A Generalised Theory of

Bit Vector Data Flow Analysis

Submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

by

Uday P. Khedker

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay.

1995

Dedicated to the memory of my high-school teacher

Late Shri. S. V. Kanade

It was obvious to me even then that the lama who was my guide was indeed a good man, and one whom I would follow to the utmost of my ability. It was clear that he knew a very great deal about me, far more than I knew myself. I was looking forward to studying with him, and I resolved that no one should have a better pupil. There was, as I could plainly feel, a very strong affinity between us, and I marvelled at the workings of Fate which had placed me in his care.

- T. Lobsang Rampa
The Third Eye

All other quotations in this thesis are from *The Little Prince* — an English translation of the French classic by Antoine De Saint-Exupéry.

Acknowledgements

> *"To me, you are still nothing more than a little boy who is just like a hundred thousand other little boys. And I have no need of you. And you, on your part, have no need of me. To you, I am nothing more than a fox like a hundred thousand other foxes. But if you tame me, then we shall need each other. To me, you will be unique in all the world. To you, I shall be unique in all the world ..."*

One realisation which dawns at the time of documenting any work of this measure is : Contrary to what the title page may indicate, any work of this measure cannot be credited to one single person (or to a small group of persons). To be of any real value, such a work must, necessarily have contributions from a wide variety of people. While their kind and degree of involvement may vary, each one of such persons plays a very vital role in shaping the work, and eventually, the person who is often credited with the work.

One person who stands apart among all is my guide Prof. D. M. Dhamdhere. I must thank him for providing a very clear-cut and objective direction to my research, as also for letting me deviate from it time and again, thus allowing evolution rather than just plain deduction of ideas. Being a tough task master that he is, he often pushed me "beyond the wall" and I must admit that on putting additional effort, I have almost always discovered that no wall existed.

Prof. S. Biswas, Prof. A. Sanyal and Prof. A. A. Diwan have been associated with this work from the very beginning in that they were entrusted with the responsibility of evaluating my work from time to time and they indeed provided very constructive criticism.

I also take this opportunity to express my gratitude to all the faculty members of this department, particularly Prof. D. B. Phatak for encouraging me in every possible way. The administrative staff too, particularly Mr. Chandran and Mrs. Athwankar have been extremely helpful. Their help and co-operation simplified many things which otherwise could have been very difficult to manage. The department also provided an excellent company of extremely close friends — Sachin Chitnis, Manoranjan Satpathy, Vinod Kulkarni, Mukesh Mohania, Sandeep Pagey, Rubin Parekhji, Ramesh Babu, to name a few. Never in my life can I expect such a large group of intimate friends available for participating in absolutely any kind of activity that I may be interested in/required to indulge in. One unwritten rule that we observed, and I must thank them all for that, was that we would never ever discuss our research with each other. In spite of this rule, or may be because of this rule, all of them have been extremely supportive in the face of any mental/physical adversity and have been equally demanding in their share in the moments of happiness.

I am also thankful to the faculty, staff and students of the Department of Computer Science, University of Pune, for providing a very congenial atmosphere for winding up the remaining part of my research. The students in particular bore the brunt of having to study a large part of this research in a course titled “Advanced Topics in Compiling”. I express my deep gratitude to Prof. H. V. Sahasrabuddhe for making all this possible.

Several persons have contributed directly/indirectly to the implementation carried out for verifying the claims made in the theory. I would particularly like to thank Kurien Jacob, Rajit Manohar, Kavita Bala, Girija Narlikar, Sandeep Kumar, Moses Charikar and Viral Acharya. Sachin Chitnis and Manoranjan Satpathy were the first victims of early versions of the incremental data flow analysis algorithm. Sachin Chitnis and Vinod Kulkarni have been very eager to make me use the computers more effectively; I don’t know how much I have learnt in the process, but most of my work progressed smoothly because of that. Sandeep Pagey was the person who introduced me to LaTeXand thereby made my life simpler.

Last, but not the least, I would like to thank my parents for being the driving force of my life all throughout. And I must thank my wife Arundhati for her patience while I was devoting my time to my research (and for her impatience while I wasn’t, asking why I wasn’t!). My twin daughters, Sneha and Mugdha, arrived in this world at a very opportune time and forced me to hasten the submission of this thesis.

Contents

Abstract

The classical theory of data flow analysis, which has its roots in unidirectional flows, is inadequate to characterise bidirectional data flow problems. We present a generalized theory of bit vector data flow analysis which explains the known results in unidirectional and bidirectional data flows and provides a deeper insight into the process of data flow analysis. This is achieved by carefully distinguishing between the information flow through a node and the information flow along an edge. This, in essence, facilitates a more powerful characterization of information flow paths which is more general than the classical notion based on graph theoretic paths. This careful distinction has far reaching consequences, both on the theory and practice of data flow analysis.

Based on the theory, we develop a worklist-based generic algorithm which is uniformly applicable to unidirectional and bidirectional data flow problems. It is simple, versatile and easy to adapt for a specific problem. The theory and the algorithm are applicable to all bounded monotone data flow problems which possess the property of the separability of solution.

The theory yields valuable information about the complexity of data flow analysis. We show that the complexity of worklist-based iterative analysis is same for unidirectional and bidirectional problems. We also define a measure of the complexity of round-robin iterative analysis which captures the influence of graph structures on the flow of information. It is uniformly applicable to unidirectional and bidirectional problems and provides a tighter bound for unidirectional problems than the traditional measure of the *depth of a graph*.

We also extend the proposed theory and methods to incremental data flow analysis. This is achieved by proposing a *functional* model for incremental data flow analysis which is independent of any technique/algorithm used for performing analysis, unlike the traditional approaches. This separation of algorithm from the process of analysis leads to a better theorisation of the notions involved in incremental data flow analysis. In particular, it facilitates a concrete definition of incremental solution making it possible to show the correctness of incremental data flow analysis, apart from providing a general algorithm-independent explanation of incremental data flow analysis.

List of Symbols

Symbols used in localised contexts have not been included in this list.

$\sqcap$	:	Confluence operator
$\top$	:	Top element of $\mathcal{L}$
$\perp$	:	Bottom element of $\mathcal{L}$
$\otimes$	:	Update operation for updating the old MFP solution incrementally
δ_f	:	Traversal along a forward edge in direction δ
δ_b	:	Traversal along a back edge in direction δ
δ_f^-	:	Traversal along a forward edge in direction δ^-
δ_b^-	:	Traversal along a back edge in direction δ^-
δ_G	:	Traversal over the graph in direction δ
δ_G^-	:	Traversal over the graph in direction δ^-
$\propto(p)$	:	The set of properties corresponding to p
$\pi(x)$	:	A partition of the set x
$\sigma(x)$	:	A subset in $\pi(x)$
$\partial\mathcal{B}_h^m$	:	A change in the bit function $\mathcal{B}_h$
Δ	:	The set containing possible changes in a bit function
$\Delta\mathcal{P}$	:	Incremental change in $\mathcal{P}$
$\Delta\mathcal{S}$	:	Incremental change in S
$\Omega(p)$	:	The set of functions influencing the value of p
Π	:	Boolean product
$\Psi\star$	:	Set of the distinct cases that may arise due to a change $\partial\mathcal{B}_h^m$
Σ	:	Boolean sum
$< u, v, \rho >$	:	The information flow path ρ from program point u to program point v
f_i^b	:	Backward node flow function for node i
f_i^f	:	Forward node flow function for node i

$g^b_{(i,j)}$	:	Backward node flow function for edge (i,j)
$g^f_{(i,j)}$	:	Forward node flow function for edge (i,j)
h^m	:	Modified (bit vector) function
n	:	$\mid N \mid$, number of nodes in G
p	:	Some property
$p \leftarrow h(p')$	:	The property p' influences the value of p through the function h
$p \leftarrow \mathcal{B}_h^(p')$	:	The property p' influences the value of p through the bit function $\mathcal{B}_h$ of h
r	:	Number of properties associated with a node
$entry(G)$	:	Set of entry nodes of G
$exit(G)$	:	Set of exit nodes of G
$in(i)$	:	Entry point of node i
$old(x)$	:	Old entity x in the previous instance of **D**
$out(i)$	:	Exit point of node i
D	:	Data flow framework
F	:	False or 0 value of a bit
I	:	Instance of a data flow framework
S	:	Specification of a data flow framework
T	:	True or 1 value of a bit
$\overline{\mathrm{F}}$	:	All bits in the bit vector are **F**
$\overline{\mathrm{T}}$	:	All bits in the bit vector are **T**
BOT	:	Value of an individual property in the bit vector corresponding to the $\perp$ element of $\mathcal{L}$
CH(p,p')	:	The chain from p to p' (In the context of exhaustive data flow analysis)
CH(p,p')	:	The set of chains from p to p' (In the context of incremental data flow analysis)
CONST_IN$_i$	:	Constant properties associated with the $in(i)$
CONST_OUT$_i$	:	Constant properties associated with the $out(i)$
FLOW$_{\mathcal{P}_r}$	:	Path flow function for an information flow path $\mathcal{P}_r$
FP(i)	:	Information associated with node i in a Fixed Point Assignment FP
IN$_i$	:	Information associated with $in(i)$
MOP(i)	:	Information associated with node i in a Maximum Safe Assignment MOP

OUT_i	:	Information associated with $out(i)$
$\mathrm{SA}(i)$	:	Information associated with node i in a Safe Assignment SA
TOP	:	Value of an individual property in the bit vector corresponding to the $\top$ element of $\mathcal{L}$
E	:	Set of edges in G
G	:	Program flow graph
M	:	$N \rightarrow \mathcal{F}$ (In the context of exhaustive data flow analysis)
M	:	$< M_{\mathcal{F}}, M_{\mathcal{G}} >$ (In the context of incremental data flow analysis)
$M_{\mathcal{F}}$	:	$N \rightarrow \mathcal{F}$
$M_{\mathcal{G}}$	:	$E \rightarrow \mathcal{G}$
N	:	Set of nodes in G
Q	:	Data flow equations
S	:	Solution of some problem $\mathcal{P}$
S'	:	Solution of the problem $\mathcal{P} + \Delta\mathcal{P}$
T_b^f	:	Forward traversal along a back edge
T_b^b	:	Backward traversal along a back edge
T_e^f	:	Forward traversal along an edge
T_e^b	:	Backward traversal along an edge
T_f^b	:	Backward traversal along a forward edge
T_f^f	:	Forward traversal along a forward edge
TR_0	:	Initial trigger set
X_0	:	Initial values of the variables in Q
$X(u)$	:	Bit vector representing the properties at program point u
$X^i(u)$	:	Property represented by the i^{th} bit at program point u
$Pos(p)$	:	Bit position of a property p in the bit vectors
$\mathcal{P}p(p)$	:	Program point with which p is associated
$\mathcal{P}_r$	:	An information flow path reaching node r
$\mathcal{P}_r^{in}$	:	An information flow path reaching $in(r)$
$\mathcal{P}_r^{out}$	:	An information flow path reaching $out(r)$
$\mathcal{B}_h^i$	:	Bit function for the i^{th} property in a bit vector
$\mathcal{B}_h^m$	:	Modified bit function of h^m
$\mathcal{D}(p)$	:	The set of properties which p depends on for its value
$\mathcal{F}$	:	Set of node flow functions

$\mathcal{G}$	:	Set of edge flow functions
$\mathcal{L}$	:	Lattice containing the elements representing the information that may be associated with the nodes of a program flow graph
$\mathcal{N}(p)$	:	The set of neighbouring properties of p (i.e. the properties which are influenced by p)
$\mathcal{N}^{-1}(p)$	:	Inverse of $\mathcal{N}(p)$ (i.e. the properties which influence p)
$\mathcal{AR}_p$	:	Affected region for the property $p \in \mathcal{B}2\mathcal{T}$
$\mathcal{BR}_p$	:	Boundary of $\mathcal{AR}_p$
$\mathcal{B}2\mathcal{T}$	:	Set of properties which have changed from BOT to TOP, locally
$\mathcal{B}2\mathcal{T}\star$	:	Set of properties which have changed from BOT to TOP, globally
$\mathcal{GC}$	:	Set of properties which change globally
$\mathcal{GCB}(p)$	:	Set of properties corresponding to p which must change to BOT due to a TOP to BOT change in p
$\mathcal{GCT}(p)$	:	Set of properties corresponding to p which may change to TOP due to a BOT to TOP change in p
$\mathcal{LC}$	:	Set of properties which change locally
$\mathcal{NCT}(p)$	:	Those properties in $\mathcal{GCT}(p)$ which cannot be TOP
$\mathcal{TR}^+$	:	Set of properties that must be added to TR_0
$\mathcal{TR}^-$	:	Set of properties that must be removed from TR_0
$\mathcal{T}2\mathcal{B}$	:	Set of properties which have changed from TOP to BOT, locally
$\mathcal{T}2\mathcal{B}\star$	:	Set of properties which have changed from TOP to BOT, globally
$\mathcal{VT}$	:	Set of TOP properties in the MFP solution
$\mathcal{VB}$	:	Set of BOT properties in the MFP solution

Chapter 1

Introduction

> *The Earth is not just an ordinary planet! One can count, there, 111 kings, 7,000 geographers, 900,000 businessmen, 7,500,000 tiplers, 311,000,000 conceited men — that is to say, about 2,000,000,000 grown-ups.*
> *To give you an idea of the size of the Earth, I will tell you that before the invention of electricity it was necessary to maintain, over the whole of the six continents, a veritable army of 462,511 lamplighters for the street lamps.*

An optimising compiler performs code improvements by applying transformations which are based on the information concerning the uses and definitions of data items is collected through the process of pre-execution (i.e. compile-time) analysis of a program which includes *control flow analysis* and *data flow analysis* [2, 32].

1.1 What Is Data Flow Analysis?

Control flow analysis traces the patterns of possible execution paths in a program. For this purpose the program is represented as a directed graph. A call graph depicts the transfer of control between procedures whereas a control flow graph shows control flow within a procedure. The construction, representation, structure, and properties of such graphs are part of control flow analysis.

Data flow analysis traces the possible definitions and uses of data along the potential control flow paths and collects the information about attributes of certain data items. A data flow analysis problem is formulated by representing the data flow information associated with the nodes of the flow graph by variables; interdependencies of the values of these variables give rise to simultaneous equations. Data flow analysis is performed by solving the system of simultaneous equations.

Data flow analysis can be *inter-procedural* or *intra-procedural*; the former is performed over the call graph of a program while the latter is performed over the control flow graph of a procedure in a program. This thesis restricts itself to the latter.

Exhaustive data flow analysis computes the information from scratch. When repeated applications of an optimising transformation are required, it should be possible to *update* the information gathered for the previous application of the transformation rather than recompute it. This is called *incremental data flow analysis*.

