, but of each of its descendants.

The choice of an unblocked node is always a "good" one, as it is known that new information has reached that node from each of its predecessors. It is not necessarily the case, however, that there always exists an unblocked node. For example, if the evaluation of node e above did not result in e's *out* information's changing, then node f would not be placed in the workgraph, would not be evaluated, and so could not unblock g.

If every node in the workgraph is blocked, then each such node is a confluence of two or more edges in the flowgraph, and information has not propagated along all of these edges. There are exactly two causes of this situation: propagation along one path to a node can "die out" as in the previous paragraph, or it may be blocked along some path to the node by some set of blocked nodes (possibly including the node in question). Nodes of the first of these two types are equivalent to unblocked nodes (thought they cannot be detected), and so their evaluation cannot lead to redundant computations. Nodes of the second type are, unfortunately, indistinguishable from nodes of the first type. Their evaluation can lead to redundancies, but certainly no worse than those occasioned by the original algorithm.

There is no additional overhead associated with maintaining a workgraph instead of a worklist due to its consideration of all flowgraph edges, since the original algorithm also needs to examine each edge at least once when placing nodes into the worklist. The worst-case complexity of the new algorithm is therefore no different from the old.

The completed algorithm is shown in Figure 6.5. Note that it is functionally equivalent to the original algorithm in Figure 6.2, in that a data-flow analysis algorithm using it will obtain the same result, though by considering graph nodes in a different order. The new iterator may be substituted for the old one by simply replacing the old iterator macro name with the new one in any data-flow analysis algorithm making use of the original. This is an instance where our graph-transformation based language allows us to change basic implementations without affecting the behavior of the application.

6.3 A dual approach

Many data-flow analysis algorithms require that local data-flow functions such as those encoded by *gen* and *kill* sets be attached to a flowgraph's edges rather than to its nodes. We will see examples later where this arrangement has advantages over the more traditional approach. For now, we simply modify the *Reach* example given earlier in this chapter to make use of this organization.

We still wish to attach our results to flowgraph nodes, since we want to know which facts hold there at the end of the analysis. Note that a simplification of result representation ensues when local data-flow functions are attached to flowgraph edges rather than to nodes. Since these local functions are applied when information is propagated along an edge rather than when it reaches a node, there is nothing to do to compute *out* information from *in*

```
(defmacro foreach-node-with-workgraph ((x G n0 H) &body body)
   (let ((C (gensym)) (W (gensym)) (W2 (gensym)) (Visited (gensym))
         (y (gensym)) (p (gensym)) (q (gensym)))
      `(let ((,x nil)
             (,W (make-graph)) (,W2 (make-graph)) (,Visited (make-graph)))
          (add-to-set! ,n0 ,W)
          (while (or (not (empty-graph ,W)) (not (empty-graph ,W2)))
             (unless
              (pxapply ((w)(x)() ((   x *) (   x *)
                                  (w  1 *) (w  0 =)
                                  (x  * *) (x  0 0)
                                  (*  * *) (*  = =))) ,W (SetHead) (,x))
              (pxapply ((w)(x)() ((   x *) (   x *)
                                  (w  1 *) (w  0 =)
                                  (x  * *) (x  0 0)
                                  (*  * *) (*  0 =))) ,W2 (SetHead) (,x)))
             (setf (graph-changed ,H) nil)
             ,@body
             (when (or (graph-changed ,H) (not (set-member? ,x ,Visited)))
                (progn
                   (add-to-set! ,x ,Visited)
                   (foreachchild (,y ,x ,G)
                      (let ((,C (rcomb ,G ,y)))
                           (-!edge ,y ,x ,C)
                           (if (empty-graph ,C)
                               (add-to-set! ,y ,W)
                               (progn
                                  (foreachedge ((,p ,q) ,C) (+!edge ,q ,p ,W2))
                                  (add-to-set! ,y ,W2))))))))))
```

Figure 6.5: A better worklisting iterator

information. As a result, there is then no difference between *in* and *out* information, and we accordingly dispense with the *in* and *out* notation, instead creating one graph, S, that associates with each node the information that is known to be true at that node. S will most closely correspond to the *in* graph in earlier examples.

The modified algorithm is shown in Figure 6.6. There are two differences from the previous version: the structure of the *gen* and *kill* graphs, and the way the information propagation is done. We deal with *gen* and *kill* first.

Recall the definition of `factor`, given in Chapter 5. By means of edgebindings, it associates a unique node with each edge in the graph being factored. The versions of the *gen* and *kill* graphs that we need for this new algorithm must associate facts with these auxiliary "edge" nodes rather than with the original graph nodes. Where the previous version would initially associate a *gen* or *kill* fact with node n, this version initially associates that fact with each edge leaving node n. It creates this association by factoring the flowgraph G to produce graphs G_1 and G_2, where $G = G_1 \circ G_2$. G_2 may then be composed on the left of the usual *gen* and *kill* graphs, giving *gen-e* $= G_2 \circ$ *gen* and *kill-e* $= G_2 \circ$ *kill*.

```
(defun Reach-edgeversion (G n0 gen-e kill-e)
   (let ((S (make-graph)))
       (foreach-node-with-worklist (x G n0 S)     ;; or ...-workgraph
           (let ((CR (reversegraph (rcomb G x))))
                (multiple-value-bind (g1 g2) (factor CR)
                     (let* ((rg1 (reversegraph g1))
                            (local-S (compose rg1 S))
                            (localizer (compose rg1 g1))
                            (local-gen (compose localizer gen-e)))
                           (union! S (compose (reversegraph g2)
                                              (union
                                                 (difference local-S kill-e)
                                                 local-gen)))))))
       S))
```

Figure 6.6: Reaching definitions, information attached to edges

The propagation part is only slightly different from the way it was in the earlier algorithm, when we account for the fact that we are now only maintaining one set per node. To calculate S at node `x`, we take the comb of S at each predecessor p of `x` in G and propagate it along the edge leading from p to `x`, applying *gen-e* and *kill-e* as we do so. The graphs `g1` and `g2` and their reverses are used only to obtain the appropriate *gen* and *kill* information, which is now attached to edges instead of nodes.

6.4 Interval analysis

Interval analysis algorithms are, of course, more complicated than iterative algorithms, but are asymptotically faster. This section gives an algorithm for one of them, that developed by Graham and Wegman [13], [24]. The previous section introduced the concept of associating information with flowgraph edges instead of nodes; that approach will be used here.

The algorithm works by converting a data-flow analysis problem into an equivalent but smaller problem. It does this conversion by shrinking the flowgraph in a well-defined way, adjusting the local data-flow information attached to edges so that a solution of the resulting problem, in conjunction with information describing the shrinkage, can be transformed to a solution of the original problem. This shrinking process is performed repeatedly until it is no longer possible. When the original flowgraph is reducible, the process as described here results in an acyclic comb graph.[3] The (now trivial) data-flow problem on the comb graph is then easily solved, and the data-flow information about all nodes that have been "removed" from the graph is deduced from this solution.

6.4.1 $\mathbf{T}_1'$ and $\mathbf{T}_2'$

The heart of the algorithm is the pair of transformations T_1' and T_2'. These transformations are described in [13] and [24].[4] Briefly, T_1' removes a looping edge $a \to a$ from a graph if there is exactly one other edge entering node a. The transformation T_2' is depicted in Figure 3.1 in Chapter 3. If there are nodes a, b, and c and edges $a \to b$ and $b \to c$, and if no other edge enters b then the transformation adds the edge $a \to c$ to the graph and deletes the edge $b \to c$. If there are no other edges leaving b, it deletes the edge $a \to b$ as well.

The algorithm uses a strategy for applying these transformations that ensures that the number of applications is linear in the number of edges in the flowgraph. This strategy, roughly speaking, is to apply transformations at the entry node of an innermost loop until that loop disappears. To be more specific, the loop entry node it chooses is the largest

[3] Graham and Wegman use a graph called a *fan graph*, wherein all non-looping edges emanate from a single node. Since our version produces an acyclic graph, and since there is no difference between an acyclic fan graph and an acyclic comb graph, we use the term *comb* here for consistency with earlier chapters.

[4] Graham and Wegman also describe a third transformation, T_3, used for reducing their fan graph to a single node. The approach given here has no need of it, and so it is omitted.

s-numbered graph node that is pointed-to by a frond. Thus, the loop entry nodes may be located easily by s-numbering the nodes in the graph, then searching for edges $a \rightarrow b$ where the s-number of b is less than the s-number of a. This entry-node identification is done at the outset of the algorithm, and a set of these nodes is formed. Each time a new header node is needed, the largest s-numbered unprocessed graph node in this set is chosen. The algorithm then applies T_1' and T_2' transformations until that node no longer heads a loop, selects the next largest s-numbered loop header node and continues.