An optimising compiler needs to perform many analyses during the process of optimisation. Thus the issues concerning the correctness, efficiency and profitability of data flow analysis become important for the optimising compiler theory and practice. Though these aspects have been explored in details and a formal theory for exhaustive data flow analysis exists [2, 32, 46], its scope is restricted to *unidirectional* data flow problems in which the data flow information associated with a node of the program flow graph is influenced either by its predecessors (*viz.* available expressions and reaching definitions), or by its successors (*viz.* live variables and busy expressions). Such data flows can be readily classified into *forward* and *backward* data flows [2, 32]. Data flow problems which unify several optimisations remain outside the purview of this theory since they typically involve *bidirectional* dependencies, i.e. the data flow information associated with a node depends on its predecessors as well as its successors. Section 1.2 introduces bidirectional data flow analysis.

Though bidirectional data flow problems have been known for over a decade, it has not been possible to explain the intricacies of bidirectional flows using the traditional theory of data flow analysis. Although a fixed point solution for a bidirectional problem exists, the flow of information and the safety of an assignment can not be characterised formally. Because of this theoretical lacuna, efficient solution techniques to bidirectional data flow problems have not been found though some isolated and ad hoc results have been obtained [17, 24, 25].

Incremental data flow analysis has received considerable attention [9], however most of the work reported in the literature consists of isolated techniques and there is little or no theoretical treatment independent of the technique being proposed. Besides, almost all approaches are restricted to unidirectional data flow problems and the issue of incremental algorithms for bidirectional problems has remained un-addressed.

1.2 Bidirectional Data Flow Analysis

1.2.1 Unified Optimising Transformations

The use of advanced data flow problems can be said to have begun with two independent efforts – one by Morel & Renvoise [45] and the other by Dhamdhere & Isaac [22], almost around the same time.

The Morel-Renvoise Algorithm for partial redundancy elimination (called MRA in the following) unifies several traditional optimising transformations, *viz.* code movement, common subexpression elimination and loop optimisation. Besides, the use of this algorithm does not require awareness of the program structure, i.e. control flow analysis need not be performed for a program in spite of the loop optimisation. These advantages lead to a reduction in the size as well as the running time of an optimiser. A 35% reduction in the size and a 30% to 70% reduction in the execution cost has been reported in [45]. It has been implemented in several production compilers *viz.* MIPS, PL.8 and Harris Night Hawk compilers for C, FORTRAN, and Ada.

The other pioneering effort towards unification [22] involved code movement, loop optimisation and strength reduction. It used the notion of *optimal placement of code* using execution frequency information for various paths in a program. While the conceptual features of optimal placement and the incorporation of strength reduction are important, feasibility of this approach is limited to those programs for which the execution frequency information is available.

1.2.2 An Introduction to Morel-Renvoise Algorithm

This section introduces the Morel and Renvoise Algorithm (MRA) [45] which is used as a representative bidirectional problem throughout the thesis.

The data flow properties and the data flow equations for MRA are given in Figure 1.1. Note that PPIN_i is the bit vector for node i which represents the property PPIN for *all expressions*, whereas PPIN_i^l is the bit representing the expression e_l.

Local property ANTLOC_i^l represents *local anticipability*, i.e. existence of an upwards exposed expression e_l in node i, while TRANSP_i^l reflects *transparency*, i.e. the absence of definition(s) of any operand(s) of e_l in the node. The global property of *anticipability* (ANTIN_i^l/ANTOUT_i^l) indicates whether expression e_l is very busy at the entry/exit of node i — a necessary and sufficient condition for the safety of placing an evaluation of e_l at the

Local data flow properties :

ANTLOC_i^l	Node i contains a computation of e_l, not preceded by a definition of any of its operands.
COMP_i^l	Node i contains a computation of e_l, not followed by a definition of any of its operands.
TRANSP_i^l	Node i does not contain a definition of any operand of e_l.

Global data flow properties :

$\text{AVIN}_i^l/\text{AVOUT}_i^l$	e_l is available at the entry/exit of node i.
$\text{PAVIN}_i^l/\text{PAVOUT}_i^l$	e_l is partially available at the entry/exit of node i.
$\text{ANTIN}_i^l/\text{ANTOUT}_i^l$	e_l is anticipated at the entry/exit of node i.
$\text{PPIN}_i^l/\text{PPOUT}_i^l$	Computation of e_l may be placed at the entry/exit of node i.
INSERT_i^l	Computation of e_l should be inserted at the exit of node i.
REDUND_i^l	First computation of e_l existing in node i is redundant.

Data flow equations :

$$\text{PPIN}_i = \text{PAVIN}_i \cdot (\text{ANTLOC}_i + \text{TRANSP}_i \cdot \text{PPOUT}_i) \cdot \prod_{j \in pred(i)} (\text{AVOUT}_j + \text{PPOUT}_j) \tag{1.1}$$

$$\text{PPOUT}_i = \prod_{k \in succ(i)} (\text{PPIN}_k) \tag{1.2}$$

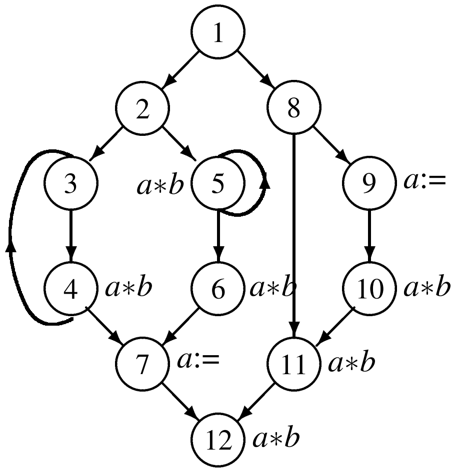
$$\text{INSERT}_i = \text{PPOUT}_i \cdot \neg\text{AVOUT}_i \cdot (\neg\text{PPIN}_i + \neg\text{TRANSP}_i)$$

$$\text{REDUND}_i = \text{PPIN}_i \cdot \text{ANTLOC}_i$$

Figure 1.1: The Morel-Renvoise Algorithm

entry/exit of the node [37]. Equations 1.1 and 1.2 do not use $\text{ANTIN}_i^l/\text{ANTOUT}_i^l$ properties explicitly; they are implied by $\text{PPIN}_i^l/\text{PPOUT}_i^l$ properties. The data flow property of *availability* ($\text{AVIN}_i^l/\text{AVOUT}_i^l$) is computed using the classical forward data flow problem [2]. The partial redundancy of an expression is represented by the *partial availability* of the expression (PAVIN_i^l) at the entry of node i. PPIN_i^l indicates the feasibility of placing an evaluation of e_l at the entry of i while PPOUT_i^l indicates the feasibility of placing it at the exit. Computations of an expression e_l are inserted at the exit of node i if $\text{INSERT}_i^l = \mathbf{T}$. REDUND_i^l indicates that the upwards exposed occurrence of e_l in node i is redundant and may be deleted.

The PPIN_i equation is slightly different from the original equation in MRA; the term PAVIN_i is used instead of the original term $\text{ANTIN}_i \cdot (\text{PAVIN}_i + \neg\text{ANTLOC}_i \cdot \text{TRANSP}_i)$ to prohibit redundant hoisting when the expression is not partially available. The PAVIN_i term represents the *profitability* of hoisting in that there exists at least one possible execution path along which the expression is computed more than once. The other two terms in the PPIN_i



Node	Transp	Antloc	Pavin	Avout	Ppin	Ppout	Insert	Redund
1	T	F	F	F	F	F	F	F
2	T	F	F	F	F	T	T	F
3	T	F	T	F	T	T	F	F
4	T	T	T	T	T	F	F	T
5	T	T	T	T	T	T	F	T
6	T	T	T	T	T	F	F	T
7	F	F	T	F	F	T	T	F
8	T	F	F	F	F	F	F	F
9	F	F	F	F	F	F	F	F
10	T	T	F	T	F	F	F	F
11	T	T	T	T	F	T	F	F
12	T	T	T	T	T	F	F	T

Figure 1.2: Program flow graph and properties for example 1.1

equation represent the *feasibility* of hoisting.

Bidirectional dependencies of MRA arise as follows : *Redundancy* of an expression is based on the notion of availability of the expression which gives rise to forward data flow dependencies (reflected by the $\prod$ term in the PPIN$_i$ equation). The *safety* of code movement is based on the notion of anticipability of the expression which introduces backward dependencies in the data flow problem (reflected by the $\prod$ term in the PPOUT$_i$ equation).

Example 1.1 : Consider the program flow graph in Figure 1.2. The partial redundancy elimination performed by MRA subsumes the following three optimisations :

- *Loop Invariant Movement* : The computations of $a * b$ in node 4 and node 5 are hoisted out of the loops and are inserted in node 2. It is readily seen from the table that $\text{REDUND}_4^l = \text{REDUND}_5^l = \text{INSERT}_2^l = \mathbf{T}$.

- *Code Hoisting* : The partially redundant computation of $a * b$ in node 12 is hoisted to node 7. As a result of suppressing this partial redundancy, the path 1-8-11-12 would have only *one* computation of $a * b$; the unoptimised program has two.

- *Common Subexpression Elimination* : The totally redundant computation of $a * b$ in node 6 is deleted as an instance of common subexpression elimination.

Note that the partially redundant computation $a * b$ in node 11 is not suppressed since hoisting it to node 8 would be unsafe — the path 1-8-9 had no computation of $a * b$ in the original program. □

1.2.3 Other Bidirectional Data Flow Problems

The algorithms by Morel & Renvoise and Dhamdhere & Isaac have inspired several other unifications and several extensions have been reported in the literature. Strength reduction has been incorporated within the Morel-Renvoise framework [19, 34, 35]. The applicability of MRA has been extended to suppression of partial redundancy of assignments apart from that of expressions within the *same* framework [21]. This increases the scope of optimisation. The MRA framework has also been used for other optimisations, *viz.* placement of load and store instructions in register optimisation [18].

Example 1.2 : Figure 1.3 presents the data flow equations of LSIA and CHSA which are used for register assignment and strength reduction optimisations respectively. In the Load-Store Insertion Algorithm [18], the problem of placing Load and Store instructions of a variable to characterise its live range for register assignment is modelled as redundancy elimination of the Load and Store instructions. Here, we consider the problem of the placement of Store instructions, which is a dual of MRA. This problem performs sinking of STORE instructions using partial redundancy elimination techniques [18].

The Composite Hoisting and Strength Reduction Algorithm (CHSA) [34, 35] unifies strength reduction and redundancy elimination into the same framework. *Additive* computations are used to update the value of a high strength expression following an update of the induction

- **The Basic Load Store Insertion Algorithm (LSIA) [18]**

$$
\begin{aligned}
\text{SPPIN}_i &= \prod_{j \in pred(i)} (\text{SPPOUT}_j) \\
\text{SPPOUT}_i &= \text{DPANTOUT}_i \cdot (\text{DCOMP}_i + \text{DTRANSP}_i \cdot \text{SPPIN}_i) \cdot \\
&\quad \prod_{k \in succ(i)} (\text{DANTIN}_k + \text{SPPIN}_k)
\end{aligned}
$$

- **Composite Hoisting and Strength Reduction Algorithm (CHSA) [34, 35]**

$$
\begin{aligned}
\text{NOCOMIN}_i &= \text{CONSTA}_i \cdot \text{NOCOMOUT}_i + \\
&\quad \sum_{j \in pred(i)} \text{CONSTB}_i \cdot \text{NOCOMOUT}_j \\
\text{NOCOMOUT}_i &= \text{CONSTC}_i + \text{CONSTD}_i \cdot \text{NOCOMIN}_i + \\
&\quad \sum_{k \in succ(i)} \text{CONSTE}_i \cdot \text{NOCOMIN}_k
\end{aligned}
$$

Figure 1.3: Data flow equations of some other bidirectional problems

variable's value. Recomputations of the high strength expression are placed at other strategic places in the program. The NOCOMIN_i/NOCOMOUT_i problem is used to limit the number of additive computations along a path following a recomputation of the high strength expression. □

1.3 Incremental Data Flow Analysis

1.3.1 The Need of Incremental Data Flow Analysis

A program undergoes changes during development and during compilation since one optimisation often leads to others. Thus repeated applications of an optimising transformation may be required. The issue of an optimal order of performing different optimisations is NP-hard. However, even a single optimisation may enhance the scope for further optimisation.

Example 1.3 : Consider the optimisation of a statement $x := i + j * 5$ occurring in a basic block of a program graph. In the intermediate representation, this statement is represented as follows :

$$\begin{array}{|lcl|}\hline t_1 & \leftarrow & j*5 \\ t_2 & \leftarrow & i+t_1 \\ x & \leftarrow & t_2 \\ z & \leftarrow & x \\ \hline\end{array}$$

Even if the basic block is part of a loop, and the entire large expression $i+j*5$ is invariant in the loop, only the quadruple $t_1 \leftarrow j*5$ moves out of the loop in the first application of the optimisation algorithm. Movement of $t_2 \leftarrow i+t_1$ can not be performed in the same application, because of the presence of $t_1 \leftarrow j*5$ preceding it in the same block. Movement of $z := x$ can only be performed after the assignment to x has been optimised. Thus, multiple passes of optimisation are necessary for a good quality optimisation. □

Clearly, it is not desirable to perform repeated optimisations if each application of an optimisation recomputes the information from scratch. Instead, it is preferable to *update* the information computed earlier by an optimisation to reflect the changes caused by the same, or some later optimisation. Thus, the flow analysis overheads for the second and every subsequent application of a transformation are reduced considerably. This provides valuable savings in the optimisation costs.

1.3.2 Traditional Work in Incremental Data Flow Analysis

The development in incremental data flow analysis has been influenced more by its need in the context of syntax directed editing than its need in the context of compiling. Consequently, it has been viewed mostly as an offshoot of the exhaustive analysis. Though incremental data flow analysis has received a considerable attention [9], most of the work reported in the literature consists of isolated techniques and there is little or no theoretical treatment independent of the technique being proposed. Despite some common trends, the treatment of incremental data flow analysis has been ad hoc in that all discussions remain specific to *a* technique. More specifically, the issue of providing an algorithm-independent definition of the *incremental solution* (i.e. the change in the old solution), has almost always been side-tracked by utilising the definitions of the final solutions. Consequently, the issues of correctness and generality (in terms of applicability to different data flow problems) have been adversely affected. Besides, almost all approaches are restricted to unidirectional data flow problems and the issue of incremental algorithms for bidirectional problems has remained un-addressed.

1.4 Scope of Work

1.4.1 Current Status of Data Flow Analysis

There have been tremendous developments in the theory and applications of data flow analysis over past two decades [32, 46]. It has gradually progressed from the original domain of code optimisation to include static semantics [1, 28, 29, 48], error recovery in parsers [5], abstract interpretation [16] etc. and is more generally viewed as a theory of discrete dynamic systems [15]. The applications to other domains, though based on rigourous theory, have their own limitations; Rosen [51] does a good job of warning against some pitfalls.

In any case, these developments remain orthogonal from the viewpoint of traditional compiler writing for imperative languages : All of them inherit the same basic limitation – unidirectional dependencies. One reason for the lack (or rarity) of research from this perspective could be attributed to the practical performance of MRA which seems to be much better than the intuitively expected notion of the performance of a general bidirectional problem.[1] However, the recent additions to the class of bidirectional problems warrant serious explorations; indeed several contemporary efforts have been reported [24, 25, 40].

1.4.2 Scope and Contributions of this Research

This research was motivated by a primary goal of explorations in bidirectional data flows, though a secondary, but probably equally important goal has been a uniform treatment of unidirectional and bidirectional flows.

The results presented in this work are applicable to all *bit vector data flow problems* — most practical data flow problems fall under this category. We also introduce the notion of the *separability of solutions* to define a larger class of the data flow frameworks which can be explained by the generalised theory with minor extensions. The results in the incremental data flow analysis are, however, restricted to bit vector data flow problems only.

The major contributions of this work are :

1. Formal characterisation of the notions in exhaustive and incremental data flow analysis (viz. information flow paths, influence of incremental changes in flow functions, dependence of data flow properties on other data flow properties etc.).

2. Formal models for exhaustive and incremental data flow analysis to :

[1] It has been observed that MRA rarely needs more than 5 iterations [34, 35, 45].

- define the exhaustive and incremental solutions of data flow problems.
- show the correctness of exhaustive and incremental solutions.

3. Generic worklist based iterative algorithms for performing exhaustive and incremental data flow analysis.

4. Significant findings in complexity of exhaustive data flow analysis :

   - *Worklist based iterative data flow analysis* - We show that the complexity of unidirectional and bidirectional data flow analysis is same.
   - *Round robin iterative data flow analysis* - We define the notion of *width* which provides the first (strict) bound on the number of iterations for bidirectional data flow analysis. For the unidirectional problems, the width provides a more accurate bound than the traditional measure of *depth*.

5. Motivation and explanation of efficient solution techniques for exhaustive data flow analysis.

It may be emphasised that these results are uniformly applicable to unidirectional and bidirectional data flows. They unfold deep insights into the process of data flow analysis and provide a firm theoretical foundation for understanding (and predicting) the behaviour of various data flow problems making it easier to devise and experiment with more unified optimising transformations.

1.4.3 An Outline

The thesis is divided into three parts. The first part deals with exhaustive data flow analysis while the second part deals with incremental data flow analysis. The two parts have been made almost independent by presenting a brief review of those concepts from part 1.4.3 which are used in part 6.7. The third (rather short) part crisply summarises the results and puts them in the perspective of some philosophical musings.

Chapter 2 – 6 constitute the first part. Chapter 2 reviews the notions from classical data flow analysis and discusses the limitations of the classical theory. Chapter 3 eliminates these limitations by proposing a formal and a more general notion of *information flow*. Chapter 4 presents a generic algorithm for exhaustive data flow analysis and shows its correctness by defining a *state transition model* which provides an alternative definition of the MFP solution.

This chapter also analyses the performance of the generic algorithm and shows that the complexity of unidirectional and bidirectional data flow analysis is same. Chapter 5 provides an efficient adaptation of the generic algorithm for MRA and discusses the experimental results. Chapter 6 applies the generalised theory to complexity of data flow analysis and proposes the first (strict) bound on the number of iterations of round robin analysis of bidirectional flows.

Chapter 7 – 12 constitute the second part. Chapter 7 reviews the traditional approaches to incremental data flow analysis and discusses their limitations. Chapter 8 summarises those concepts of part 1.4.3 which are used in part 6.7 thereby making the two parts as independent as possible. Chapter 9 presents a *functional model* for incremental data flow analysis which facilitates a formal definition of the change in the old solution. Chapter 10 develops a generic algorithm for incremental data flow analysis which faithfully implements the definitions provided in chapter 9. Chapter 11 goes a step further by modifying the algorithm developed in chapter 10 to incorporate wordwise processing of bit vectors as against original bitwise processing. Chapter 12 demonstrates the correctness of the functional model.

The third part consists of a single chapter which provides a summary of the results.

Appendix A presents a procedure to compute the regular expressions representing the information flow paths. Appendix B contains the data on empirical performance of the solution procedure for MRA presented in chapter 5 while appendix C contains the empirical data relating the width to the number of iterations for round robin data flow analysis. Appendices D and E present the empirical data about the performance of the incremental data flow analysis algorithms.

Part I

Exhaustive Data Flow Analysis

Chapter 2

Classical Data Flow Analysis

"But where do you think he would go?"
"Anywhere. Straight ahead of him."
Then the little prince said earnestly :
"That doesn't matter. Where I live, everything is so small!"
And, with perhaps a hint of sadness, he added :
"Straight ahead of him, nobody can go very far ..."

This chapter presents an overview of the classical theory of data flow analysis and compares various solution methods and their complexities. Our description is based mostly on [31, 32, 44]; a more detailed treatment can be found in [2, 31, 32, 36, 39, 44, 52]. The concluding part of this chapter motivates the need for a more general setting.

2.1 Data Flow Frameworks

A data flow framework is defined as a triple $\mathbf{D} = < \mathcal{L}, \sqcap, \mathcal{F} >$ (Figure 2.1). Elements in $\mathcal{L}$ represent the information associated with the entry/exit of a basic block. Thus, each element in $\mathcal{L}$ is a set of facts associated entry/exit of a basic block. $\sqcap$ is the set union (alternatively, boolean SUM denoted by Σ) or intersection (alternatively, boolean PRODUCT denoted by Π) operation which determines the way the global information is combined when it reaches a basic block. A function $f_i \in \mathcal{F}$ represents the effect on the information as it flows through basic block i.[1]

A data flow framework is characterised by any or all of the following :

[1] Alternatively, the functions can be associated with in-edges(out-edges) of node i for forward(backward) flow problems.

Data Flow Framework : D = $< \mathcal{L}, \sqcap, \mathcal{F} >$, where

▷ $< \mathcal{L}, \sqcap >$ is a semi-lattice such that :

- $\mathcal{L}$ is a partially ordered set (often finite).
- $\sqcap$ is a binary meet operation which is commutative, associative, and idempotent.
- The partial order (denoted $\sqsubseteq$) is reflexive, antisymmetric, and transitive.
  $\forall\, a, b \in \mathcal{L} : a \sqsubseteq b$ *iff* $a \sqcap b = a$
- There are two special elements *top* (denoted $\top$) and *bottom* (denoted $\bot$).[a]
  $\forall\, a \in \mathcal{L} : a \sqcap \top = a$
  $a \sqcap \bot = \bot$
- $\mathcal{L}$ has finite height. (i.e. length of every strictly descending chain $a \sqsubset b \sqsubset \cdots \sqsubset z$ is finite).
  If the length of every strictly descending chain is bounded by a constant, say H, we say that $\mathcal{L}$ has *strictly finite height*, or simply *height* H).

▷ $\mathcal{F} \subseteq \{f : \mathcal{L} \to \mathcal{L}\}$ is a class of functions such that :

- $\mathcal{F}$ contains an identity function ι.
  $\forall\, a \in \mathcal{L},\ \iota(a) = a$
- $\mathcal{F}$ is closed under composition.
  $\forall\, f_1, f_2 \in \mathcal{F} :\ f_1 \circ f_2 \in \mathcal{F}$
- $\forall\, a \in \mathcal{L},\ \exists f \in \mathcal{F}$ such that $a = f(\bot)$

▷ **D** is *monotone* if and only if :

$\forall\, a, b \in \mathcal{L}, \forall\, f \in \mathcal{F} :\ f(a \sqcap b) \sqsubseteq f(a) \sqcap f(b)$. This is same as

$$a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$$

▷ **D** is *distributive* if and only if :

$\forall\, a, b \in \mathcal{L}, \forall\, f \in \mathcal{F} :\ f(a \sqcap b) = f(a) \sqcap f(b)$

▷ **D** is *k-bounded* if and only if :

$$\forall\, f \in \mathcal{F},\ \exists k :\ \prod_{i=0}^{k-1} f^i = f^*, \text{ where } f^{j+1} = f \circ f^j,\ f^0 = \iota, \text{ and } f^* \equiv f^0 \sqcap f^1 \sqcap f^2 \sqcap \cdots$$

Instance of a Data Flow Framework : I = $< G, M >$, where

▷ $G = < N, E, n_0 >$ is a control flow graph where N is the set of nodes representing basic blocks, E is the set of edges, and n_0 is a unique entry node with in-degree zero.

▷ $M : N \to \mathcal{F}$.It is extended to paths as follows :

- If $\rho = (n_0, n_1, \cdots, n_i)$ is a *path* in G then
  $M(\rho) = M(n_0) \circ M(n_1) \circ \cdots \circ M(n_{i-1})$
- If ρ is a null path then $M(\rho)$ is an identity function.

[a] In some cases either $\top$ or $\bot$ may not exist. However, it can always be added artificially. It may not be a natural element of $\mathcal{L}$ but it helps in performing data flow analysis.

Figure 2.1: Data Flow Framework

- Algebraic properties of functions in $\mathcal{F}$ (*viz.* monotonicity, distributivity, continuity, etc. [32]).
- Finiteness properties of functions in $\mathcal{F}$ (*viz.* boundedness [44], fastness [31], rapidity, [36] etc.).
- Finiteness properties of $\mathcal{L}$ (*viz.* height [32, 44]).
- Partitionability properties of $\mathcal{L}$ and $\mathcal{F}$ [63].

There is an important subclass of k-bounded partitionable problems[2] called the *bit vector problems* [32] which has been extensively discussed in the literature [25, 32, 44, 46], though it is defined only informally (*viz.* in [32, 63]). We provide a formal definition in section 3.1.3 and use it in the exposition of our theory.

2.2 Data Flow Equations

To formulate a data flow problem, the data flow properties associated with each node of the flow graph are represented as variables which, as noted earlier, are elements in $\mathcal{L}$. Interdependencies of the values of these variables give rise to simultaneous equations. Thus, solving a data flow problem reduces to solving a system of simultaneous equations.

A data flow problem is posed as a pair $<Q, X_0>$, where Q is a system of equations parameterised by the nodes of the flow graph and whose terms may include constants. These constants may represent information derived from other data flow problems. $X_0 : N \to \mathcal{L}$ is a conservative initialisation. For the entry node it is usually, though not always, $\perp$. For the non-entry nodes, such an estimate is almost always $\top$ and is needed in the case of iterative methods only.

Let $pred(i)$ and $succ(i)$ denote the set of predecessors and successors of node i. The equations $X = Q(Y)$ have the following form[3] :

$$\text{IN}(i) = \begin{cases} X_0(n_0) & \text{if } i = n_0 \\ \displaystyle\bigsqcap_{j \in pred(i)} M(j)(\text{IN}(j)) & \text{otherwise} \end{cases} \tag{2.1}$$

[2]We use the terms *data flow problem* and *data flow framework* interchangeably, though the latter is more formal.

[3]Though we present the definitions for forward problems only, analogous definitions exist for backward problems.

Note that the equations may well be written in terms of information at the node exit [56]. Alternatively, both IN and OUT may be used. Further, the function M may be dropped and the node numbers may be used as subscripts of $f \in \mathcal{F}$ as shown below.

$$\mathrm{IN}_i = \begin{cases} X_0(n_0) & \text{if } i = n_0 \\ \displaystyle\prod_{j \in pred(i)} (\mathrm{OUT}_j) & \text{otherwise} \end{cases} \tag{2.2}$$

$$\mathrm{OUT}_i = f_i(\ \mathrm{IN}_i\ ) \tag{2.3}$$

We use this form in the thesis.

2.3 Solutions of a Data Flow Problem

The solution of a data flow problem is an assignment of values $X\ : N \to \mathcal{L}$ to the nodes of the flow graph.

2.3.1 Safe Solution

An assignment SA is *safe* if the information at a node does not exceed the information that can be gathered along any path from n_0 to that node [31], i.e.

$$\forall\, i \in N\ :\ \mathrm{SA}(i) \sqsubseteq M(\rho)(X_0(n_0))$$

where ρ is a path from n_0 to i. A safe assignment guarantees the correctness of optimisations; an unsafe assignment may result in semantics changing optimisations.

The *Meet Over Paths* solution of a data flow problem represents the information reaching a basic block along *all possible* program paths [2, 32, 44]. Let $paths(i)$ denote the set of all paths from n_0 to i. Then,

$$\forall\, i \in N\ :\ \mathrm{MOP}(i) = \prod_{\rho \in paths(i)} M(\rho)(X_0(n_0))$$

Note that MOP is the maximum safe assignment.

2.3.2 Fixed Point Solution

An assignment FP is a *fixed point* of an instance of a data flow framework if it is a fixed point of equation 2.1.

A fixed point guarantees the consistency of information associated with the nodes of the flow graph. A *Maximum Fixed Point*, MFP, contains all other fixed points. It can be shown that MFP is contained in MOP. Thus, every fixed point is a safe assignment, though not vice-versa.

An assignment X is *acceptable*[4] if and only if it is safe and contains all fixed points of Q. For the monotone data flow problems, MOP and MFP typically exist. For the distributive problems, MFP is always equal to MOP. Since MFP represents the maximum information that can be gathered in practice, *the goal of data flow analysis can also be defined as finding MFP*.[5] Though this means that we may not be able to capture all information for the non-distributive problems, it does not matter since no algorithm capable of computing MOP for all instances of arbitrary monotone data flow problems exists anyway [36].

2.4 Performing Data Flow Analysis

There are two broad categories of the approaches to data flow analysis : *iterative* methods and *elimination* methods.

2.4.1 Iterative Methods

The iterative method of data flow analysis solves the system of equations by initialising the node variables to some conservative values and recomputing them successively till a fixed point is reached. The *round robin* version (traced back to [62]) recomputes the data flow properties of all the nodes repeatedly, till the values stabilise. If the size of the bit vector is r, there are $O(n \cdot r)$ properties. Thus, in the worst case $O(n \cdot r)$ iterations over the graph may be needed. Each iteration involves computation of the properties for n nodes. If all the r bits can be processed in one step, the complexity becomes $O(n^2 \cdot r)$. In the unidirectional problems the flow is in one direction only, hence the nodes can be visited in the postorder or reverse postorder depending upon the direction of flow. Thus, $d+2$ iterations are sufficient where, d is the depth[6] of the flow graph [2, 32]. Hence the complexity is $(d+2) \cdot n$. The *worklist* version visits the nodes selectively and involves $O(n \cdot r)$ work [32].

For programs with $r = O(n)$, all the r bits can not be processed in one step; processing the r bits of a bit vector would itself require $O(n)$ steps. Hence the bounds on the iterative methods are $O(n^4)$ and $O(n^2)$.

---

[4] Marlowe & Ryder [44] use this term for a slightly different notion; we follow Graham & Wegman [31].

[5] In some cases the desired information may be different from MFP. For example in the case of *hoisting-though-loop* effect [24], the desired fixed point lies below MFP.

[6] Not to be confused with the *nesting depth*.