Figure 6.7 is the skeleton of the algorithm that applies these transformations to reduce the flowgraph to a comb. Fronds are first determined by a single pass through the graph, controlled by the `foreach` loop, and are copied into the `Fronds` graph. Each is then removed from this graph (in reverse s-number order), and processed by T_1' and T_2' transformations while possible. T_1' and T_2' transformations are performed on the graph entry node, `n0`, as a final step.

```
(defun graham-wegman (G n0)
   (let ((N-O (node-order (reversegraph (s-number G n0)))) (Fronds (make-graph)))
      (foreachedge ((x y) G)
         (when (pmatch ((x y)()() ((  y *)
                                   (x 1 *)
                                   (* * *)) (x < y)) G (x y) () N-O)
            (+!edge y x Fronds)))
      (while (not (empty-graph Fronds))
         (pxapply (()(h)(x) ((  h *) (  h *)
                             (x 1 *) (x 0 =)
                             (* * *) (* = =))) Fronds ()(h) N-O)
         (apply-T1 G h)
         (while (apply-T2 G h) (apply-T1 G h)))
      (apply-T1 G n0)
      (while (apply-T2 G n0) (apply-T1 G n0))))
```

Figure 6.7: Skeleton of Graham-Wegman algorithm

Like the algorithm in the previous section, the Graham-Wegman algorithm attaches local data-flow functions to flowgraph edges instead of to flowgraph nodes. We will use the same notation as was used in the previous algorithm, letting S be the graph associating data-flow information with flowgraph nodes.

Application of a T_1' or T_2' transformation to a flowgraph necessitates a recomputation of the data-flow functions attached to the affected flowgraph edges. This recomputation is so that the resulting information propagation problem is equivalent to the pre-transformed one in that it will yield the same answer. Let us examine the T_1' transformation first.

```
       u    v    *                    u    v    *
     --------------                 --------------
 u |   *   1.p   *               u |   =    =    =
 v |   *   1.q   *      ---->    v |   =    0    =
 * |   *    0    *               * |   =    =    =
```

Figure 6.8: T_1'

T_1' may be represented by a single one of our graph transformations, shown in Figure 6.8. Its action is simply to remove a looping edge from the flowgraph. The corresponding necessary changes to the data-flow functions nearby are more complicated. Let ℓ be the looping edge $n \to n$ at header node n, and let e be the lone other edge $w \to n$ entering n; these edges are bound to variables q and p, respectively, by the transformation in Figure 6.8. Let the local data-flow functions along edges e and ℓ be f_e and f_ℓ, respectively. If S_w is the set of all facts known to be true at node w, then the information that is known to be true at node n is $f_e(S_w)$ combined with $f_\ell^k(f_e(S_w)$ via the flowgraph confluence operator for all $k \geq 1$. We see that if we replace the function attached to the edge e with this new function when we remove edge ℓ, the data-flow solution at node n will remain unchanged.

This function is fairly easy to compute when the local functions are encoded *gen* and *kill*-style. As before, let us use the *Reach* example to illustrate; our confluence operator is then $\cup$. Since the *Reach* problem is *fast* in the sense of Graham and Wegman, our new function may be computed as $f_e(S_w) \cup f_\ell(f_e(S_w))$. Now, letting $kill_e$ and gen_e be the sets of definitions associated with edge e, $kill_\ell$ and gen_ℓ be the sets of definitions associated with edge ℓ, $f_e(x) = x - kill_e \cup gen_e$ and $f_\ell(x) = x - kill_\ell \cup gen_\ell$ where x is any set of definitions known to reach node w. Letting $f_{\ell e}$ be defined for the moment as $f_\ell \circ f_e$,

$$
\begin{aligned}
f_{\ell e}(x) &= f_\ell(f_e(x)) \\
&= ((x - kill_e) \cup gen_e) - kill_\ell \cup gen_\ell \\
&= (x - kill_e) - kill_\ell \cup (gen_e - kill_\ell) \cup gen_\ell \\
&= x - (kill_e \cup kill_\ell) \cup (gen_e - kill_\ell \cup gen_\ell)
\end{aligned}
$$

giving $kill_{\ell e} = kill_e \cup kill_\ell$ and $gen_{\ell e} = gen_e - kill_\ell \cup gen_\ell$, so

$$
\begin{aligned}
f_e(x) \cup f_{\ell e}(x) &= ((x - kill_e) \cup gen_e) \cup ((x - kill_{\ell e}) \cup gen_{\ell e}) \\
&= (x - kill_e) \cup (x - kill_{\ell e}) \cup (gen_e \cup gen_{\ell e})
\end{aligned}
$$

$$\begin{aligned} &= x - (kill_e \cap kill_{\ell e}) \cup (gen_e \cup gen_{\ell e}) \\ &= x - (kill_e \cap (kill_e \cup kill_\ell)) \cup (gen_e \cup (gen_e - kill_\ell) \cup gen_\ell) \\ &= x - kill_e \cup (gen_e \cup gen_\ell). \end{aligned}$$

The *kill* information associated with the new edge e is thus seen to be equal to the *kill* information that was already associated with e. The *gen* information at e must be augmented with the *gen* information that is associated with edge ℓ. Thus, we see we want to modify the *kill* graph by merely removing the node ℓ and all of its out-going edges. The *gen* graph, on the other hand, has the node ℓ and its out-going edges removed, but for each edge $\ell \to x$ that is removed, the edge $e \to x$ is added.

The T_1' transformation shown in Figure 6.11 binds the variables e and ℓ to the edges as discussed above. The transformations shown in Figure 6.9 may be used to modify the *gen* and *kill* graphs after a successful application of T_1' to the flowgraph. The *gen* graph is modified by the application of two transforms, while the *kill* graph is modified by the application of only one.

```
gen:          A    *                     A    *
            ---------                  ---------
        l |  1    0       ---->    l |  0    =
        * |  *    *                * |  =    =

              B    *                     B    *
            ---------                  ---------
        e |  *    *       ---->    e |  1    =
        * |  *    *                * |  =    =


kill:         C    *                     C    *
            ---------                  ---------
        l |  1    0       ---->    l |  0    =
        * |  *    *                * |  =    =
```

Figure 6.9: T_1' *gen* and *kill* modifiers

The finished `apply-T1` function is shown in Figure 6.10. The function is augmented with two parameters more (*gen* and *kill* graphs) than were shown in the skeleton of the Graham-Wegman algorithm shown in figure 6.7; this will be rectified below.

We wish to adopt the same strategy with T_2'. Recall that this transformation searches for graph nodes a, b, and c such that the edges $a \to b$ and $b \to c$ are in the graph, and so that no other edge enters node b. When these conditions are satisfied, the T_2'

```
(defun apply-T1 (G h gen kill)
   (let ((e nil) (l nil) (A nil))
      (when (pxapply ((v)(p q)(u) ((   v   *) (   v *)
                                   (u 1.p *) (u  = =)
                                   (v 1.q *) (v  0 =)
                                   (*  0  *) (*  = =))) G (h) (e l))
            (when (pxapply ((q)(A)() ((  A *) (  A *)
                                      (q 1 0) (q 0 =)
                                      (* * *) (* = =))) gen (l) (A))
                  (pxapply ((p A)()() ((  A *) (  A *)
                                       (p * *) (p 1 =)
                                       (* * *) (* = =))) gen (e A) ()))

            (pxapply ((q)()(B) ((   B *) (   B *)
                                (q  1 0) (q  0 =)
                                (*  * *) (*  = =))) kill (l) ()))))
```

Figure 6.10: Algorithm to apply a T_1' transformation

transformation adds the edge $a \to c$ to the graph and deletes the edge $b \to c$. In the case that there are no other edges emanating from node b, the transformation deletes the edge $a \to b$ as well.

There are, unfortunately, two factors operative that complicate matters: first, as just noted, the transformation does something slightly different when only one edge leaves the node b. Second, the edge $a \to c$ may or may not already exist in the flowgraph. While that edge's existence or non-existence does not affect the transformation of the flowgraph, it does make the computation of the new data-flow functions a bit more involved. We deal with these complications by resorting to a four-fold case analysis. The four cases are:

1. No other edge leaves b; $a \to c$ doesn't already exist.
2. Some other edge leaves b; $a \to c$ doesn't already exist.
3. No other edge leaves b; $a \to c$ already exists.
4. Some other edge leaves b; $a \to c$ already exists.