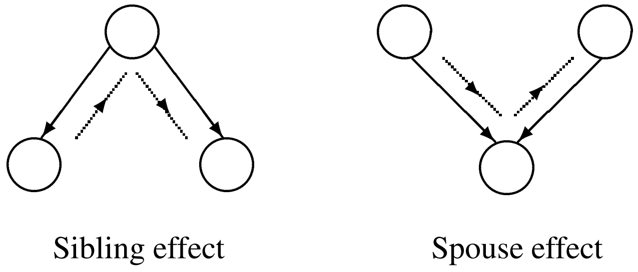


Figure 2.2: Sibling and spouse effects

2.4.2 Elimination Methods

Elimination methods reduce the amount of effort required to solve a data flow problem by utilising the structural properties of a flow graph [3, 31, 32, 46, 60]. The flow graph is reduced to one node by successive applications of graph transformations which use *graph parsing* or *graph partitioning* to identify regions to obtain a derived graph. The data flow properties of a node in a region are determined from the data flow properties of the region's header node. This enables delayed substitution of some values in the simultaneous equations. For unidirectional flow problems, these methods are typically $O(\mathcal{N})$, where $\mathcal{N}$ is the total number of nodes in the sequence of reduced graphs. A comparison of various elimination methods appears in [56]. It has been shown that the elimination methods cannot be extended to general bidirectional data flow problems, though they have been used to solve a restricted class of bidirectional problems [24].

For programs with $r = O(n)$ the elimination methods are $O(\mathcal{N} \cdot n)$.

2.5 Limitations of the Classical Theory

The limitations of the classical theory of data flow analysis are easy to trace. It is based on strictly unidirectional flow — information reaches one end of a basic block, flows through it, and emanates from the other end. As a consequence, the information flows from a node either to its predecessors or its successors.

In bidirectional problems, apart from the above flows, the following kinds of information flow may exist (refer to Figure 2.2) :

- information at one successor of a node may influence the information at another successor of the same node, and

- information at one predecessor of a node may influence the information at another predecessor of the same node.

We term these as the *sibling effect* and the *spouse effect* respectively. (Two nodes are *siblings* if they have a common predecessor; we call them *spouses* if they have a common successor.)

Example 2.1 : In example 1.1, $\text{ANTLOC}_9^l = \text{TRANSP}_9^l = \mathbf{F}$, consequently PPIN_9^l becomes $\mathbf{F}$. This makes $\text{PPOUT}_8^l = \mathbf{F}$ which sets $\text{PPIN}_{11}^l = \mathbf{F}$. This flow from the entry of 9 to the entry of 11 is an example of the sibling effect. Since PPOUT_8^l is $\mathbf{F}$, PPOUT_{10}^l also becomes $\mathbf{F}$ via PPIN_{11}^l; this is an example of the spouse effect. □

Such flows just can not arise in unidirectional data flow problems and it is not surprising that the traditional theory fails to characterise them. It follows that any characterisation which can not handle these flows gracefully, will at best be an isolated and ad hoc attempt at explaining bidirectional data flows [17, 24, 25, 40]. The present work overcomes this obstacle by proposing a more refined notion of information flow which handles the sibling/spouse effects elegantly and thereby provides a generalised theory which handles unidirectional and bidirectional problems uniformly.

Chapter 3

A Generalised Theory of Data Flow Analysis

Perhaps you will ask me, "Why there are no other drawings in this book as magnificent and impressive as this drawing of the baobabs?"
The reply is simple. I have tried. But with others I have not been successful. When I made the drawing of the baobabs I was carried beyond myself by the inspiring force of urgent necessity.

This chapter defines the scope of the theory and generalises the traditional concepts to handle unidirectional and bidirectional data flow problems uniformly. Section 3.1.3 defines the bit vector problems by formalising the notion of separability of solutions. Section 3.2 generalises the notions of edge, node, and path flow functions by defining the notion of information flow which captures the manner in which the information could flow, including the complex flows which arise when the spouse/sibling effects are present. Section 3.3 proposes generic data flow equations which facilitate a uniform specification of data flow problems by parameterising the equations appropriately. Finally, section 3.4 characterises the safety of assignment.

3.1 Preliminary Concepts

3.1.1 Traversals

We define a flow graph by $G = < N, E, entry(G), exit(G) >$ where $entry(G)$ and $exit(G)$ denote the (non-null) sets of entry and exit nodes, i.e. nodes with zero in-degree and zero out-degree, respectively.

A *program point* refers to the entry/exit of a basic block. For a basic block i, its entry and exit points are denoted by $in(i)$ and $out(i)$ respectively. Program point u is a *neighbour* of program point v if u and v are adjacent in G and the information at u is influenced by the information at v. Thus $in(j)$ is a neighbour of $out(i)$ where $j \in succ(i)$, and $out(j)$ is a neighbour $in(j)$ for a forward data flow problem.

Given a depth first spanning tree of G, we differentiate between *back edges* and non-back edges; we term the latter as *forward edges*.[1] Thus the term forward edges, as used in this thesis, includes the conventional notions of forward as well as cross edges [2]. A *forward traversal* along an edge is a tail-to-head traversal of the edge, while a *backward traversal* is a head-to-tail traversal. We use the following notations :

- T_e^f/T_e^b: Forward/backward traversal along an edge.
- T_f^f/T_f^b: Forward/backward traversal along a forward edge.
- T_b^f/T_b^b: Forward/backward traversal along a back edge.

A forward edge traversal indicates traversal *along* the direction of control flow whereas a backward edge traversal indicates a traversal *against* the direction of control flow.

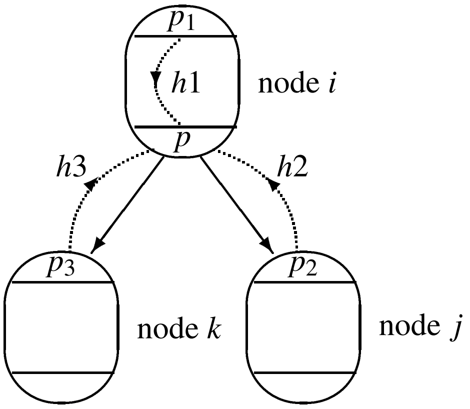
3.1.2 Data Flow Information and Data Flow Properties

When the sets of information are implemented as bit vectors, each bit represents a data flow property. There is one bit vector for the entry and one for the exit of each node. The lattice elements $\top$ and $\perp$ are "all bits true" (denoted $\overline{\mathrm{T}}$) and "all bits false" (denoted $\overline{\mathrm{F}}$) , or vice-versa, depending on $\sqcap$. When $\sqcap$ is Π, $\top$ is $\overline{\mathrm{T}}$ while $\perp$ is $\overline{\mathrm{F}}$; the situation is exactly opposite in when $\sqcap$ is Σ. TOP is the value of an individual property in a bit vector which represents the $\top$ element of the lattice while BOT is the value of a property in the bit vector which represents the $\perp$ element.

The data flow properties associated with program points $in(i)$ and $out(i)$ are denoted by IN_i and OUT_i respectively.

Definition 3.1 : Inflow/outflow Properties
Data flow properties that represent the information reaching/emanating from a node are called

[1] We use the terms *forward edges* and *back edges* as synonyms of *advancing edges* and *retreating edges* respectively. We prefer the former because they express the intuitive notion of direction more clearly. For non-reducible flow graphs, a back edge in this thesis means a retreating edge.



$$
\begin{aligned}
&\mathcal{P}p(p) = out(i),\;\; \mathcal{P}p(p_1) = in(i)\\
&\mathcal{P}p(p_2) = in(j),\;\; \mathcal{P}p(p_3) = in(k)\\
&p \leftarrow h1(p_1),\; p \leftarrow h2(p_2),\; p \leftarrow h3(p_3)\\
&\mathcal{N}^{-1}(p) = \{p_1, p_2, p_3\}\\
&p \in \mathcal{N}(p_1),\; p \in \mathcal{N}(p_2),\; p \in \mathcal{N}(p_3)\\
&\mathcal{N}^{-1}(out(i)) = \{in(i), in(j), in(k)\}\\
&out(i) \in \mathcal{N}(in(i)), out(i) \in \mathcal{N}(in(j))
\end{aligned}
$$

Figure 3.1: Several functions may influence a property.

inflow/outflow properties.

The inflow/outflow properties are associated with the entry/exit of a node depending upon the direction of the flow. From equations 2.2 and 2.3, it is evident that for a forward problem IN_i represents the inflow properties while OUT_i represents the outflow properties of node i. However, for a backward problem, OUT_i represents the inflow properties whereas IN_i represents the outflow properties.

Example 3.1 : For the problem of Reaching Definitions, REACH_IN_i represents the inflow properties while REACH_OUT_i represents the outflow properties. However, in the case of Live Variables, LIVE_OUT_i represents the inflow properties and LIVE_IN_i represents the outflow properties. Note that for bidirectional problems (viz. MRA), the same property may be an inflow as well as an outflow property. For instance, PPIN_i^l may become **F** due to PPOUT_j^l where $j \in pred(i)$, thus PPIN_i^l represents an inflow property. On becoming **F**, PPIN_i^l may cause $\text{PPOUT}_{j'}^l$ to become **F**, for some predecessor j'; here PPIN_i^l represents an outflow property. □

The program point for a property p is denoted by $\mathcal{P}p(p)$. Two properties belonging to different program points are called *corresponding* properties if they represent information about the same data item, *viz.* the same variable or the same expression. By definition, the relation of correspondence is reflexive, i.e. a property corresponds to itself. Two corresponding properties are *neighbours* of each other if their program points are neighbours in G.

If a property p' influences the value of a neighbouring property p through a flow function h, it is denoted by $p \leftarrow h(p')$. Clearly, $p \in \mathcal{N}(p')$ and $p' \in \mathcal{N}^{-1}(p)$. Whenever the program points of the two properties are required along with function, we write h as $h_{(u',u)}$ where

$\mathcal{P}p(p)$ is u and $\mathcal{P}p(p')$ is u'. The notation describing neighbourhood is also extended to the corresponding program points u and u'. Thus, $u \in \mathcal{N}(u')$ and $u' \in \mathcal{N}^{-1}(u)$. The order (u', u) determines the direction of the flow. If p and p' belong to neighbouring nodes i and j, p is an inflow property of i while p' is the corresponding outflow property of j. If p and p' belong to the same node, p is an outflow property whereas p' is the corresponding inflow property. Figure 3.1 contains an illustration of of these notions.

3.1.3 Bit Vector Frameworks

Definition 3.2 : Separability of Solutions[2]
A data flow framework $\mathbf{D} = < \mathcal{L}, \sqcap, \mathcal{F} >$ *possesses the property of the separability of solutions if* $\exists$ *semilattices* $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_n$ *such that an element* $X \in \mathcal{L}$ *can be represented by a tuple* $< X^1, X^2, \ldots, X^n >$ *where* $X^i \in \mathcal{L}_i$, $1 \leq i \leq n$ *and :*

1. $\forall\, X, Y \in \mathcal{L} : X \sqcap Y \ \equiv < X^1 \sqcap Y^1, X^2 \sqcap Y^2, \ldots, X^n \sqcap Y^n >$

2. $\exists\, h^i : \mathcal{L}_i \to \mathcal{L}_i, 1 \leq i \leq n$ *such that* $\forall\, h \in \mathcal{F}$, $h(X) \equiv <h^1(X^1), h^2(X^2), \ldots, h^n(X^n)>$

3. *The height of each* $\mathcal{L}_i$ *is bounded by a constant.*

Elements in each $\mathcal{L}_i$ represent different values of a *single* data flow property. The first two conditions ensure the independence of data flow properties while the third condition ensures the number of distinct values of a property is bounded by a constant. Let the smallest bound on the height of all $\mathcal{L}_i$ be H. Then, a property may have at most $H+1$ distinct values. Note that the separability of solutions implies that a *factorisation* exists for $(\mathcal{L}, \mathcal{F})$ [52], the *effective* height of L is H [52], and the functions in $\mathcal{F}$ are $(H+1)$-bounded [44].

It is easy to see that the bit vector problems satisfy all the three conditions : Data flow properties, represented by single bits, are independent of each other and each property may have two distinct values. Define bit functions *start*, *stop*, *propagate*, and *negate* such that for a bit b, $start(b) = \mathbf{T}$, $stop(b) = \mathbf{F}$, $propagate(b) = b$, $negate(b) = \neg b$ [25]. Let X^i be the i^{th} bit in a bit vector X. Let the size of the bit vector be k.

Definition 3.3 : Bit vector function
A bit vector function h *is a mapping* $\{\, \mathbf{T}, \mathbf{F} \,\}^k \to \{\, \mathbf{T}, \mathbf{F} \,\}^k$ *such that* h *can be written as a tuple of bit functions* $h \equiv < \mathcal{B}_h^1, \mathcal{B}_h^2, \ldots, \mathcal{B}_h^k >$ *where* $\mathcal{B}_h^i$ *is the bit function for the* i^{th} *bit, i.e. if*

[2] This notion is analogous, though not identical, to Zadeck's notion of *cluster partitionability* [63].

$Y = h(X)$, then

$$\begin{aligned} X &\equiv\ <X^1, X^2, \ldots, X^k> \\ Y &\equiv\ <Y^1, Y^2, \ldots, Y^k> \quad \textit{where},\ Y^i \equiv \mathcal{B}_h^i(X^i),\ 1 \le i \le k. \end{aligned}$$

Lemma 3.1 : *A bit vector function h is monotonic if and only if it does not negate any bit.*
Proof : A data flow framework is monotone if and only if all functions h satisfy the following :

$$\forall X, Z \in \mathcal{L} \ : \ X \sqsubseteq Z \Rightarrow h(X) \sqsubseteq h(Z) \tag{3.1}$$

Consider $X, Z \in \mathcal{L}$[3] such that $X \sqsubseteq Z$. The three possibilities for the ordered pair $<X^i, Z^i>$ are <TOP, TOP>, <BOT, TOP> and <BOT, BOT>. Consider a bit vector function h such that for all bits i, $\mathcal{B}_h^i \in \{start, stop, propagate\}$. The result of application of h to X and Z is :

$<X^i, Z^i>$	$<\mathcal{B}_h^i(X^i), \mathcal{B}_h^i(Z^i)>$		
	start	*stop*	*propagate*
<TOP, TOP>	<**T**, **T**>	<**F**, **F**>	<TOP, TOP>
<BOT, TOP>	<**T**, **T**>	<**F**, **F**>	<BOT, TOP>
<BOT, BOT>	<**T**, **T**>	<**F**, **F**>	<BOT, BOT>

In all the nine cases, $\mathcal{B}_h^i(X^i) \sqsubseteq \mathcal{B}_h^i(Z^i)$. Since this holds for every bit i, $h(X) \sqsubseteq h(Z)$.

For the converse, consider a bit vector function h which satisfies (3.1). To prove that it cannot negate a bit, assume that it has some $\mathcal{B}_h^i \equiv$ *negate*. Consider a pair $<X^i, Z^i>$ = <BOT, TOP>. The result of applying h to X and Z is $<\mathcal{B}_h^i(X^i), \mathcal{B}_h^i(Z^i)>$ which is <TOP, BOT>. Hence $\mathcal{B}_h^i(X^i) \not\sqsubseteq \mathcal{B}_h^i(Z^i)$, which is a contradiction. Hence h cannot have a *negate* bit function. □

Definition 3.4 : **Bit vector framework**
A data flow framework is bit vector framework if and only if all flow functions are monotonic bit vector functions.

Lemma 3.2 : *A bit vector function h is a monotonic bit vector function if and only if it can be expressed in the form $h(X) = C_1 + C_2 \cdot X$ where $C_1, C_2, X \in \{\ \mathbf{T}, \mathbf{F}\ \}^k$.*
Proof : Let $h \equiv\ <\mathcal{B}_h^1, \mathcal{B}_h^2, \ldots, \mathcal{B}_h^k>$ be a monotonic bit vector function. $h(X)$ can be expressed as $C_1 + C_2 \cdot X$ where $C_1 \equiv\ <C_1^1, C_1^2, \ldots, C_1^k>$ and $C_2 \equiv\ <C_2^1, C_2^2, \ldots, C_2^k>$. The bits C_1^i and C_2^i

[3]Note that for bit vector problems, $\mathcal{L}$ is essentially $\{\ \mathbf{T}, \mathbf{F}\ \}^k$.

can be set depending upon the bit function $\mathcal{B}_h^i \in \{start, stop, propagate\}$.

$\mathcal{B}_h^i$	C_1^i	C_2^i
start	**T**	*don't care*
stop	**F**	**F**
propagate	**F**	**T**

For the converse, let there be a function $h(X) = C_1 + C_2 \cdot X$. The values of the bits C_1^i and C_2^i define the corresponding bit function $\mathcal{B}_h^i$ as follows :

C_1^i	C_2^i	$\mathcal{B}_h^i$
T	**T**	*start*
T	**F**	*start*
F	**F**	*stop*
F	**T**	*propagate*

Since $\mathcal{B}_h^i \in \{start, stop, propagate\}$, from lemma 3.1 h is a monotonic bit vector function. □

Lemma 3.3 : *A bit vector framework is fast (i.e. 2-bounded).*

Proof : Obvious. □

The influence of p' on p through a function h is denoted by $p \leftarrow \mathcal{B}_h(p')$ where the context demands a bit function and by $p \leftarrow h(p')$ where the context demands a bit vector function.

3.2 Characterising the Flow of Information

3.2.1 The Notion of Information Flow

Since $x \sqcap \text{TOP} = x$ and $x \sqcap \text{BOT} = \text{BOT}$, a TOP value for a data flow property is an intermediate value until the data flow analysis is completed whereas BOT is a final value even during analysis. Thus, a BOT value implies a useful item of information from the viewpoint of data flow analysis, whereas a TOP value implies that such information can not be concluded during analysis. For iterative data flow analysis, the data flow properties are initialised to TOP for all nodes (except for the graph entry/exit nodes, which may have other values). Some properties change to BOT due to the local effect of computations in a node/along an edge, *viz.* when a definition is generated, or an expression is killed. These properties, in turn, change the neighbouring properties to BOT.

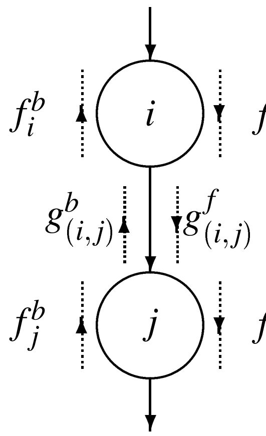


Figure 3.2: Flow functions

Definition 3.5 : **Information flow**
Information flows from a program point u to a program point v when a property at u, on becoming BOT, causes the corresponding property at v to become BOT.

Note that the information flow is transitive. Section 4.4 shows that the incorporation of information flows due to all BOT properties in the program flow graph leads to a fixed point of the data flow equations.

3.2.2 Node and Edge Flow Functions

There are two fundamental kinds of flows in a data flow analysis problem :

(i) Information flows *within* a node, i.e. between the entry and exit of the node :

Represented by *node flow* functions $f \in \mathcal{F}$. These are the traditional transfer functions.

(ii) Information flows along an edge :

Represented by *edge flow* functions g. We define a new set $\mathcal{G}$ to contain g.

The mapping between nodes and node flow functions is defined by the function $M_{\mathcal{F}} : N \rightarrow \mathcal{F}$ while the mapping between edges and edge flow functions is defined by the function $M_{\mathcal{G}} : E \rightarrow \mathcal{G}$. As is customary, we drop these mappings and subscript the flow functions directly by nodes/edges as the case may be. Thus, the node flow function for node i is denoted by f_i while the edge flow function for an edge $e = (i, j)$ is denoted by g_e or $g_{(i,j)}$.

$$
\begin{aligned}
\mathrm{PPIN}_i &= \mathrm{PAVIN}_i \cdot (\,\mathrm{ANTLOC}_i + \mathrm{TRANSP}_i \cdot \mathrm{PPOUT}_i) \cdot \\
&\qquad \prod_{j \in pred(i)} (\,\mathrm{AVOUT}_j + \mathrm{PPOUT}_j) \cdot \\
&\qquad \sum_{j \in pred(i)} (\,\mathrm{PPIN}_j \cdot \neg \mathrm{ANTLOC}_j + \mathrm{AVOUT}_j) \\
\mathrm{PPOUT}_i &= \prod_{k \in succ(i)} (\,\mathrm{PPIN}_k)
\end{aligned}
$$

Figure 3.3: The MMRA Equations [21]

The flow functions are determined directly from the data flow equations governing a problem. We refer to the functions by their type names f and g respectively, with an appropriate superscript f or b to indicate whether the flow is in the forward or the backward direction.

If a particular flow does not exist in a data flow problem, the corresponding function is the constant function $\top$. For example if forward flow of information *through* a node does not exist, f^f is $\top$. Similarly, if forward flow of information reaching a node entry (and hence the forward confluence of information) does not exist, g^f is $\top$. Analogous remarks hold for f^b and g^b.

Example 3.2 : Table 3.1 summarises the flow functions for some data flow problems. Note that there is no backward flow in the case of reaching definitions while there is no forward flow in the case of live variables. □

Definition 3.6 : **Non-singular data flow problem**
A data flow problem is non-singular if it involves more than one distinct confluence operator.

Clearly, a non-singular data flow problem is not a data flow framework as defined in Figure 2.1.

Example 3.3 : All examples considered in this thesis are singular except the Modified MRA (MMRA, for short) data flow problem [21] which, apart from the Π term of MRA, contains a Σ term to inhibit redundant code movement (Figure 3.3). □

3.2.3 Information Flow Paths

Let T_{e_i} denote a traversal along the graph edge e_i, i.e. T_{e_i} can be T_e^f/T_e^b.

- **Reaching Definitions**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) =$ REACH_GEN$_i$ + $\neg$ REACH_KILL$_i \cdot X$	REACH_OUT$_i$	REACH_IN$_i$
$f_i^b(X) = \top$	REACH_IN$_i$	REACH_OUT$_i$
$g^f_{(j,i)}(X) = X$ (i.e. identity function $\imath$)	REACH_IN$_i$	REACH_OUT$_j$
$g^b_{(i,k)}(X) = \top$	REACH_OUT$_i$	REACH_IN$_k$

- **Live Variables**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \top$	LIVE_OUT$_i$	LIVE_IN$_i$
$f_i^b(X) =$ LIVE_GEN$_i$ + $\neg$ LIVE_KILL$_i \cdot X$	LIVE_IN$_i$	LIVE_OUT$_i$
$g^f_{(j,i)}(X) = \top$	LIVE_IN$_i$	LIVE_OUT$_j$
$g^b_{(i,k)}(X) = X$ (i.e. identity function $\imath$)	LIVE_OUT$_i$	LIVE_IN$_k$

- **Morel-Renvoise Algorithm (MRA)**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) = \top$	PPOUT$_i$	PPIN$_i$
$f_i^b(X) =$ ANTLOC$_i$ + TRANSP$_i \cdot X$	PPIN$_i$	PPOUT$_i$
$g^f_{(j,i)}(X) =$ AVOUT$_j + X$	PPIN$_i$	PPOUT$_j$
$g^b_{(i,k)}(X) = X$ (i.e. identity function $\imath$)	PPOUT$_i$	PPIN$_k$

- **Basic Load Store Insertion Algorithm (LSIA)**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) =$ DCOMP$_i$ + DTRANSP$_i \cdot X$	SPPOUT$_i$	SPPIN$_i$
$f_i^b(X) = \top$	SPPIN$_i$	SPPOUT$_i$
$g^f_{(j,i)}(X) = X$ (i.e. identity function $\imath$)	SPPIN$_i$	SPPOUT$_j$
$g^b_{(i,k)}(X) =$ DANTIN$_k + X$	SPPOUT$_i$	SPPIN$_k$

- **Composite Hoisting and Strength Reduction Algorithm (CHSA)**

$h(X)$	$Y \leftarrow h(X)$	X
$f_i^f(X) =$ CONSTD$_i \cdot X$	NOCOMOUT$_i$	NOCOMIN$_i$
$f_i^b(X) =$ CONSTA$_i \cdot X$	NOCOMIN$_i$	NOCOMOUT$_i$
$g^f_{(j,i)}(X) =$ CONSTB$_i \cdot X$	NOCOMIN$_i$	NOCOMOUT$_j$
$g^b_{(i,k)}(X) =$ CONSTE$_i \cdot X$	NOCOMOUT$_i$	NOCOMIN$_k$

Table 3.1: Examples of Flow Functions

Problem	Function types	Information flow paths
Reaching Def.	f^f, g^f	$(T_e^f)^+$
Live Variables	f^b, g^b	$(T_e^b)^+$
MRA	f^b, g^b, g^f	$((T_e^b)^+(T_e^f \mid \varepsilon))^+$
LSIA	f^f, g^f, g^b	$((T_e^f)^+(T_e^b \mid \varepsilon))^+$
CHSA	f^f, f^b, g^b, g^f	$T_e^f(T_e^b \mid T_e^f)^*$

Table 3.2: Some examples of information flow paths

Definition 3.7 : Information flow path
An information flow path (*ifp*) *is a sequence of edge traversals* $T_{e_1}, T_{e_2}, \ldots, T_{e_k}$ *along which information can flow during data flow analysis.*

We use the notation $< u, v, \rho >$ for an *ifp* ρ from program point u to a program point v. Note that an *ifp* may not follow a graph theoretic path. Where convenient, we will represent an *ifp* as a sequence of nodes, leaving the traversal of the edges connecting these nodes implicit.

Since *ifp*'s can be statically determined from the flow functions, they follow a pattern which can be described by a regular expression. Appendix A provides a procedure to construct the regular expressions representing the *ifp* patterns. These regular expressions should be contrasted with Tarjan's *path expressions* [61]; the latter are restricted to graph theoretic paths and can not be used to characterise *ifp*'s arising out of sibling/spouse effects.

Example 3.4 : Table 3.2 contains examples of the *ifp* patterns. These patterns provide valuable insights about how the information could flow in a given data flow problem. □

Example 3.5 : Consider the graph in Figure 1.2. Some sequences of edge traversals which may form *ifp*'s, and the data flow problems in which these *ifp*'s are valid, are :

- $(1, 8, 11, 12) = T_f^f\ T_f^f\ T_f^f$: Reaching Definitions
- $(11, 10, 9, 8) = T_f^b\ T_f^b\ T_f^b$: Live variables
- $(5, 2, 3, 4, 7) = T_f^b\ T_f^f\ T_b^b\ T_f^f$: MRA
- $(7, 4, 3, 2) = T_f^b\ T_b^f\ T_f^b$: LSIA
- $(2, 5, 6, 7, 4, 3) = T_f^f\ T_f^f\ T_f^f\ T_f^b\ T_f^b$: CHSA

□

For bit vector frameworks, *ifp*'s are necessarily acyclic. Note, however, that the underlying graph theoretic path may be cyclic since a node may appear in the path once for its entry point and once for its exit point.

3.2.4 The Path Flow Function

Consider an *ifp* $< u, v, \mathcal{P}_r >$ from a graph entry/exit node to the entry/exit of a node r. For such an *ifp*, $u \in \{in(n_0), out(n_x)\}$ where $n_0 \in entry(G)$, $n_x \in exit(G)$, and $v \in \{in(r), out(r)\}$. Let $\mathcal{P}_r = q_1, q_2, \cdots, q_k, q_{k+1} = T_{e_1}, T_{e_2}, \cdots, T_{e_k}$. It follows that $q_1 \in \{n_0, n_x\}$ and $q_{k+1} = r$. Consider an *ifp* fragment $< u, v', \rho >$ of $\mathcal{P}_r$, terminating with edge $e_i = (q_i, q_{i+1})$, such that $v' = in(q_{i+1})$ if $T_{e_i} = T_e^f$ and $v' = out(q_{i+1})$ if $T_{e_i} = T_e^b$.

Let $flow_i$ denote the path flow function of ρ (i.e. the *ifp* terminating with e_i). We define

$$flow_1 = \begin{cases} g^f_{e_1} \circ f^f_{n_0} & \text{if } T_{e_1} = T_e^f \text{ (i.e. } u = in(n_0)) \\ g^b_{e_1} \circ f^b_{n_x} & \text{if } T_{e_1} = T_e^b \text{ (i.e. } u = out(n_x)) \end{cases}$$

Since $e_{i+1} = (q_{i+1}, q_{i+2})$, information flows from q_{i+1} to q_{i+2}. $flow_{i+1}$ is obtained by composing the functions $f_{q_{i+1}}$ and $g_{e_{i+1}}$ with $flow_i$, as shown in Table 3.3. Note that if $v = out(r)$ and $T_{e_k} = T_e^f$ then there is a forward flow through node r. Similarly, if $v = in(r)$ and $T_{e_k} = T_e^b$ then there is a backward flow through node r. Thus, path function for the *ifp* $\mathcal{P}_r$ is :

$$\text{FLOW}_{\mathcal{P}_r} = \begin{cases} f_r^f \circ flow_k & \text{if } T_{e_k} = T_e^f \text{ and } v = out(r) \\ f_r^b \circ flow_k & \text{if } T_{e_k} = T_e^b \text{ and } v = in(r) \\ flow_k & \text{otherwise} \end{cases}$$

In a unidirectional problem, the typical sequence of edges in a path is either $(T_e^f)^*$ or $(T_e^b)^*$. In either case, the information *necessarily* flows through all intermediate nodes. It can be verified from Table 3.3 that in such a case, the edge flow functions appear in composition with the node flow functions and never in isolation from them. Thus there is no need to treat them separately. Under such circumstances, the flow can be adequately characterised by functions which could be associated with the nodes or edges interchangeably. $\mathcal{F}$ has been the set of such functions in the classical theory.

However, in bidirectional problems the sibling/spouse effects exist and the information may flow from $in(q_i)$ to $in(q_{i+2})$ via $out(q_{i+1})$ or $out(q_i)$ to $out(q_{i+2})$ via $in(q_{i+1})$. Though the information flows along the edges $e_i = (q_i, q_{i+1})$ and $e_{i+1} = (q_{i+1}, q_{i+2})$, it does not flow *through* node q_{i+1}. Thus, unlike the unidirectional problems, the edge and node flows must be represented distinctly. It is easy to see that all the above flows are handled uniformly by the generalised path flow function.