Each of these cases may be handled by a single one of our graph transformations. We denote them by the names T_{2a}, T_{2b}, T_{2c} and T_{2d}, respectively. The corresponding transformations are shown in Figure 6.11. It should be noted that in order to bind the edge $a \to c$ to a variable (in order to recompute *gen* and *kill* information), the first two of these transformations need to be followed by a (based) pattern "checking" for this newly-added edge. The latter two bind this (previously-existing) edge as a side-effect of the pattern match.

```
T2a:          u   v   w   *                    u   v   w   *
            ----------------                 ----------------
        u  |  *  1.p  0   *               u  |  =   0   1   =
        v  |  0   0  1.q  0               v  |  =   =   0   =
        w  |  *   0   *   *      ---->    w  |  =   =   =   =
        *  |  *   0   *   *               *  |  =   =   =   =


T2b:      u   v   w   x   *                    u   v   w   x   *
        --------------------                 --------------------
    u  |  *  1.p  0   *   *               u  |  =   =   1   =   =
    v  |  *   0  1.q  1   *               v  |  =   =   0   =   =
    w  |  *   0   *   *   *      ---->    w  |  =   =   =   =   =
    x  |  *   0   *   *   *               x  |  =   =   =   =   =
    *  |  *   0   *   *   *               *  |  =   =   =   =   =


T2c:          u   v   w   *                    u   v   w   *
            ----------------                 ----------------
        u  |  *  1.p 1.r  *               u  |  =   0   =   =
        v  |  0   0  1.q  0               v  |  =   =   0   =
        w  |  *   0   *   *      ---->    w  |  =   =   =   =
        *  |  *   0   *   *               *  |  =   =   =   =


T2d:      u   v   w   x   *                    u   v   w   x   *
        --------------------                 --------------------
    u  |  *  1.p 1.r  *   *               u  |  =   =   =   =   =
    v  |  *   0  1.q  1   *               v  |  =   =   0   =   =
    w  |  *   0   *   *   *      ---->    w  |  =   =   =   =   =
    x  |  *   0   *   *   *               x  |  =   =   =   =   =
    *  |  *   0   *   *   *               *  |  =   =   =   =   =
```

Figure 6.11: T_{2a}, T_{2b}, T_{2c} and T_{2d}

We will denote the edges $a \to b$, $b \to c$ and $a \to c$ by the names p, q and r. We need to (re-)compute the *gen* and *kill* information attached to the affected edges. As before, let f_e be the data-flow function attached to edge e that is defined to be $f_e(x) = x - kill_e \cup gen_e$, as before also defining $kill_e$ be the set of definitions associated with the edge e via the *kill* graph, and letting gen_e be the set of definitions associated with the edge e via the *gen* graph.

In each of the four cases, the edge q is deleted from the flowgraph. Accordingly, we will remove its *gen* and *kill* information from the *gen* and *kill* graphs. The edge p will require special treatment when it is removed (by a T_{2a} or T_{2c} transformation); we defer a discussion of this edge until later. Meanwhile, we deal with the edge r. Should it not already exist, the data-flow function that should be newly attached to it is $f_q(f_p)$, while if it does exist, the function already there (f_r) should be replaced with $f_r \cap f_q(f_p)$.

The first case yields, via a calculation similar in spirit to the one above, $f_r(x) = x - (kill_p \cup kill_q) \cup (gen_q \cup (gen_p - kill_q))$, while the second case gives $f_r(x) = x - (kill_r \cap (kill_p \cup kill_q)) \cup (gen_r \cup gen_q \cup (gen_p - kill_q))$. A difficulty is that $kill_q$ is needed in the computation of the new *gen*. We compute $gen_p - kill_q$ separately via a pair of combs, and union the result in afterward. The remainder can be done by the transformations shown in Figure 6.12, for the case where r does not already exist, or in Figure 6.13 for the case where it does. The transformations in Figure 6.13 are interesting in that they both use the pattern portion as a "demultiplexer", then use the replacement portion to implement a logic function of these nodes.

```
kill:      A    B    C    *                 A    B    C    *
          --------------------             --------------------
     p |  1    0    1    0             p |  =    =    =    =
     q |  0    1    1    0    ---->    q |  =    0    0    =
     r |  *    *    *    *             r |  1    1    1    =
     * |  *    *    *    *             * |  =    =    =    =


gen:                 D    *                 D    *
                    ---------              ---------
               q |  1    0    ---->    q |  0    =
               r |  *    *             r |  1    =
               * |  *    *             * |  =    =
```

Figure 6.12: T_{2a} and T_{2b} *gen* and *kill* modifiers

```
kill:       A B C D E F G *             A B C D E F G *
           -----------------           -----------------
         p | 1 0 0 1 0 1 1 0   ---->  p | = = = = = = = =
         q | 0 1 0 1 1 0 1 0          q | = 0 = 0 0 = 0 =
         r | 0 0 1 0 1 1 1 0          r | = = 0 = = = = =
         * | * * * * * * * *          * | = = = = = = = =

gen:        A B C D E F G *             A B C D E F G *
           -----------------           -----------------
         p | 1 0 0 1 0 1 1 0   ---->  p | = = = = = = = =
         q | 0 1 0 1 1 0 1 0          q | = 0 = 0 0 = 0 =
         r | 0 0 1 0 1 1 1 0          r | = 1 = 1 = = = =
         * | * * * * * * * *          * | = = = = = = = =
```

Figure 6.13: T_{2c} and T_{2d} *gen* and *kill* modifiers

6.4.2 $\mathbf{T}_{2a}$ and $\mathbf{T}_{2c}$: Moving edges to a piggybank

The T_{2a} and T_{2c} transformations present us with a different problem: they delete the node v (and thus the edge p). Regrettably, this edge and its attendant data-flow function are needed to determine the information that pertains at node v in the original flowgraph from the corresponding answer for node u.

Rather than get rid of this information, we create a "piggybank" graph to hold all such $u \to v$ edges deleted from the flowgraph by either T_{2a} or T_{2c}. Each such edge is entered into the piggybank at most once, since it may be deleted from the flowgraph at most once. When we remove such an edge from the flowgraph, we leave the *gen* and *kill* graphs alone, since the function associated with that edge remains valid.

No such deleted edge is ever a back-edge in the original flowgraph, since v is never a loop header dominating u. Thus, at the end of the reduction to a comb graph, the union of the piggybank graph and the final comb graph is a *tree* on which we may solve our information propagation problem quite simply and efficiently.[5]

We may now exhibit `apply-T2`; this is done in Figures 6.14 and 6.15.

[5] Equivalently, Graham and Wegman's T_3' transformation moves the comb graph's edges to the piggybank, forming the same tree.

```
(defun apply-T2 (G h gen kill piggybank)
   (let ((p nil)(q nil)(r nil)(w nil))
      (cond ((or (and (pxapply ((u)(p q w)(v)
                                ((    u   v   w  *) (   u  v  w  *)
                                 (u   *  1.p  0  *) (u  =  0  1  =)
                                 (v   0   0  1.q 0) (v  =  =  0  =)
                                 (w   *   0   *  *) (w  =  =  =  =)
                                 (*   *   0   *  *) (*  =  =  =  =))) G (h) (p q))
                      (pmatch ((u w)(r)() ((   w  *)
                                           (u 1.r *)
                                           (*  *  *))) G (h w) (r))
                      (pxapply ((p)()(u v) ((   v  *) (   v *)
                                            (u *:p *) (u  1 =)
                                            (*  *  *) (*  = =))) piggybank (p) ()))
                 (and (pxapply ((u)(p q)(v w x)
                                ((    u   v  w  x  *) (   u  v  w  x  *)
                                 (u   *  1.p *  *  *) (u  =  =  1  =  =)
                                 (v   *   0 1.q 1  *) (v  =  =  0  =  =)
                                 (w   *   0  *  *  *) (w  =  =  =  =  =)
                                 (x   *   0  *  *  *) (x  =  =  =  =  =)
                                 (*   *   0  *  *  *) (*  =  =  =  =  =))) G (h) (p q))
                      (pmatch ((u w)(r)() ((   w  *)
                                           (u 1.r *)
                                           (*  *  *))) G (u w) (r))))
             (T2-gen-n-kill-with-new-r gen kill p q r) t)

            ((or (and (pxapply ((u)(p q r)(v w)
                          ((    u   v   w  *) (   u  v  w  *)
                           (u   *  1.p 1.r *) (u  =  0  =  =)
                           (v   0   0  1.q 0) (v  =  =  0  =)
                           (w   *   0   *  *) (w  =  =  =  =)
                           (*   *   0   *  *) (*  =  =  =  =))) G (h) (p q r))
                      (pxapply ((p)()(u v) ((   v  *) (   v *)
                                            (u *:p *) (u  1 =)
                                            (*  *  *) (*  = =))) piggybank (p) ()))
                 (pxapply ((u)(p q r)(v w x)
                          ((    u   v   w  x  *) (   u  v  w  x  *)
                           (u   *  1.p 1.r *  *) (u  =  =  =  =  =)
                           (v   *   0  1.q 1  *) (v  =  =  0  =  =)
                           (w   *   0   *  *  *) (w  =  =  =  =  =)
                           (x   *   0   *  *  *) (x  =  =  =  =  =)
                           (*   *   0   *  *  *) (*  =  =  =  =  =))) G (h) (p q r)))
             (T2-gen-n-kill-with-preexisting-r gen kill p q r) t)
            (t nil))))
```

Figure 6.14: Algorithm for applying T_2' transformations

```
(defun T2-gen-n-kill-with-new-r (gen kill p q r)
   (let ((extra (comb gen p)))
      (foreachedge ((pp b) extra)
         (when (pmatch ((q b)()() ((  b *)
                                   (q 1 *)
                                   (* * *))) kill (q b)())
               (-!edge p b extra)))
      (pxapply ((p q r)()(A B C) ((  A B C *) (  A B C *)
                                  (p 1 0 1 0) (p = = = =)
                                  (q 0 1 1 0) (q = 0 0 =)
                                  (r * * * *) (r 1 1 1 =)
                                  (* * * * *) (* = = = =))) kill (p q r) ())
      (pxapply ((q r)()(D) ((  D *) (  D *)
                            (q 1 0) (q 0 =)
                            (r * *) (r 1 =)
                            (* * *) (* = =))) gen (q r) ())
      (union! gen extra)))
```

```
(defun T2-gen-n-kill-with-preexisting-r (gen kill p q r)
   (let ((extra (comb gen p)))
      (foreachedge ((pp b) extra)
         (when (pmatch ((q b)()() ((  q *)
                                   (b 1 *)
                                   (* * *))) kill (q b)())
               (-!edge p b extra)))
      (pxapply ((p q r)()(A B C D E F G)
               ((   A B C D E F G *) (   A B C D E F G *)
                (p  1 0 0 1 0 1 1 0) (p  = = = = = = = =)
                (q  0 1 0 1 1 0 1 0) (q  = 0 = 0 0 = 0 =)
                (r  0 0 1 0 1 1 1 0) (r  = = 0 = = = = =)
                (*  * * * * * * * *) (*  = = = = = = = =))) kill (p q r) ())
      (pxapply ((p q r)()(A B C D E F G)
               ((   A B C D E F G *) (   A B C D E F G *)
                (p  1 0 0 1 0 1 1 0) (p  = = = = = = = =)
                (q  0 1 0 1 1 0 1 0) (q  = 0 = 0 0 = 0 =)
                (r  0 0 1 0 1 1 1 0) (r  = 1 = 1 = = = =)
                (*  * * * * * * * *) (*  = = = = = = = =))) gen (p q r) ())
      (union! gen extra)))
```

Figure 6.15: Algorithms for T_2' *gen* and *kill* graphs

6.4.3 Putting it all together

```
(defun graham-wegman-reach (GG n0 gen-e kill-e)
   (let ((N-O (node-order (reversegraph (s-number GG n0))))
         (Fronds (make-graph)) (G (copy-graph GG))
         (piggybank (make-graph)) (h nil))
      (foreachedge ((x y) G)
         (when (pmatch ((x y)()() ((  y *)
                                    (x 1 *)
                                    (* * *)) (x < y)) G (x y) ())
            (+!edge y x Fronds)))
      (while (not (empty-graph Fronds))
         (pxapply (()(h)(x) ((  h *) (  h *)
                             (x 1 *) (x 0 =)
                             (* * *) (* = =))) Fronds ()(h) N-O)
         (apply-T1 G h gen-e kill-e)
         (while (apply-T2 G h piggybank gen-e kill-e)
                (apply-T1 G h gen-e kill-e)))
      (apply-T1 G n0 gen-e kill-e)
      (while (apply-T2 G n0 piggybank gen-e kill-e)
             (apply-T1 G n0 gen-e kill-e))

      ;; Now propagate over simplified graph:
      (Reach-edgeversion (union G piggybank) n0 gen-e kill-e)))
```

Figure 6.16: The Graham-Wegman algorithm

Figure 6.16 shows the entire algorithm. The first half is as before, except that parameters to `apply-T1` and `apply-T2` have been updated to agree with those functions. The second part propagates data-flow information over the tree resulting from the graph reduction process. The "Reach-edgeversion" algorithm developed above is used for this propagation. Since it propagates from the root of the tree outward, it visits each node in the tree exactly once.

Notice that all patterns used in `apply-T1` and `apply-T2` are either based, or are applied to combs that were generated from known nodes. This fact implies that both `apply-T1` and `apply-T2` are O(1) operations, which in turn implies that the asymptotic complexity of our implementation is the same as that of the implementation given in [13].

6.4.4 Data-flow problems other than *Reach*

The algorithm just displayed may appear to be fairly specialized for solving the reaching definitions problem. A quick look reveals, however, that everything is applicable to any forward data-flow problem on a reducible flow graph except the two functions `T2-gen-n-kill-with-new-r` and `T2-gen-n-kill-with-preexisting-r` and the `union!`

calculation at the end of the algorithm. By defining the transformations RT_1' and RT_2' presented in [13], one can easily modify this algorithm to solve backward data-flow problems on reducible flow graphs as well.

We have now shown that the graph-based approach is powerful enough to implement some of the most sophisticated data-flow analysis algorithms known, and without any loss of efficiency.

6.5 A mixed approach

The Graham-Wegman algorithm as described only works properly when given a reducible flowgraph as input, otherwise it does not reduce the flowgraph to a comb graph. The iterative algorithms are asymptotically less efficient but are relatively insensitive to the type of graph upon which they operate. We would like to have the best of both worlds at our disposal.

Notice that the last two sections featured algorithms that associated flow information with flowgraph edges rather than with nodes. In fact, the algorithm given in Figure 6.16 combines the iterative algorithm with the Graham-Wegman algorithm in order to propagate information over the final tree. Should this version of the Graham-Wegman algorithm be given a non-reducible flowgraph, it will shrink that flowgraph as much as possible, creating a simpler flowgraph. The graph resulting from the union of this reduced graph with the piggybank graph will not be a tree, but the iterative algorithm used will have no difficulty propagating information on this flowgraph. This mixed strategy should, in fact, be faster than using iteration alone, since the Graham-Wegman process will remove all of the single-entry loops from the program, and these account for most program loops. The resulting algorithm will thus approximate the speed of an elimination algorithm, while imposing no restrictions on the graphs on which it works.

Wegman [24] gave a different way of extending the Graham-Wegman algorithm to non-reducible flowgraphs. He provided more general versions of the T_1' and T_2' transformations that worked even when a header node was entered by multiple flowgraph edges. It should be fairly clear that our approach could be extended in the same way. What is interesting about the method of the previous paragraph is not so much that it extends the Graham-Wegman algorithm, but rather the way it does this extension – by combining two entirely dissimilar algorithms. The effort required to effect this combination without our graph-transformational approach would probably render such an undertaking infeasible. With our approach, it is simply a matter of six lines of code.

6.6 Constant Propagation

As a more complicated data-flow problem, we consider constant propagation. It requires nothing particularly different as a method used for propagating information, but has other implementational difficulties that render its use infrequent. In particular, it requires that the data-flow analyzer be able to evaluate arbitrary constant expressions in the language being compiled, and that it be able to compare constant values for equality (necessitating the knowledge of all built-in data types in the language being compiled).

This section will give methods for implementing four important constant propagation algorithms using the graph transformational approach: Kildall's algorithm, Reif and Lewis' improvement, and the algorithms of Wegman and Zadeck that, as a side-effect, improve the control-flow graph by pruning edges that can be proved never to be traversed.

All three of these algorithms work with a three-level lattice, whose top level contains the symbol $\top$, whose middle level contains all of the constants being considered, and whose bottom level contains the symbol $\perp$. They initially associate with each variable reference the symbol $\top$, and change these associations downward in the lattice when new values are discovered for variables. If a variable reference is associated with $\top$, no values have yet been discovered for that reference. If a variable reference is associated with a constant, then only that constant has been found to be a value of that reference of that variable. If a reference is associated with the symbol $\perp$, then two or more values have been discovered for that reference, meaning that the variable may not be considered to have a constant value at that location in the program.

We will have the constant propagator maintain this lattice in the form of a graph. A reference to a variable will be associated with some node in the lattice by having an edge explicitly point from a node uniquely associated with that reference to the appropriate lattice graph node. Special lattice nodes will be allocated to represent $\top$ and $\perp$. Intermediate-level nodes, each representing a different constant, will be added to the graph as constants are discovered.

Once these decisions have been made, we can propagate constants with some straightforward algorithm if we can deal with two problems: 1) evaluating expressions, and

2) performing meet operations (which requires deciding whether two constants are equal). A third problem is how the caller is to know the constant values and types that are associated with our lattice nodes.

The second problem is easy to solve: two constants are equal if their associated nodes are the same. The first problem may and must be solved by some other part of the compiler that then has the opportunity to maintain hash tables associating constant values with graph nodes, provided that it has access to the *new-node* facility for creating a new graph node whenever a new constant appears. This strategy can also solve the third problem. If we provide enough primitive operations for this external facility to build and access graphs, then the analyzer need know nothing about the source language, such as the interpretation of constants of various types or the evaluation of expressions. But we must provide these facilities anyway, so that the compiler may create initial graphs and interpret the answers of data-flow problems that we compute. So we need no special extensions in order to solve all three of these problems, other than a mechanism for calling routines external to the graph transformation language.

6.6.1 Kildall

Kildall's algorithm [18] propagates constants along the flowgraph. In other words, it maintains an instance of each variable at each flowgraph node, regardless of whether or not there is a reference to that variable there, in order that it may propagate all possible constants from one flowgraph node to the next. Our version will accept the graphs *ND*, *NU*, and DV as input. *ND* has an edge from each flowgraph node at which a definition occurs to a node representing that definition, while DV has an edge from each definition node to a node representing the variable defined at that node. *NU* has an edge from each flowgraph node to each node representing a variable used (accessed) at that flowgraph node. These graphs are easily produced at control-flow analysis time.