	$flow_{i+1}$	
	$T_{e_{i+1}} = T_e^f$	$T_{e_{i+1}} = T_e^b$
$T_{e_i} = T_e^f$	$g^f_{e_{i+1}} \circ f^f_{q_{i+1}} \circ flow_i$ (Forward Flow)	$g^b_{e\,i+1} \circ flow_i$ (Flow between spouses)
$T_{e_i} = T_e^b$	$g^f_{e_{i+1}} \circ flow_i$ (Flow between siblings)	$g^b_{e_{i+1}} \circ f^b_{q_{i+1}} \circ flow_i$ (Backward Flow)

Table 3.3: Computing the flow function

3.3 Specification of a Data Flow Problem

A data flow problem is completely specified by the pair **S =** $< Q, X_0 >$, where Q is the system of equations and X_0 is the set of initial values for the properties of the nodes.

3.3.1 Data Flow Equations

The entry/exit properties of node i can be computed from various flows as follows :

$$\begin{aligned} \text{IN}_i &= \text{IN}_i^f \sqcap \text{IN}_i^b \sqcap \text{CONST\_IN}_i \\ \text{OUT}_i &= \text{OUT}_i^f \sqcap \text{OUT}_i^b \sqcap \text{CONST\_OUT}_i \end{aligned}$$

CONST_IN/CONST_OUT are the *constant* properties which represent the information already known concerning the entry/exit of a node. However, unlike the local properties of a node, the constant properties typically represent *lower order* data flow properties, i.e. properties computed by an earlier data flow analysis. If no such information is involved in the problem, the CONST_IN/CONST_OUT properties are $\top$.

Example 3.6 : For all unidirectional problems referred in this thesis, the CONST_IN and CONST_OUT properties are $\top$. However, for MRA, CONST_IN is PAVIN while CONST_OUT is $\top$. For CHSA, CONST_IN is $\top$ while CONST_OUT is CONSTC. □

$\text{IN}_i^f/\text{OUT}_i^f$ and $\text{IN}_i^b/\text{OUT}_i^b$ represent the contributions from the forward and backward flows respectively. Clearly, $\text{IN}_i^f/\text{OUT}_i^b$ represent the inflow while the $\text{IN}_i^b/\text{OUT}_i^f$ represent the outflow component of the entry/exit information of node i. They are computed as follows :

$$\begin{aligned} \text{IN}_i^f &= \bigsqcap_{j \in pred(i)} g^f_{(j,i)}(\text{OUT}_j) \\ \text{IN}_i^b &= f_i^b(\text{OUT}_i) \\ \text{OUT}_i^b &= \bigsqcap_{k \in succ(i)} g^b_{(i,k)}(\text{IN}_k) \\ \text{OUT}_i^f &= f_i^f(\text{IN}_i) \end{aligned}$$

Thus, the data flow equations become

$$\mathrm{IN}_i = \prod_{j \in pred(i)} g^f_{(j,i)}(\mathrm{OUT}_j) \sqcap f^b_i(\mathrm{OUT}_i) \sqcap \mathrm{CONST\_IN}_i \tag{3.2}$$

$$\mathrm{OUT}_i = \prod_{k \in succ(i)} g^b_{(i,k)}(\mathrm{IN}_k) \sqcap f^f_i(\mathrm{IN}_i) \sqcap \mathrm{CONST\_OUT}_i \tag{3.3}$$

Equations 3.2 - 3.3 are the generic data flow equations. Specific problems can be treated as special cases of these equations.

3.3.2 Initialisation

We define X_0 to consist of two classes of values : *Boundaryinfo* and *Initinfo*.

Boundaryinfo contains values specifying the inter-procedural information reaching the entry/exit of the graph. For an entry node, *Boundaryinfo* specifies the value associated with $in(i)$ while for an exit node, *Boundaryinfo* specifies the value associated with $out(i)$. These values are important for the correctness of any optimisation based on the solution of the data flow problem. A wrong specification may lead to an unsafe solution and may thus lead to an incorrect optimisation.

Let $Boundaryinfo_i$ denote the value associated with an entry/exit node i.

- $Boundaryinfo_i$ is $\top$, if either $i \in entry(G)$ and the forward confluence does not exist, or $i \in exit(G)$ and the backward confluence does not exist.
- If $i \in entry(G)$ and the forward confluence exists, or $i \in exit(G)$ and the backward confluence exists, $Boundaryinfo_i$ is determined by inter-procedural information if available, else, by the semantics of the data flow problem as explained in the following.

Let $\overline{\mathrm{T}}$ and $\overline{\mathrm{F}}$ denote "all bits **T**" and "all bits **F**" respectively. Table 3.4 provides *Boundaryinfo* for some representative data flow problems for local variables/expressions involving local variables. Consider the example of live variable analysis. For local variables, the values in *Boundaryinfo* are determined as follows : The information being gathered is a set of predicates, each of which represents that a variable is live. For local variables, all these predicates are false at the exit of a program, hence the values in *Boundaryinfo* are $\overline{\mathrm{F}}$ for the exit nodes. Since there is no forward flow, *Boundaryinfo* values for entry nodes are $\top$ which is $\overline{\mathrm{F}}$ for a union problem.

Initinfo specifies values for the internal nodes of the program flow graph. These are required in the case of iterative methods only. Using the confluence operator as a criterion, the *Initinfo* values are defined to be $\top$; anything else might lead to a fixed point lower than

Data Flow Problem	Direction	$\sqcap$	$\top$	$\perp$	*Boundaryinfo$_i$*	
					$i \in entry(G)$	$i \in exit(G)$
Reaching Definitions	Forward	OR	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$
Live Variables	Backward	OR	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$
Available Expressions	Forward	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$	$\overline{\text{T}}$
Very Busy Expressions	Backward	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{F}}$
Dead Variables	Backward	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{T}}$	$\overline{\text{T}}$
MRA	Bidirectional	AND	$\overline{\text{T}}$	$\overline{\text{F}}$	$\overline{\text{F}}$	$\overline{\text{F}}$

Table 3.4: *Boundaryinfo* values for local variables/expressions involving local variables.

MFP. Given correct *Boundaryinfo*, *Initinfo* influences the quality of information (vis-a-vis the maximality of information), but not its safety.

The distinction between *Boundaryinfo* and *Initinfo* is usually not made in the literature, leading to avoidable confusion. Although *Boundaryinfo* has no connection with the confluence operator, the values in *Boundaryinfo* are often recommended as $\perp$.[4] To correctly determine the *Boundaryinfo*, we only need to ask the following two questions : (i) What is the information being gathered ? and (ii) What is the information available from the caller procedure ?

Example 3.7 : Consider the problem of dead variable analysis which is a dual of the problem of live variable analysis. It is an intersection problem in which a predicate indicates that a variable is dead. For local variables, all predicates are true at the exit of a program. Hence the values in *Boundaryinfo* are $\overline{\text{T}}$ for the exit nodes.

This should be contrasted with the problem of Very Busy Expressions for which the *Boundaryinfo* values are $\overline{\text{F}}$ for the expressions involving local variables. Both the problems are backward intersection problems with an identical form of data flow equations, yet they have different *Boundaryinfo* values. □

Clearly, it is incorrect to link the confluence operator with *Boundaryinfo*.

[4][32] specifies $\perp$, [44] cautiously mentions "often $\perp$" while [2] specifies $\top$ with a remark that the values may be $\perp$ in some cases.

3.4 Solutions of Data Flow Problems

As noted in section 2.3, an acceptable solution is characterised by the safety and maximality of fixed point.

3.4.1 Safe Solution

Let $const(u)$ return the value of the constant property associated with program point u. For all p, p' and $h \in \mathcal{F} \cup \mathcal{G}$ such that $p \leftarrow h(p')$, we replace h in FLOW by $h \sqcap const(\mathcal{P}p(p))$. Let $< u, v, \mathcal{P}_r >$ be denoted by $\mathcal{P}_r^{in}$ if $v = in(r)$ and by $\mathcal{P}_r^{out}$ if $v = out(r)$. Further, let η denote *Boundaryinfo* at the program point u which belongs to a graph entry/exit node.

A safe assignment SA : $N \to \mathcal{L}$ is a function with two components, $<$ SIN, SOUT $>$ such that, for a node r, and all $\mathcal{P}_r^{in}$ / $\mathcal{P}_r^{out}$:

$$\mathrm{SIN}(r) \sqsubseteq \mathrm{FLOW}_{\mathcal{P}_r^{in}}(\eta) \tag{3.4}$$

$$\mathrm{SOUT}(r) \sqsubseteq \mathrm{FLOW}_{\mathcal{P}_r^{out}}(\eta) \tag{3.5}$$

Maximum SA represents the MOP solution. Any solution which is not contained in MOP is unsafe.

The *ifp*'s reduce to graph theoretic paths for unidirectional problems. For a forward unidirectional problem, $\mathcal{P}_r^{in}$ reduces to graph paths from $in(n_0)$ to $in(r)$ where n_0 is an entry node, and $\mathcal{P}_r^{out}$ does not exist. Thus, the path flow function $\mathrm{FLOW}_{\mathcal{P}_r^{in}}$ reduces to $M(\rho)$ defined in Figure 2.1 and used for characterising MOP solution in section 2.3. Analogous remarks hold for backward problems.

3.4.2 Fixed Point Solution

A fixed point solution is the fixed point of equations 3.2 - 3.3. Maximum fixed point is obtained by setting *Initinfo* values to $\top$.

For a forward unidirectional problem, the generic data flow equations reduce to equations 2.2 and 2.3. Analogous remarks hold for backward problems.

3.5 Looking Back

As promised towards the end of chapter 2, this chapter provides a neat and elegant characterisation to handle the sibling and spouse effects rather naturally. A closer look at the propositions in this chapter reveal that this powerful characterisation is based on an extremely simple idea : distinction between the node flows and the edge flows. This careful distinction has far reaching consequences, both on the theory and practice of data flow analysis.

Chapter 4

Performing Data Flow Analysis

> *"It may well be that this man is absurd. But he is not so absurd as the king, the conceited man, the businessman, and the tippler. For, at least his work has some meaning. When he lights his street lamp, it is as if he brought one more star to life, or one flower. When he puts out his lamp, he sends the flower, or the star, to sleep. That is a beautiful occupation. And since it is beautiful, it is truly useful."*

This chapter presents a worklist based generic algorithm for performing data flow analysis. Arising out of a generalised theory, this algorithm is uniformly applicable to unidirectional as well as bidirectional data flow problems. This unification establishes the fact that the bidirectional data flow problems are inherently no more complex than the unidirectional problems. Section 4.3 analyses the complexity of the algorithm and makes several suggestions to improve its practical performance. Section 4.4 establishes the correctness of the algorithm by defining the solution computed by the algorithm in terms of a state transition model, and by showing that the solution obtained is acceptable.

4.1 Characteristics of Data Flow Frameworks

Let $p \leftarrow \mathcal{B}_h(p')$. If p' is BOT, it may cause p to become BOT.

All bit vector problems have the following important characteristics :

- *MBVP* : A property changes from TOP to BOT *only*.
- *SBVP* : $\forall\ p', \forall\ h$ such that $p \leftarrow h(p')$, if p' causes p to become BOT, it does so *on its own and not in combination with other corresponding properties*.

These characteristics arise from monotonicity and singularity respectively :

(i) For convergence on MFP, *Initinfo* is $\top$. Since the functions involved are monotonic, if there is a change, it must be from TOP to BOT only.

(ii) Data flow frameworks have been defined in terms of a semilattice which implies a single confluence operator.

Since unidirectional flow problems typically have only one confluence, *SBVP* was never emphasised or mentioned explicitly in the literature. In the case of bidirectional problems, if all confluences merge the global information using the same boolean operator, as is the case in MRA, *SBVP* holds automatically.

Example 4.1 : As noted in example 3.3, MMRA uses two confluences. Consequently, *SBVP* is violated : $\text{PPIN}_j = \mathbf{F}$ can not set the PPIN_i of a successor i to $\mathbf{F}$ on its own, but may do so in combination with PPIN of other predecessors of i. $\square$

Lemma 4.1 : *All bit vector data flow frameworks possess the SBVP property.*
Proof : Let $X(u)$ represent the properties at program point u. The data flow equations can be written in an abstract form as :

$$X(u) = \bigsqcap_{v \in \mathcal{N}^{-1}(u)} h(X(v)) \sqcap \text{CONST}(u)$$

Let p be a property in $X(u)$ and p', the corresponding property in $X(v)$. We know that $p \sqsubseteq \mathcal{B}_h(p')$. By the definition of $\sqcap$, $\mathcal{B}_h(p') = \text{BOT} \Rightarrow p = \text{BOT}$. Thus, p can become BOT regardless of the values of other properties. Thus, p' changes p to BOT on its own and not in combination with other properties. $\square$

There are two important implications of the characteristics *MBVP* and *SBVP* :

(i) A considerable reduction in the work for performing data flow analysis is possible because :

  (a) *MBVP* guarantees that a property p which has become BOT need not be recomputed as it has attained its final value.

  (b) *SBVP* guarantees that all neighbouring properties can be *refined*, rather than recomputed, to incorporate the effect of a property changing to BOT.[1]

[1]Refinement is not a new concept; it can be traced in the worklist-based iterative algorithm in [32]. Here we just make it explicit.

Let $X(u)$ and $X'(u)$ represent the old and new values at program point u, and let v be the program point at which values have changed. Equation 4.1 defines recomputation, while equation 4.2 defines refinement :

$$X'(u) = \prod_{v \in \mathcal{N}^{-1}(u)} h(X'(v)) \sqcap \text{CONST}(u) \tag{4.1}$$

$$X'(u) = X(u) \sqcap h(X'(v)) \tag{4.2}$$

When contrasted with recomputation, which uses the values of *all* neighbouring properties, refinement decreases the amount of work by a factor that depends on the number of in-edges/out-edges of a node. Lemma 4.2 shows the equivalence between refinement and recomputation.

(ii) *MBVP* guarantees the termination of data flow analysis — the values of properties change in one direction only.