Let N be the set of flowgraph nodes, and let V be the set of nodes each of which uniquely represents some variable in the program. We can construct a graph (called NV) that contains the $|N| \times |V|$ edges from each $n \in N$ to each $v \in V$. Each of the edges in NV represents an instance of a variable at a node. Factoring this graph gives us graphs NI (nodes-to-instances) and IV (instances-to-variables). These intermediate nodes thus produced are our instances; we initialize our algorithm by creating a graph (IC) associating each variable instance node with some value in the constant graph, namely, $\top$. We also associate each definition with a constant (initially $\top$ too). This associated constant node represents the value assigned to the variable. Thus, in a statement such as `x := x + 1`, we have separate constant values being maintained for each of the two occurrences of `x`.

We next create a *propagation graph*, P, that contains edges paralleling the flowgraph, but whose nodes are the variable instances that we are keeping track of. For each variable v and each flowgraph edge $e = x \rightarrow y$, we add an edge e_v to this propagation graph where the tail of e_v is the instance of v associated with node x (via NI), and the head is the instance of v associated with node y (also via NI). The one exception is that if node x is a definition of variable v, then the tail of e_v is the definition node instead of the "use" node associated with x. This situation is determined with ND.

Construction of these initial conditions, including the construction of the propagation graph, is shown in Figure 6.17.

```
(defun kildall-init (G NV ND NU DV)
   (multiple-value-bind (NI IV) (factor NV)
      (let ((IC (make-graph)) (Lattice (make-graph)) (NV (make-graph))
            (top (new-node)) (bot (new-node))
            (true (new-node)) (false (new-node)))
         (+!edge top true Lattice) (+!edge top false Lattice)
         (+!edge true bot Lattice) (+!edge false bot Lattice)
         (foreachnode (n G)
            (foreachedge ((nn x) (comb NI n))
               (+!edge x top IC)))
         (foreachedge ((nn d) ND)
            (+!edge d top IC))
         (let ((P (intersection
                        (compose (compose (reversegraph NI) G) NI)
                        (compose IV (reversegraph IV))))
               (def2inst (intersection
                        (compose DN NI)
                        (compose DV (reversegraph IV)))))
            (foreachedge ((nn d) ND)
               (when (pmatch ((d)(a)() ((  a *)
                                        (a 1 *)
                                        (* * *))) def2inst (d) (a))
                     (-!edge a top P)
                     (+!edge d top P)))
            (let ((NR (intersection NI (compose NU (reversegraph IV)))))
               (values NI IV NR IC Lattice P top bot))))))
```

Figure 6.17: Initialization for Kildall's algorithm

There are then two parts to the algorithm: a *meet* operation at a node that propagates information from predecessor nodes, and an *evaluation* step that, if the node is a definition node, takes constant information collected in the meet operation, and evaluates the expression to produce a new value for the defined variable.

The meet step at node x uses the propagation graph to check the values of the instances of the predecessors of each variable, and applies Kildall's "lowering" operation on the lattice value associated with the current instance. An algorithm for this meet operation is shown in Figure 6.18. For each variable instance i, the function constructs the small comb graph V with i as the root and all values of the predecessors of i as the leaves. It then edits this graph as follows: if $\bot$ appears as any leaf, then all other leaves are removed. Otherwise, three transformations are performed in sequence: first, all $\top$'s are removed from V. Then, if there are two (or more) leaves, all leaves are removed and $\bot$ is inserted as a leaf. Finally, if the graph is empty, $\top$ is inserted as a leaf. This procedure guarantees that the graph will then contain exactly one edge, and that its leaf node will represent the meet of all values propagating to i. This edge is then inserted into the graph IC, replacing any old value if the new value is different.

```
(defun kildall-meet (x NI IC P top bot)
   (foreachedge ((xx i) (comb NI x))
      (let ((V (compose (rcomb P i) IC)) (w nil))
         (unless (pxapply ((i b)()() ((  b *) (  b *)
                                      (i 1 *) (i = 0)
                                      (* * *) (* 0 0))) V (i bot) ())
            (pxapply ((i t)()() ((  t *) (  t *)
                                 (i 1 *) (i 0 =)
                                 (* * *) (* = =))) V (i top) ())
            (pxapply ((i b)()() ((  x y b *) (  x y b *)
                                 (i 1 1 * *) (i 0 0 1 0)
                                 (* * * * *) (* * * * *))) V (i bot) ())
            (pxapply ((i t)()() ((  t *) (  t *)
                                 (i 0 0) (i 1 =)
                                 (* * *) (* = =))) V (i top) ()))
         (pmatch ((i)(w)() ((  w *)
                            (i 1 0)
                            (* * *))) V (i) (w))
         (pxapply ((i w)()() ((  w *) (  w *)
                              (i 0 *) (i 1 0)
                              (* * *) (* = =))) IC (i w) ())))))
```

Figure 6.18: Meet operation for Kildall's algorithm

The evaluation step at node x is trickier, as it depends on the assumptions that the rest of the compiler is maintaining a table of associations between nodes and constants, and is willing to evaluate expressions in light of the information stored there. An algorithm for doing the evaluation step under these assumptions is shown in Figure 6.19.

```
(defun kildall-eval (x NR IC IV ND top bot)
   (let ((XC (compose (comb NR x) IC)) (d nil))
      (when (pmatch ((x)(d)() ((  d *)
                               (x 1 *)
                               (* * *))) ND (x) (d))
         (if (pmatch ((x b)()() ((  b *)
                                 (x 1 *)
                                 (* * *))) XC (x bot) ())
             (pxapply ((d b)()() ((  b *) (  b *)
                                  (d 0 *) (d 1 0)
                                  (* * *) (* = =))) IC (d bot) ())
             (unless (pmatch ((x t)()() ((  t *)
                                         (x 1 *)
                                         (* * *))) XC (x top) ())
                (let ((VC (make-graph)))
                   (foreachedge ((xx y) (comb NR x))
                      (union! VC (compose (reversegraph (comb IV y)) IC)))
                   (let ((n (expression-eval x VC)))
                      (pxapply ((d n)()() ((  n *) (  n *)
                                           (d 0 *) (d 1 0)
                                           (* * *) (* = =))) RC (d n) ()))))))))
```

Figure 6.19: Evaluation operation for Kildall's algorithm

This function does an evaluation at node x if that node is a definition. It creates a small graph, XC, that collects together all of the nodes in the lattice graph that are associated with variables that are referenced by the expression at node x, and examines them. There are three possibilities: either 1) $\perp$ occurs in the set of constants, whereupon the answer is $\perp$, or 2) $\top$ occurs in the set of constants, whereupon the expression can not (yet) be evaluated, or 3) only constant values appear, meaning that each variable in the expression has associated with it exactly one constant value. Possibility (3) means that we can now evaluate the expression. We do this by creating a graph VC associating with each referenced variable the constant value associated with this instance of the variable. We then pass this association graph, along with the node x to some function that is in the compiler proper (in this case, `expression-eval`). It is responsible for finding the expression associated with node x and for looking up the values of each of the variables in the expression, using VC to associate variables with constant value nodes, and its own internal table to associate these nodes with real constant values. It should then evaluate the expression, and return a graph node (perhaps newly created) that is uniquely associated with the resulting constant value. The final transformation causes the definition node in the *IC* graph to refer to this returned value node if it does not already do so.

The entire algorithm may now be put together, and has been done so in Figure 6.20.

```
(defun kildall (G n0 NV ND NU DV)
  (multiple-value-bind (NI IV NR IC Lattice P top bot)
                       (kildall-init G NV ND NU DV)
    (foreach-node-with-workgraph (x G n0 IC)
       (setf (graph-changed IC) nil)
       (kildall-meet x NI IC P top bot)
       (when (graph-changed IC)
          (kildall-eval x NR IC IV ND top bot)))))
```

Figure 6.20: Kildall's algorithm

6.6.2 Reif & Lewis

Kildall's algorithm works, but is fairly inefficient because it propagates lots of information from each node to the next that isn't used there. Each variable has an instance at each node whether or not that variable is referenced or changed at that node. Reif and Lewis devised a more efficient way to propagate constants by propagating along a sparser graph, called a DefUseChain graph. In this new graph, arcs go from each definition of a variable to each use site of that definition. The use of this new graph implies that the only nodes that require reevaluation after a new value is calculated for a definition are those nodes that depend on that definition, not all the nodes in between them.

The algorithm is a simple change to Kildall's algorithm. Similar to the way we create the graph *NR* associating with nodes instances of variables that are referenced at those nodes, we can create graphs *RV* (referenced instances to variables), and *RC* (referenced instances to constants). We also need a graph *REACH* relating uses to definitions; this is what is calculated by the Reach algorithms discussed earlier in this chapter. We will substitute *NR*, *RV* and *RC* for their dense equivalents (*NI*, *IV* and *IC*) in Kildall's algorithm, and we will use the propagation graph

$$P = ((NR^R \circ REACH) \cap (RV \circ DV^R))^R$$

instead of the much denser version used above. This version has arcs only from definitions to referred-to instances that are reached by those definitions, and is, in fact, the DefUseChain graph referred-to by Reif and Lewis.

```
(defun reif-lewis (G NV ND NU DV REACH)
   (multiple-value-bind (NR RV RC Lattice P top bot)
                        (rl-init G NV ND NU DV)
      (let ((F (compose (compose ND P) (reversegraph NR))))
         (foreach-node-with-workgraph-init-all (x F RC)
            (setf (graph-changed RC) nil)
            (rl-meet x NR RC P top bot)
            (when (graph-changed RC)
               (rl-eval x NR RC RV ND top bot))))))
```

Figure 6.21: Reif and Lewis's algorithm

The other main difference, shown in the algorithm in Figure 6.21, is that a different graph (`F`) is used in place of the flowgraph `G`. The flowgraph associates with each node those nodes which may follow it in temporal sequence; this is not what we want here. Suppose n is a flowgraph node at which a definition of variable v occurs. The new graph associates with n each flowgraph node f such that a use of v occurs at f and the definition of v at n reaches f. In other words, F associates with each node n exactly those nodes at which re-evaluations need to take place whenever knowledge about a variable defined at n changes. F may not be a flowgraph, since it may not have a unique entry node. Accordingly, we use a slightly different iterator, one that puts all the nodes of F into the initial worklist. This makes sure that an evaluation occurs at each node at least once.

The Reif & Lewis example is important here not just because of its efficiency advantages over Kildall's algorithm, but because it demonstrates one of the main reasons for our graph-transformational approach: that multiple flow analysis algorithms may be combined easily. In the present case, the results of a reaching definitions analyzer were incorporated into the constant propagation algorithm simply by using the resulting graph. Consider the work necessary to do such an incorporation with the more conventional bit-vector approach. The propagation graph used in Reif & Lewis would have to be constructed by consulting the flowgraph, the bit-vectors resulting from the Reach analyzer, the mapping between bit-vector positions and definitions, the mapping between definitions and variables, the mapping between flowgraph nodes and variable references, and the mapping between variable definitions and variable references, *each of which would probably be maintained with a different data structure.* The use of the common graph-based system makes the construction of the propagation graph a single line of code rather than an algorithm whose size is perhaps larger than that of the propagation algorithm itself.

A more elaborate example of this phenomenon follows.

6.6.3 Wegman & Zadeck's constant propagation algorithms

In 1985, Wegman and Zadeck [25] published an efficient algorithm for constant propagation that takes expressions controlling conditional branches into consideration. If the algorithm determines that a branch can only go one way, it prunes the flowgraph at that point, removing the edges that can never be followed. Besides refining the flowgraph, this procedure can find more constants, since confounding values can not be generated by never-executed sections of code.

The presentation in their paper gave two algorithms, the second of which elaborated on the first to handle the flow of constant values across program segments better. We will take this same expositional approach here, as it leads to a clearer presentation.

First, we need extra information in order to use the algorithm. We already have a graph (ND) that can be used to determine whether or not a node is a definition site. We need a second graph, NQ that parallels ND, assigning a special node to each conditional branch node in order to maintain a value for the conditional expression. NQ will also be used to distinguish nodes representing conditional branches. Two other graphs will be used: TB, which contains conditional flowgraph arcs that are executed when a condition is true, and FB, which contains conditional flowgraph arcs that are executed when a condition is false. We'll also need two more special nodes, named *true* and *false*, so that we can recognize the direction a conditional branch will take from the value of the condition.

The algorithm, given a flowgraph `G` and a propagation graph `P`, builds a second flowgraph `G2` and propagation graph `P2` that are refinements of the inputs. `G2` contains no arcs that the algorithm can show never to be taken, and `P2` incorporates this improved control-flow information in the form of fewer dependencies over which to propagate.

The original algorithms used two worklists instead of one. This presentation follows that idea, but clarifies their use. One of these worklists is termed FTWL, meaning "first-timer's worklist". A node is entered into this worklist the first time it is found to be executable. The other worklist is termed CVWL, or "changed-value worklist". A node is entered into the CVWL whenever the value calculated at that node changes. Thus, the first time any node is considered it is entered into the FTWL. All subsequent times (at most two, since the lattice of constants has three levels) it is entered on the CVWL.

The algorithm is outlined in Figure 6.22. The initialization is essentially the same as in the Reif and Lewis algorithm, except that we are also generating the special nodes *true* and *false*, and presumably notifying the ambient compiler about their existence.

```
(defun ConditionalDef (G n0 NV ND NQ NU DV REACH TB FB)
   (multiple-value-bind (NR RV RC Lattice P top bot true false)
      (CD-init G NV ND NU DV)
      (let ((F (reversegraph (compose REACH (reversegraph ND))))
            (FTWL (make-graph)) (CVWL (make-graph)) (w (new-node))
            (G2 (make-graph)) (P2 (make-graph))
            (x nil) (d nil))
         (+!edge w n0 FTWL)
         (while (or (not (empty-graph FTWL)) (not (empty-graph CVWL)))
               (while (not (empty-graph FTWL))
                       (pxapply ((w)(x)() ((  x *) (  x *)
                                           (w 1 *) (w 0 =)
                                           (* * *) (* = =))) FTWL (w) (x))
                       (if (pmatch ((x)(d)() ((  d *)
                                              (x 1 *)
                                              (* * *))) ND (x) (d))
                           (first-time-def-node x FTWL CVWL G G2 P P2 RC)
                           (conditional-node x FTWL NQ G2 RC TB FB bot)))
               (when (not (empty-graph CVWL))
                       (pxapply ((w)(x)() ((  x *) (  x *)
                                           (w 1 *) (w 0 =)
                                           (* * *) (* = =))) CVWL (w) (x))
                       (if (pmatch ((x)(d)() ((  d *)
                                              (x 1 *)
                                              (* * *))) ND (x) (d))
                           (usual-def-node x CVWL G2 P RC)
                           (conditional-node x FTWL NQ G2 RC TB FB bot)))))))
```

Figure 6.22: Wegman and Zadeck's ConditionalDef algorithm

This algorithm has its own special way of propagating information, due to the presence of two worklists instead of one. It removes a node from the worklists, and does a simple case analysis on it. If the node is a conditional node, the action taken does not depend on where it came from. If, however, it is a definition, then either `first-time-def-node` or `usual-def-node` is invoked, depending on whether possible control flow has been propagated past the node. These def-node sub-algorithms evaluate the expression at the visited node, and propagate that information to each of the successors of that node (in `P`). They then place each such successor node into the appropriate worklist, depending on whether or not that node has been previously found to be executable, as determined by whether or not there is an edge entering it in `G2`. The conditional action evaluates the expression at the conditional node, then uses this information to place the appropriate subsequent node or nodes into the FTWL, if it or they have not previously been determined to be executable.

Wegman and Zadeck's second algorithm, termed ConditionalConstant, refines the propagation graph by adding intermediate nodes to indicate regions of the code through which values flow. The ordinary propagation graph P that we have been using heretofore may contain arcs from definitions to uses that only hold along execution paths that may be found to be invalid during some portion of the algorithm. Thus, the algorithm using it may not find some constants that are "ruined" by definitions that necessarily pass through non-executed code sections. To take care of these definitions, Wegman and Zadeck use the GlobalValue graph of Reif and Tarjan in place of the DefUseChain graph of Reif and Lewis. The difference between these two graphs is that the GlobalValue graph has additional points called "birthpoints" that collect together strands of the DefUseChain graph at places where flowgraph confluences occur. Birthpoints are added by inserting identity assignments in the right places, causing propagated constants to pass through them. Then, if a section of code containing one such identity assignment is deemed non-executable, all def-use chains passing through that section of code are cut off, and not entered into `P2`. Definitions that flow through such a section of code then can never be propagated further.

The algorithm of Reif and Tarjan to produce a GlobalValue graph works in linear time, but is extremely complicated, and a complete presentation does not seem to have appeared in the literature in one place. We have developed a fairly simple algorithm for computing this graph using graph transformations that can run in almost linear time. The idea is to add identity assignments at and just prior to each merge birthpoint of each variable. A merge birthpoint of a variable is a merge node through which some D-U chain for that variable passes. The algorithm is as follows:

Compute LIVE via any of the methods given earlier in this chapter. It
    describes which variables are live on exit of each node.
For each merge node n in the flowgraph
    For each predecessor p of n
        For each variable v live at p
            Add an identity assignment $v := v$ between p and n
            If v is not redefined at n
                Add an identity assignment $v := v$ between n and each
                    of its successors.

Adding an identity assignment is a simple matter of creating a new node and adding arcs in the relevant graphs (G, ND, DV, etc). The resulting set of graphs may be used in the ConditionalDef algorithm, converting it into the more powerful ConditionalConstant algorithm.