Lemma 4.2 : *Refinement and recomputation yield identical results for singular data flow problems.*

Proof : The equation for recomputation can be rewritten as

$$\text{X}'(u) = h_1(\text{X}'(v_1)) \sqcap \cdots \sqcap h_i(\text{X}'(v_i)) \sqcap \cdots \sqcap \text{CONST}(u)$$

Let the properties change at the neighbouring point v_i only, i.e. $\text{X}'(v_j) = \text{X}(v_j),\ j \neq i$. Further,

$$\begin{aligned}
&\text{X}'(v_i) \neq \text{X}(v_i) \\
&\quad \Rightarrow \text{X}'(v_i) \sqsubseteq \text{X}(v_i) && \ldots \text{ since properties can change towards } \bot \text{ only} \\
&\quad \Rightarrow h_i(\text{X}'(v_i)) \sqsubseteq h_i(\text{X}(v_i)) && \ldots \text{ from monotonicity} \\
&\quad \Rightarrow h_i(\text{X}'(v_i)) \sqcap h_i(\text{X}(v_i)) = h_i(\text{X}'(v_i)) && (4.2.\text{A})
\end{aligned}$$

$$\begin{aligned}
\text{X}'(u) &= h_1(\text{X}(v_1)) \sqcap \cdots \sqcap h_i(\text{X}'(v_i)) \sqcap \cdots \sqcap \text{CONST}(u) \\
&= h_1(\text{X}(v_1)) \sqcap \cdots \sqcap h_i(\text{X}'(v_i)) \sqcap h_i(\text{X}(v_i)) \sqcap \cdots \sqcap \text{CONST}(u) && \ldots \text{ from (4.2.A)} \\
&= h_1(\text{X}(v_1)) \sqcap \cdots \sqcap h_i(\text{X}(v_i)) \sqcap \cdots \sqcap \text{CONST}(u) \sqcap h_i(\text{X}'(v_i)) \\
&= \text{X}(u) \sqcap h_i(\text{X}'(v_i))
\end{aligned}$$

Since refinement is applied for every neighbour v_i whenever $\text{X}(v_i)$ changes, refinement yields the same result as recomputation. □

4.2 Performing Data Flow Analysis

4.2.1 Wordwise Analysis

Since the size of a bit vector may vary with the size of the program, we propose to process the bit vectors in parts. Further, we partition the problem of data flow analysis so as to process a specific part of a bit vector, rather than the entire bit vector, in each step. We select a part, process it all over the graph and then select the next part which needs to be processed. At one extreme, each part may consist of one bit as in [25]; at the other extreme, each part may be the largest chunk of a bit vector which can be processed in one machine operation, which typically is a *machine word*. We follow the latter approach, which we term *wordwise analysis*.

Wordwise analysis results in considerable savings in the work to be performed since all parts may not require processing for all nodes of the graph. More formally, let N be the set of nodes, and M, the set of words. The set $N \times M$ is partitioned into two subsets, the set NM_p which requires processing, and the set NM_{np} which does not. Wordwise analysis implies selecting all entries for a specific word from NM_p and processing them. The traditional approach of processing all words of a bit vector partitions only N. Let N_p and N_{np} be the partitions. Since *all* words in M are processed for each node in N_p, it results in more work.

Henceforth, the discussion will be in terms of properties belonging to $< u, w >$ where u is a program point and w is a word.

4.2.2 The Basic Algorithm

Figure 4.1 contains the basic algorithm which is a generalisation of the worklist-based iterative method. The work is divided in two phases : *initialisation* and *propagation*, performed by the procedures *init* and *settle* respectively.

Initialisation

As noted in section 3.2, information flow is initiated by the properties whose initial values are BOT due to local effects of computations existing within a node/along an edge. The *Initial Trigger Set*, denoted TR_0, contains all such properties.

$$\begin{aligned}
Bprops &= \{p \mid \text{ either } \mathcal{P}p(p) = in(i), i \in entry(G) \text{ or} \\
&\qquad \mathcal{P}p(p) = out(i), i \in exit(G)\} \\
TR_0 &= \{p \mid p \leftarrow \mathcal{B}_h(p') \text{ such that } \mathcal{B}_h(\text{TOP}) = \text{BOT or} \\
&\qquad p \in Bprops \text{ and } p = \text{BOT}\} \qquad (4.3)
\end{aligned}$$

```
procedure dfa()
{   init()
    settle()
}
procedure init()
{   for each word w
        for each node i of the graph
        {   Set all inflow properties in word w to TOP
            Compute the outflow properties in word w
            if any property is BOT then        /* it belongs to TR_0 */
                Insert i in the worklist for word w
        }
}
procedure settle()
{   for each word w
        for each node i in the worklist of w
            propagate(i, w)
}
procedure propagate(i, w)
                    /* propagate the effect of the outflow properties of i */
{   for each neighbouring node k of i
    {   Refine the inflow properties of k in word w      /* using g_e */
        Compute the outflow properties of k in word w /* using f_k */
        if some property changes to BOT then
            propagate(k, w)
    }
}
```

Figure 4.1: The Basic Algorithm

After incorporating the initialisation in equations 3.2 and 3.3, TR_0 can be computed from the following equations :

$$\mathrm{IN}_i = \begin{cases} \mathit{Boundaryinfo}_i & \text{if } i \in \mathit{entry}(G) \\ \displaystyle\prod_{j \in pred(i)} g^f_{(j,i)}(\top) \sqcap f^b_i(\top) \sqcap \mathrm{CONST\_IN}_i & \text{otherwise} \end{cases} \tag{4.4}$$

$$\text{OUT}_i = \begin{cases} \mathit{Boundaryinfo}_i & \text{if } i \in \mathit{exit}(G) \\ \prod_{k \in succ(i)} g^b_{(i,k)}(\top) \prod f^f_i(\top) \prod \text{CONST\_OUT}_i & \text{otherwise} \end{cases} \tag{4.5}$$

Example 4.2 : For MRA, TR_0 contains the following properties :

$$\begin{aligned} TR_0 &= \{\text{PPIN}^l_i \mid i \in \mathit{entry}(G) \text{ or } \text{PAVIN}^l_i = \mathbf{F} \text{ or } \text{ANTLOC}^l_i = \text{TRANSP}^l_i = \mathbf{F}\} \\ &\quad \cup \{\text{PPOUT}^l_i \mid i \in \mathit{exit}(G)\} \end{aligned}$$

□

Procedure *init* constructs TR_0 by computing the outflow properties for each node i. If any property p in word w is BOT, node i is inserted in the worklist for word w.

Propagation

Propagation selects a node from the worklist of a given word and propagates the transitive influence of its BOT properties. Effectively, many bits in a word are processed simultaneously. The outflow properties in the current word of a node's bit vector may change the inflow properties of neighbouring nodes, which are refined to incorporate their influence. From these inflow properties, the corresponding outflow properties of the node are computed. If any outflow property changes to BOT, it becomes a candidate for propagation whose influence is propagated to its neighbours by the recursive call in procedure *propagate*.

4.2.3 A Generic Algorithm for Data Flow Analysis

The generic algorithm embodies two major deviations from the traditional algorithms :

1. Wordwise analysis :

   This reduces the amount of work required for data flow analysis.

2. Distinction between entry and exit points of a node :

   This is necessary for the treatment of bidirectional flows.

Figures 4.2 and 4.3 contain the algorithm which uses equations 3.2, 3.3 and 4.4, 4.5 for the IN and OUT properties of a node. The specific points to be noted are :

- Some speedup can be achieved by accumulating as many changes as possible for a *<program_point, word>* pair before refining the properties of neighbouring program points. Hence the pairs with fewer BOT properties are processed later by maintaining the worklists in sorted order according to the number of BOT properties.

```
procedure dfa ()
{   init ()
    settle ()
}
procedure init ()
{   for each word w
        for each node i
        {   if i ∈ entry(G) then
                IN_i = Boundaryinfo_i
            else
                IN_i = ⊓_{j∈pred(i)} g^f_(j,i)(⊤) ⊓ f^b_i(⊤) ⊓ CONST_IN_i
            if any property in IN_i is BOT then    /⋆ it belongs to TR_0 ⋆/
                Insert < i, in(i) > in LIST_w
            if i ∈ exit(G) then
                OUT_i = Boundaryinfo_i
            else
                OUT_i = ⊓_{k∈succ(i)} g^b_(i,k)(⊤) ⊓ f^f_i(⊤) ⊓ CONST_OUT_i
            if any property in OUT_i is BOT then /⋆ it belongs to TR_0 ⋆/
                Insert < i, out(i) > in LIST_w
        }
}
procedure settle ()
{   for each word w
    {   while ∃ an entry <node, program_point> in LIST_w
            Delete <node, program_point> from LIST_w
            if program_point = in(node) then
                propagate_in (node, w)
            else propagate_out (node, w)
    }
}
```

Figure 4.2: A Generic Algorithm for Data Flow Analysis

- The recursive calls during propagation have been eliminated by inserting a node in the worklist during propagation also. Apart from eliminating the overheads associated with

procedure *propagate_in* (i, w)
{ $\text{OUT}_i = \text{OUT}_i \sqcap f_i^f(\text{IN}_i)$ /⋆ refinement using f_i^f ⋆/
    if any property in OUT_i becomes BOT **then**
        Insert $< i, out(i) >$ in LIST_w if not already present
    for all $j \in pred(i)$
    { $\text{OUT}_j = \text{OUT}_j \sqcap g^b_{(j,i)}(\text{IN}_i)$ /⋆ refinement using $g^b_{(j,i)}$ ⋆/
        if any property in OUT_j becomes BOT **then**
            Insert $< j, out(j) >$ in LIST_w if not already present
    }
}
procedure *propagate_out* (i, w)
{ $\text{IN}_i = \text{IN}_i \sqcap f_i^b(\text{OUT}_i)$ /⋆ refinement using f_i^b ⋆/
    if any property in IN_i becomes BOT **then**
        Insert $< i, in(i) >$ in LIST_w if not already present
    for all $k \in succ(i)$
    { $\text{IN}_k = \text{IN}_k \sqcap g^f_{(i,k)}(\text{OUT}_i)$ /⋆ refinement using $g^f_{(i,k)}$ ⋆/
        if any property in IN_k becomes BOT **then**
            Insert $< k, in(k) >$ in LIST_w if not already present
    }
}

Figure 4.3: A Generic Algorithm for Data Flow Analysis (contd. from Fig. 4.2)

recursion, this aids in delayed propagation.

- For refinement of properties during propagation, it is sufficient to apply a function h' instead of $h(\text{Z}) = \text{C}_1 + \text{C}_2 \cdot \text{Z}$ where h' is defined as follows.
  - For intersection problems : $\forall \text{Z} \in \mathcal{L}$, $h'(\text{Z}) = \text{C}_1 + \text{Z}$, since

$$\begin{aligned}
\text{X}'(u) &= \text{X}(u) \cdot h(\text{X}'(v)) && \ldots \text{ from equation (4.2)} \\
&= \text{X}(u) \cdot h(\text{X}(v)) \cdot h(\text{X}'(v)) && \ldots \text{ since } \text{X}(u) \sqsubseteq h(\text{X}(v)) \\
&= \text{X}(u) \cdot (\text{C}_1 + \text{C}_2 \cdot \text{X}(v)) \cdot (\text{C}_1 + \text{C}_2 \cdot \text{X}'(v)) \\
&= \text{X}(u) \cdot (\text{C}_1 + \text{C}_2 \cdot \text{X}(v) \cdot \text{X}'(v)) \\
&= \text{X}(u) \cdot ((\text{C}_1 + \text{C}_2 \cdot \text{X}(v)) \cdot (\text{C}_1 + \text{X}'(v))) \\
&= \text{X}(u) \cdot h(\text{X}(v)) \cdot h'(\text{X}'(v)) \\
&= \text{X}(u) \cdot h'(\text{X}'(v)) && \ldots \text{ since } \text{X}(u) \sqsubseteq h(\text{X}(v))
\end{aligned}$$

– For union problems : $\forall Z \in \mathcal{L}, h'(Z) = C_2 \cdot Z$, since

$$
\begin{aligned}
X'(u) = X(u) + h(X'(v)) & & \ldots \text{ from equation (4.2)} \\
= \; & X(u) + h(X(v)) + h(X'(v)) & \ldots \text{ since } X(u) \sqsubseteq h(X(v)) \\
= \; & X(u) + (C_1 + C_2 \cdot X(v)) + (C_1 + C_2 \cdot X'(v)) & \\
= \; & X(u) + C_1 + C_2 \cdot X(v) + C_2 \cdot X'(v) & \\
= \; & X(u) + (C_1 + C_2 \cdot X(v)) + (C_2 \cdot X'(v)) & \\
= \; & X(u) + h(X(v)) + h'(X'(v)) & \\
= \; & X(u) + h'(X'(v)) & \ldots \text{ since } X(u) \sqsubseteq h(X(v))
\end{aligned}
$$

Thus, only two operations are required per function application instead of three.

The generic algorithm is uniformly applicable to unidirectional and bidirectional flows. Since the algorithm belongs to iterative class of methods, a data flow analysis algorithm for a given problem can be automatically constructed from the data flow equations without the knowledge of the semantics of the underlying problem.[2] Alternatively, the generic algorithm can be augmented to interpret the data flow equations and execute the appropriate lines. Adaptation of the algorithm is a significant advantage as far as reliability and ease of development of a data flow analysis module is concerned.

Chapter 5 presents an adaptation of the algorithm for MRA

4.3 Complexity of Data Flow Analysis

4.3.1 Complexity of the Generic Algorithm

Since the size of a bit vector may vary with the size of the program (section 4.2.1), the unit of work for performance analysis should be the work required to process *one word* rather than the work required to process one bit vector. Hence, in the following, an *operation* refers to one bit vector operation on the properties located in one word of a node. The bounds derived in this section assume that the number of edges, e, is $O(n)$ where n denotes the number of nodes. Figure 4.4 gives the notations used in this section.

To estimate the complexity, we develop a notion of *orthogonality* in bit vector processing. Let n_1 and n_2 be some two nodes in the worklist for the word being processed. The order in which the effect of n_1 and n_2 is propagated does not matter; final result is a superposition of the effects of various nodes regardless of the order in which they are processed. Similarly,

[2]Elimination methods cannot be extended to general bidirectional data flow problems [24].

n	number of nodes.
e	number of edges.
no_w	number of words.
d_in_i	the in-degree of node i.
d_out_i	the out-degree of node i.
$\mathcal{B}_j$	$\{ k \mid k$ is a bit in word j and figures in TR_0 for some node $\}$.
b_j	$\lvert \mathcal{B}_j \rvert$.
n_j	number of nodes in the worklist for word j after TR_0 construction.
$op\_prop^j_k$	number of operations performed while processing the properties corresponding to the bit k in $\mathcal{B}_j$.
max_op_prop	$\max(op\_prop^j_k) \quad 0 \leq k \leq b_i,\ 1 \leq j \leq no\_w$.
$op\_word^j_i$	number of operations performed while propagating the effect of word j of node i.
max_op_word	$\max(op\_word^j_i) \quad 0 \leq j \leq no\_w,\ 1 \leq i \leq n$
t_op_j	total number of operations required to refine the properties of word j.
t_work	total work performed during propagation.

Figure 4.4: Notations used for performance analysis

since the bits are independent, the words may also be processed in any order. Thus, while propagating the effect of a node on some worklist, we may safely ignore the presence of other nodes in the same worklist, and all nodes in all other worklists. If we can estimate the amount of work done for one node in one worklist, it is easy to estimate the total work done — it is the sum of the work done for all nodes in all worklists.

Note that due to the orthogonality among words and among the nodes in one worklist, all changes that have to take place in a given word due to a given node in the worklist, will take place simultaneously. If some bit changes later, it must be due to the effect of some other node in the same worklist. This argument forms the basis of the following lemma.