6.6.4 Discussion

We have seen four constant propagation algorithms, each of which is more powerful or more efficient than the last. The Reif & Lewis algorithm improves on Kildall's algorithm by substituting sparse graphs for dense ones. Wegman and Zadeck's ConditionalDef algorithm adds a second worklist to maintain knowledge about which flowgraph nodes can or cannot be executed, using this information to keep never-executed definitions from confounding the constant propagator. ConditionalDef is thus able to find more constants than can Kildall's or Reif & Lewis's algorithms. Wegman and Zadeck's ConditionalConstant algorithm substitutes a yet-more-sophisticated propagation graph to stop the propagation of definitions that pass through (but don't occur in) never-executed sections of code. As a result, ConditionalConstant is able to find more constants than can ConditionalDef.

Our system should allow us to discover answers to many open questions regarding these algorithms. We would like to be able to compare the amount of space and time consumed by each on real programs, and would like to be able to compare typical numbers of constants discovered by each. Such information would tell us whether the added complexity of the Wegman and Zadeck algorithms can pay for themselves on typical inputs. In addition, we would like to measure the amount of flowgraph pruning that the Wegman and Zadeck algorithms can be expected to do. Having comparable implementations of each of these algorithms is a big step toward answering these questions.

The most advanced algorithm presented above, the ConditionalConstant algorithm, uses the results of live variable analysis to modify the flowgraph, then passes the modified flowgraph to the ConditionalDef algorithm. ConditionalDef, following Reif & Lewis, then uses the results of reaching definitions analysis to produce a sparse propagation graph. This is a good example of how our graph-transformation system can make the combining of multiple algorithms into one a reasonable thing to do, especially with respect to development and maintenance costs. While it is certainly possible to write algorithms that are incompatible with others, the natural way of representing associations with graphs results in one natural data format for expressing data-flow solutions. The elimination of any need for data format conversion code thus shortens algorithms and enhances ease-of-use, readability, and maintainability, while not restricting the analysis algorithms that may be implemented or increasing their computational complexity.

Chapter 7

Experimental Results

The system described in this dissertation was incorporated into an experimental compiler optimizer, replacing its pre-existing analysis phase. This chapter describes the work we did performing this implant and our experiences with the result. It closes with some directions for further research and a summary of the project.

7.1 An optimizing Modula-2 compiler

Under construction at UC Berkeley is an experimental portable optimizing compiler for Modula-2. This compiler's optimizer and code generator are implemented using a portable language-independent system called Dora, which is organized around an intermediate language called DILS (Dora's Intermediate Language Schema). An overview of Dora and DILS will be presented here, but the reader is referred to [10] for a complete treatment.

7.1.1 DILS and Dora

DILS is an intermediate language schema based on the lambda calculus extended with a store. It differs from more standard forms of intermediate language in that it contains no operators with which to implement language operations. Rather, it is a *schema* in the sense that it provides a framework for an IL: it can be extended to include whatever operators are necessary. What DILS does provide is control-flow based on the lambda calculus. It includes lambda abstraction, function application, and a "labels" facility for defining sets of mutually-recursive functions. Non-local flow of control is modeled by continuation captures and continuation applications. DILS also supports the notions of variables, operators, and constants, although it actually implements only a few types of the latter.

By being an extensible framework rather than a pre-defined intermediate representation, DILS achieves machine- and language-independence. It also is able to express programs at varying levels of abstraction, from a very high-level source-like representation down to representations at the assembly code level. This makes it a nice vehicle for optimizing programs, since optimizations can be written once, yet be applied at several levels.

Dora may be thought of as a collection of languages specialized for writing optimizers for programs expressed in DILS. It consists of a language for giving descriptions of operators, a tree-transformation language based on tree pattern-matching, and an attribution language for assigning and manipulating DILS tree node attributes.

Together, Dora and DILS comprise a machine-independent language-independent environment for prototyping optimizing compilers. This is the environment into which the present system was installed. It should be noted that DILS, with its similarity to LISP, does not support the notions of "statement" or "basic block", but instead is organized at the expression level. Flow of control is expressed with functions, continuations, and left-to-right evaluation of parameters. These conditions complicate the problem of performing intraprocedural flow analysis, and so provide a worthy test for a system purporting to make analysis easy.

7.1.2 Analysis algorithms used by the optimizer

The optimizer is a functional prototype of UOPT, the portable optimizer developed in Frederick Chow's Ph.D. dissertation [5]. UOPT is interesting from the flow analysis perspective in that it incorporates the algorithm of Morel and Renvoise [20] which propagates information bidirectionally along flowgraph edges. This is the most advanced analysis algorithm used by UOPT, the others falling into the class of standard intraprocedural bitvectoring algorithms.

The original hand-coded data-flow analyzer that our system replaced computed the same information as Morel and Renvoise's algorithm, but in a way different from that in the original specification. The approach taken in the Dora implementation was to break the bidirectional analysis into two unidirectional flows. This was done as follows.

As in the original algorithm, the local properties *TRANSP*, *COMP*, and *ANTLOC* (transparent, computed, and locally anticipated) are provided by each flowgraph node, from which the global properties *AVAIL*, *ANT*, and *PAVAIL* (availability, anticipability, and partial availability) may be calculated by standard unidirectional means. From these, the property *MMA* (might-make-available) is computed at each node x via the formula

$$MMA(x) \stackrel{def}{=} AVAIL(x) \vee \left(ANT(x) \wedge \prod_{s \in succ(x)} (PAVAIL(s) - AVAIL(s)) \right)$$

The property *CMA* (can-make-available) is then defined to be

$$CMA(x) \stackrel{def}{=} \prod_{s \in sib(x)} MMA(s)$$

where $x \in sib(y)$ if there is some flowgraph node z that is both a successor of x and a successor of y. The *sib* relation is defined in terms of successors and not predecessors because we are interested in converting partially available expressions into available expressions at common successor nodes. [1]

Both the *MMA* and *CMA* computations can be seen to use unidirectional flows, the necessity for bidirectional flow having been obviated by the *sib* relation. The properties desired for optimization can be determined from *CMA*, *MMA*, and more standard data flow information such as *AVAIL*. For example, the property *INS*, which tells where in the flowgraph to insert expressions, may be computed as $CMA - PAVAIL$.

7.2 Installation of the graph system

The graph system was written using the COMMON LISP package facility, and so was incorporated into Dora simply by the addition of its code. This made the algorithms of Chapter 6 available to Dora. New code was then written to construct the flowgraph, to construct auxiliary graphs from DILS code, and to allow Dora to use the flow analyzer's result graphs. Finally, calls to the original hand-coded data-flow functions were replaced with calls to their newly-available equivalents.

The library of available graph-based analyzers includes analyzers for availability, partial availability, anticipability and partial anticipability, among others. Each of these has

[1] Note that the graph representing the *sib* relation may be calculated very simply by `(compose (reverse G) G)` where `G` is the relevant flowgraph.

been implemented using three underlying algorithms: worklist, workgraph, and Graham-Wegman. All of these are in Dora, and the implementation of each analyzer that is to be used is selectable at run-time.

Performing control-flow analysis on DILS was a somewhat less straightforward process than performing it on an ordinary statement-oriented language such as PASCAL or C would be. This is primarily due to the fact that a completely general DILS control-flow analyzer must be able to handle the problems introduced by functional languages such as LISP, in which it is frequently the case that a non-constant expression must be evaluated to determine which function to call. Dora contains a powerful tree pattern-matcher and tree transformer specialized for working with DILS that makes the job of constructing control-flow graphs for DILS fairly easy. This being the case, we did not re-implement the control-flow analyzer using the graph transformation system.

We can, however, make a convincing case that it is feasible to use our system for control-flow analysis by noting that Dora's tree pattern matching primitives can be implemented using graph transformations. An implementation based on this approach would probably not manipulate DILS as efficiently as Dora can. The point is not that this would be a good method for performing control-flow analysis, but rather that our language is sufficiently powerful for the task.

We implemented several functions to extract the other information needed by the data-flow analysis algorithms from a tree-like graph representation of DILS code. Again, given that Dora has specialized features for dealing with DILS, it is doubtful that this was the most efficient method of computing these graphs. Nevertheless, the exercise demonstrated that our system is capable of dealing with this kind of complexity, and that the coding effort to construct these auxiliary graphs is small.

7.3 Comparison with a hand-coded analyzer

Our system was designed primarily from the standpoint of making flow analysis algorithms easy to implement and easy to use. Since speed is often an extremely important concern, our aim was to make our system as fast as possible, but without compromising our primary goal.

The work required to incorporate our system into Dora and to make it perform useful flow analysis (as distinct from the work necessary to construct our system and its library of analyzers) took only three afternoons. Since this was the first use of the transformation system, an appreciable fraction of that time was spent streamlining its interface to make it easier to use. This cost will not recur, and so we anticipate that subsequent uses will require even less effort.

In effect, we have shown that the addition of a flow-analysis component to a compiler can be an easy and low-cost proposition. From when we started the integration process to the point at which Dora was using it to perform real optimizations, no time needed to be spent designing, implementing, or most importantly, debugging flow analysis algorithms.

Table 7.1 gives some measured timings for the calculation of availability and partial availability information on four test cases. It can be noticed from the fourth column that the graph-based system is slower than the original by a factor of less than seven. The two simplest cases are less slow, probably because they contain no loops. We anticipate that a re-implementation of the graph data structure and its sub-structures in light of knowledge of how they are typically accessed will lead to substantial speed improvements.

# nodes	Original	Graph-based	Relative slowness
26	50μs	217μs	4.3x
40	70μs	400μs	5.7x
90	217μs	1417μs	6.5x
232	850μs	5717μs	6.7x

Table 7.1: Speed comparison

It should be noted that these measurements are not completely conclusive, since faster and more sophisticated algorithms are easily used with our system, and their improved asymptotic behavior may well cause them to outperform the hand-coded original (which was worklist-based).

Estimates of the amount of memory required to run the system are difficult to come by, because the LISP garbage collector confounds many attempted measurements. Nevertheless, we can say a few things based on the implementation we used. A typical adjacency-list structure consumes one cell per node and two cells per edge. Each of our graphs include two such; one for the graph itself and one for the reverse of that graph. In addition, we maintain a tree structure atop each adjacency-list to make edge insertions, deletions, and existence tests fast. The total space cost comes to two cells per node and eight cells per edge, roughly four times normal. This cost can be minimized by generating the reverse graph structure only when needed, and by building tree-structures over adjacency-lists only when those lists become large. Since most graphs are sparse, it should be reasonably easy to approximate the space cost of a traditional adjacency-list quite closely.

It is difficult to estimate the space taken by the graphs used to associate data-flow attributes with flowgraph nodes compared to bit-vectors. This is due to the fact that a bit-vector is a dense representation, while a graph is a sparse one. Should further measurements show there to be a great discrepancy, the system could be modified to take hints to the effect that certain graphs should be represented internally as bit-vectors. Of course, the high-level interface to these graphs would remain unchanged.

7.4 Directions for further research

This section suggests several avenues worthy of further exploration, ranging from paths to a fuller understanding of graph transformations to ways of constructing better analyzers.

The present work raises some interesting questions about algorithms for matching patterns in graphs. Recall from Chapter 3 that the search algorithm used here considers the pattern variables in a fixed order which is determined by a topological ordering of a pattern's NCC. There are many such topological orderings; the one we use is determined by the "alphabetically least" sub-ordering on the variable names. It may well be the case, however, that the use of different topological sorts may cause pattern match failures to be detected earlier, hence speeding-up the whole matching process on average. (The worst-case time will, of course, still be exponential.) This facet of graph pattern matching, along with the possibility of finding better heuristics for choosing pattern seeds, is worth investigating.

It would also be interesting and useful to be able to characterize patterns in order to tell which patterns are likely to match quickly. In our development of algorithms, we often found several ways to implement the same transformation, based on different ways to match some graph structure and to assign nodes to variables. A study of patterns could lead to a better pattern compiler that could optimize a pattern automatically after analyzing its structure.

A study of how transformations can interact could also lead to more efficient transforming by allowing us to combine two adjacent transformations into one that is more efficient, or to split one transformation into two if circumstances warrant. The development of an algebra of patterns and transformations could thus provide many opportunities for optimizing our flow analysis algorithms.

A somewhat different direction may be taken by noting that our language could be used to analyze programs written in itself. This could produce not only the obvious benefit of allowing us to optimize our analysis programs, it might also aid us in constructing incremental algorithms automatically from non-incremental versions. We could use dependency information, for example, to decide which intermediate graphs in an algorithm needed to be updated if a change were made to an input graph. Further, if the information being maintained about these graphs was fine-grained enough, we might be able to decide *a priori* which parts of a graph needed updating, thus minimizing the time for the update operation. The feasibility of this suggestion will only be determined by further analysis.

7.5 Project summary

The primary goal of this project was to demonstrate that an approach based solely on graphs and transformations of graphs was sufficient for performing flow analysis, and could facilitate the use of multiple analysis algorithms in a single program. A second goal was to demonstrate that such an approach could lead to a substantial degree of software re-use, allowing the application of even the hardest-to-implement flow analysis algorithms to become commonplace. We believe that we have met both of these goals.

We first developed a small family of graph transformations and constructed algorithms for carrying them out. We then built a programming language based on this transformational machinery, and extended it with utilities written using graphs and transformations. A library of program flow analyzers was subsequently developed, depending only on the facilities present in this (extended) language. Finally, this entire system was incorporated into an existing compiler back-end, replacing its pre-existing analysis phase with standard routines from the analyzer library. It was demonstrated, through the development of several advanced algorithms, that different implementations of the same analyzers may be written easily and made use of without disturbing the enclosing compiler.

It should be noted that the flow analysis algorithms that we implemented were written without taking any of the peculiarities of DILS into account. They were constructed simply to operate on a flowgraph having associated *gen* and *kill* graphs, whether that flowgraph is expression-based, as with DILS, or based on basic blocks, as is the case in more conventional optimizers. The fact that these algorithms do not change just because the flowgraph on which they work belongs to a different paradigm of compiler construction supports the claim that our graph-based language can make a data-flow analyzer act like a black-box. Indeed, the present system makes it easy to take standard flow analysis components from a library and incorporate them into an optimizer. In other words, the system seems to make possible a high degree of software re-use, at least in the domain of flow analysis on graphs. This being the case, the argument that some analysis algorithms are too complicated to warrant implementation loses much of its weight.

In our system, even the most complicated algorithms interact with the compiler and with each other through a single easily-understood data structure, the directed graph, and through a small and easily-characterized family of transformations. This minimal computational basis helps to ensure not only that this package can be incorporated into a compiler with minimal programming effort, but also that flow analysis algorithms can interact with one another without requiring elaborate interfaces.

We have shown that graphs are a suitable representation for flow analysis data structures, and that our family of transformations is powerful enough to allow us to solve most if not all program flow analysis problems. Furthermore, our transformations are efficient enough to solve these problems without changing the asymptotic time bounds of any algorithm we implement. They are also fairly efficient in a real sense: our experimental prototype is almost fast enough to be used in real compilers.

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, MA, 1986.

[2] Allen, F. E. Control flow analysis. SIGPLAN *Notices 5*, 7 (1970), 1–19.

[3] Banning, J. P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1979), pp. 29–41.

[4] Burke, M., and Cytron, R. Interprocedural dependence analysis and parallelization. In *Proceedings of* SIGPLAN *'86 Symposium on Compiler Construction* (July 1986), pp. 162–175.

[5] Chow, F. C. *A Portable Machine-independent Global Optimizer – Design and Measurements.* Ph.D. dissertation, Stanford University, Dec. 1983.

[6] Cocke, J. Global common subexpression elimination. SIGPLAN *Notices 5*, 7 (1970), 20–24.

[7] Cooper, K. D., and Kennedy, K. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of* SIGPLAN *'84 Symposium on Compiler Construction* (June 1984), pp. 247–258.

[8] Cooper, K. D., and Kennedy, K. Efficient computation of flow insensitive interprocedural summary information – a correction. SIGPLAN *Notices 23*, 4 (Apr. 1988), 35–42.

[9] Cytron, R., Lowry, A., and Zadeck, K. Code motion of control structures in high-level languages. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1986), pp. 70–85.

[10] Farnum, C. D. *Pattern-Based Languages for Prototyping of Compiler Optimizers.* Ph.D. dissertation, CS Division, EECS, University of California, Berkeley, Dec. 1990.

[11] Ferrante, J., Ottenstein, K. J., and Warren, J. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[12] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[13] Graham, S. L., and Wegman, M. N. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*, 1 (Jan. 1976), 172–202.

[14] Harary, F. *Graph Theory.* Addison Wesley, 1972.

[15] Hecht, M. S. *Flow Analysis of Computer Programs.* Elsevier North Holland Inc., New York, 1977.

[16] Hecht, M. S., and Ullman, J. D. A simple algorithm for global data flow analysis problems. *SIAM J. Comput. 4*, 4 (Dec. 1977), 519–532.

[17] Hunt, H. B., Szymanski, T. G., and Ullman, J. D. Operations on sparse relations. *Commun. ACM 20*, 3 (Mar. 1977), 171–176.

[18] Kildall, G. A. A unified approach to global program optimization. In *Conference Record, ACM Symposium on Principles of Programming Languages* (Oct. 1973), pp. 194–206.

[19] Kuck, D. J., Kahn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1981), pp. 207–218.

[20] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Commun. ACM 22*, 2 (Feb. 1979), 96–103.

[21] Steele, G. L., and Sussman, G. J. The revised report on Scheme: a Dialect of LISP. Tech. Rep. 452, M.I.T. Artificial Intelligence Laboratory, Cambridge, MA, Jan. 1978.

[22] Tarjan, R. E. Fast algorithms for solving path problems. *J. ACM 28*, 3 (July 1981), 594–614.

[23] Warshall, S. A theorem on Boolean matrices. *J. ACM 9*, 1 (Jan. 1962), 11–12.

[24] Wegman, M. N. *General and Efficient Methods for Global Code Improvement.* Ph.D. dissertation, CS Division, EECS, University of California, Berkeley, Dec. 1981.

[25] Wegman, M. N., and Zadeck, F. K. Constant propagation with conditional branches. In *Conference Record, Twelfth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1985), pp. 291–299.