Lemma 4.3 : *After constructing TR_0, the effect of the BOT properties of a given node in the worklist for a given word can be completely propagated in $O(n)$ operations.*

Proof : Let node i be reached while propagating the effect of properties in word m of some node r. Without loss of generality, assume that the effect of IN_i properties is propagated first. Since two operations are required for refinement (section 4.2.3), the effect of changes in IN_i can be propagated in $2 \cdot d\_in_i + 2$ operations — $2 \cdot d\_in_i$ operations for refining the OUT properties of predecessors and two operations for refining OUT_i. The effect of the BOT properties in OUT_i, if any, can be propagated in $2 \cdot d\_out_i$ operations. Thus, the maximum number of operations performed on visiting node i is $2 \cdot d\_in_i + 2 \cdot d\_out_i + 2$. In the worst case all the nodes may have to be visited. Hence,

$$max\_op\_word = \sum_{i=1}^{n}(2 \cdot d\_in_i + 2 \cdot d\_out_i + 2) = 4 \cdot e + 2 \cdot n$$

which is $O(n)$. □

Lemma 4.4 : *The effect of a given property in* TR_0 *can be completely propagated in O(n) operations.*

Proof : As in lemma 4.3, it can be shown that $max\_op\_prop = 4 \cdot e + 2 \cdot n$. □

Lemma 4.5 : $O(n^2)$ *operations are needed to propagate the influence of all properties in* TR_0.

Proof : Propagating the influence of the properties of a node may subsume the influence of the properties of some other node. Thus,

$$\begin{aligned} t\_op_j &\leq n_j \cdot max\_op\_word \\ t\_work &= \sum_{j=1}^{no\_w} t\_op_j \\ &\leq \sum_{j=1}^{no\_w} n_j \cdot max\_op\_word \\ &\leq max\_op\_word \cdot \sum_{j=1}^{no\_w} n_j \end{aligned} \tag{4.6}$$

Further, since all properties in a word are processed simultaneously,

$$\begin{aligned} t\_op_j &\leq b_j \cdot max\_op\_prop \\ t\_work &= \sum_{j=1}^{no\_w} t\_op_j \\ &\leq \sum_{j=1}^{no\_w} b_j \cdot max\_op\_prop \\ &\leq max\_op\_prop \cdot \sum_{j=1}^{no\_w} b_j \end{aligned} \tag{4.7}$$

Both *max_op_word* and *max_op_prop* are $4 \cdot e + 2 \cdot n)$. Hence, it follows from 4.6 and 4.7 that,

$$t\_work \leq 4 \cdot e + 2 \cdot n \cdot \min(\sum_{j=1}^{no\_w} n_j, \sum_{j=1}^{no\_w} b_j) \tag{4.8}$$

The worklist length for a word is $O(n)$ hence $\sum n_j$ could be O(n^2). Since *no_w* is O(n), $\sum b_j$ is O(n) and hence, min($\sum n_j, \sum b_j$) is $O(n)$. Since $e = O(n)$, *t_work* is O(n^2). □

Lemma 4.6 : TR_0 *can be constructed in O(n^2) operations.*

Proof : It is evident from equations 4.4 and 4.5 that $3 \cdot d\_in_i + 3$ operations are required to compute the values of IN_i — two operations for computing each $g^f_{(j,i)}(\top)$, two operations for computing $f^b_i(\top)$ and $d\_in_i + 1$ operations for meet. Similarly, $3 \cdot d\_out_i + 3$ operations are required to compute the values of OUT_i. Hence the total number of operations for one word is

$$\sum_{i=1}^{n} (3 \cdot d\_in_i + 3 \cdot d\_out_i + 6) = 6 \cdot (e + n)$$

which is $O(n)$. Since there are $O(n)$ words, the complexity is $O(n^2)$. $\square$

Theorem 4.1 : *Total work done by the proposed algorithm is* $O(n^2)$.

A closer look at the proofs of lemmas 4.5 and 4.6 reveals that refinement does not seem to play any role in the complexity. Since e is $O(n)$, it is perfectly valid to assume that the degree of a node is bounded by a constant. In such a case even if the properties are recomputed, the order of the work involved remains the same. However, refinement has a practical significance as it reduces the number of operations by a constant factor.

4.3.2 Performance in Practical Situations

Though the theoretical complexity of the algorithm is $O(n^2)$, there are several reasons to believe that the performance would be better in practice.

Initialisation

The edge flow functions for almost all known data flow frameworks are either identity functions or functions of the form $h(X) = C_1 + X$ for the intersection problems (*viz.* MRA and LSIA) and $h(X) = C_1 \cdot X$ for union problems (*viz.* CHSA). Thus the application of edge flow functions, $h(\top)$, required for the TR_0 construction (equations 4.4, 4.5), is almost always $\top$ and the number of operations per word reduces from $3 \cdot d\_in_i + 3 \cdot d\_out_i + 6$ to 3 — two operations for the node flow function and one for the meet with constant properties. In the case of unidirectional problems, since the constant properties CONST_IN and CONST_OUT are typically $\top$, the number of operations further reduces to 2. These operations incorporate the local effect of a node and represent the minimum amount of work that must be done for any data flow problem. Note that though the bound on initialisation is $O(n^2)$, it requires only one traversal over the graph.

Propagation

Let K be defined as follows :

$$K \;=\; \min\Big(\sum_{j=1}^{no\_w} n_j,\ \sum_{j=1}^{no\_w} b_j\Big) \tag{4.9}$$

Then the bound on the propagation becomes $O((e+n)\cdot K)$.

After the initialisation is performed, we can evaluate K and determine a more realistic bound on the work required by propagation. In the best case, both no_w and K might be 1, in which case the work is $O(n)$. The actual work performed by the propagation is likely to be even better than the estimate in terms of K :

- We *do not* process the individual bits but words of bits.
- In practice, the effect of the BOT properties of a node in the worklist *may not* propagate over the entire graph.
- Since propagation is delayed as far as possible, the effect of some nodes may be subsumed by propagation for other nodes. Hence the effect of all nodes in the worklist may not have to be propagated separately.
- Some other heuristics can be employed for the worklist organisation to select appropriate nodes for propagation. In the case of MRA, forward node flow does not exist (i.e. f_k^f is $\top$) hence the information flow is predominantly backwards. Thus, it may be beneficial to process the nodes in the postorder as it may propagate the effect of some changes more rapidly.

4.4 Correctness of Data Flow Analysis

Instead of directly showing the correctness of an algorithm, it is often easier to define the solution obtained by the algorithm and show the correctness of the solution. In this section we first define an abstraction of the generic algorithm. Then we provide a definition of the solution obtained by the algorithm. Finally, we show the correctness of the solution.

4.4.1 A State Transition Model for Data Flow Analysis

Following section 4.1, data flow analysis can be viewed as a process of successive refinement of properties, each refinement changing some property from TOP to BOT. We can visualise each configuration of data flow properties as a *state*. Thus each refinement constitutes one state transition. We propose a state transition model of data flow analysis in this section.

Definition 4.1 : **Dependence**
A property p depends on a property p′ if p′ is capable of changing p to BOT.

$$Depends(p) = \{p' \mid p \leftarrow \mathcal{B}_h(p') \text{ and } \mathcal{B}_h \equiv propagate\}$$

Definition 4.2 : **Seed**
A property p′ is a seed of a property p if p depends on p′ and p′ is BOT.

$$Seed(p) = \{p' \mid p' \in Depends(p) \text{ and } p' = \text{BOT}\}$$

Since a property could influence the corresponding properties of several neighbouring nodes, many properties may have a common seed. Similarly, a property may have multiple seeds.

States and State Transitions

Definition 4.3 : **State**
A state S_i is a configuration of the values of properties defined by the pair $<TR_i,CA_i>$, where the sets TR_i and CA_i are defined as follows.

Definition 4.4 : **Trigger Set**
Trigger set TR_i is a set of properties which are BOT in state S_i.

Definition 4.5 : **Candidate Set**
Candidate set CA_i is a set of properties which are TOP in state S_i and have a seed in TR_i, i.e.

$$CA_i = \{p \mid p = \text{TOP and } Seed(p) \cap TR_i \neq \emptyset\}$$

The properties contained in a trigger set are the properties which are *not* going to change any further. However, they influence the properties contained in the candidate set.

Observation 4.1 : $i_1 < i_2 \Rightarrow TR_{i_1} \subset TR_{i_2}$. □

Let $S_0, S_1, S_2, \ldots, S_f$ be the sequence of states during an analysis where S_0 is the initial state and S_f is the final state.

- The initial state S_0 is characterised by TR_0 which was defined in section 4.2.2.

  Note that a $p \in TR_0$ has no seed. A $p \notin TR_0$ may have multiple seeds.

- The final state S_f is characterised by $CA_f = \emptyset$.

For a given set of *Boundaryinfo* values, there is a unique start and a unique final state.

A state transition is caused by a change in the value of some property. We denote a transition from state S_i to state S_{i+1} on a property p becoming BOT, by $S_i \xrightarrow{p} S_{i+1}$.

Observation 4.2 : $S_i \xrightarrow{p} S_{i+1} \Rightarrow \text{Seed}(p) \cap TR_i \neq \emptyset$ *and* $p \in TR_{i+1}$. □

Consider a transition $S_i \xrightarrow{p} S_{i+1}$. The property p may have multiple seeds and several properties may depend on p. Since any sequence of transitions is necessarily acyclic, the state transition diagram for the complete model is a directed acyclic graph. For a given model, different analyses may follow different sequences of transitions, all starting in state S_0 and ending in state S_f.

Definition 4.6 : **Chain**[3]
A chain $CH(p', p)$ *is the sequence* $p_0, p_1, \ldots, p_n$ *where* $p_0 = p', p_n = p$ *and* $p_{j-1} \in \text{Seed}(p_j)$, $0 < j \leq n$. *The first property* p' *may or may not have a seed.*

The length of a chain $\text{CH}(p', p) = p_0, p_1, \ldots, p_n$ is n. The properties $p_0, \cdots, p_{n-1}$, are the *transitive seeds* of p. Let *Depends*$\star(p)$ and *Seed*$\star(p)$ denote the transitive closures of *Depends*(p) and *Seed*(p) respectively. It follows that,

$$\forall p,\ Seed\star(p) \subseteq Depends\star(p)$$

Since a property can have multiple seeds, a property may belong to several chains. Two chains $\text{CH}(p_1, p_n)$ and $\text{CH}(p'_1, p'_n)$ are said to *overlap* if they share a property which is not the first property of any of them (i.e. either p_1 or p'_1). Since a property changes only once, a chain is necessarily acyclic.

A chain is realized by a sequence of state transitions.

Number of States and State Transitions

It is obvious that the total number of states is exponential in the number of properties. More formally, if the size of the bit vector is r and there are n nodes in the program flow graph, the total number of properties is $2nr$. Since each property could be either **T** or **F** the total number of combinations is 2^{2nr}.

[3]Not to be confused with the lattice chains.

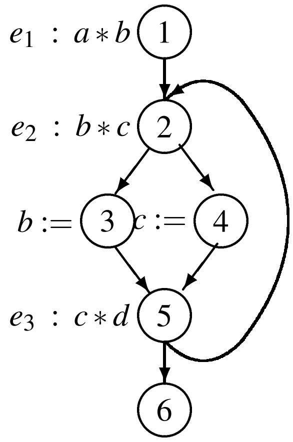


Figure 4.5: Graph for state transition model of available expressions.

Since a chain fragment is itself a chain, the total number of chains is exponential in the number of properties. However, a property changes only once and the total number of transitions during an analysis is bounded by *the sum of the lengths of all non-overlapping chains*. For a given property, this sum cannot exceed the number of program points (which is $2n$). If there are r bits, the number of transitions is bounded by $2nr$.

The goal of data flow analysis is to construct TR_f for the final state S_f. This is achieved by constructing TR_0 and tracing all chains for all properties. This approach can be applied uniformly irrespective of the direction of data flow, the confluence operator and the graph topology.

Example 4.3 contains an illustration of a state transition model.

Example 4.3 : For the simplicity of exposition, consider a forward unidirectional problem *viz.* the problem of available expressions analysis for the program flow graph in Figure 4.5. Assume that all the variables are local variables. Then *Boundaryinfo* is $\overline{\text{F}}$ (see section 3.3.2). Let IN_i^l and OUT_i^l represent the AVIN_i^l and AVOUT_i^l properties respectively.

- **Start State** : $S_0 = < TR_0, CA_0 >$.

  (i) *Boundaryinfo* is $\overline{\text{F}}$ for node 1, and $\overline{\text{T}}$ (i.e. $\top$) for node 6. Further, since COMP and TRANSP are **F** for e_1 and e_2 in node 3 and for e_2 and e_3 in node 4,

  $TR_0 = \{\text{IN}_1^1, \text{IN}_1^2, \text{IN}_1^3, \text{OUT}_3^1, \text{OUT}_3^2, \text{OUT}_4^2, \text{OUT}_4^3\}$.

  (ii) $CA_0 = \{\text{OUT}_1^2, \text{OUT}_1^3, \text{IN}_5^1, \text{IN}_5^2, \text{IN}_5^3\}$.

- **Dependences and Chains :**

  Let p and p' be the properties for the expression $e_l \in \{e_1, e_2, e_2\}$. The dependence relation $Depends(p) = \{p'\}$ holds for the following pairs $< p, p' >$

  (i) $< \text{IN}_j^l, \text{OUT}_i^l >$ such that edge (i, j) exists in the graph, and

  (ii) $< \text{OUT}_i^l, \text{IN}_i^l >$ such that $\text{COMP}_i^l = \mathbf{F}$ and $\text{TRANSP}_i^l = \mathbf{T}$.

  In particular, dependence does not hold for the pairs $< \text{OUT}_i^l, \text{IN}_i^l >$, for the (expression, node) pairs $(l, i) \in \{(1,1),(2,2),(1,3),(2,3),(2,4),(3,4),(3,5)\}$.

  The possible chains for the property $\text{OUT}_3^1 \in TR_0$ are :

  (i) $\underline{\text{OUT}_3^1, \text{IN}_5^1, \text{OUT}_5^1, \text{IN}_6^1, \text{OUT}_6^1}$.

  (ii) $\text{OUT}_3^1, \text{IN}_5^1, \underline{\text{OUT}_5^1, \text{IN}_2^1, \text{OUT}_2^1, \text{IN}_3^1}$.

  (iii) $\text{OUT}_3^1, \text{IN}_5^1, \text{OUT}_5^1, \text{IN}_2^1, \underline{\text{OUT}_2^1, \text{IN}_4^1, \text{OUT}_4^1}$.

  The underlines indicate the non-overlapping chains.

- **State Transitions :**

  If OUT_3^1 is selected from TR_0 as the first property and the chains are traced in the order they are enumerated, the sequence of transitions is

$$S_0 \overset{\text{IN}_5^1}{\to} S_1 \overset{\text{OUT}_5^1}{\to} S_2 \overset{\text{IN}_6^1}{\to} S_3 \overset{\text{OUT}_6^1}{\to} S_4 \overset{\text{IN}_2^1}{\to} S_5 \overset{\text{OUT}_2^1}{\to} S_6 \overset{\text{IN}_3^1}{\to} S_7 \overset{\text{IN}_4^1}{\to} S_8 \overset{\text{OUT}_4^1}{\to} S_9 \cdots S_f.$$

- **Final State :** $S_f = < TR_f, CA_f >$

  (i) From the definition of the final state, $CA_f = \emptyset$.

  (ii) TR_f contains all properties except the following properties which are TOP in the final state : $\{\text{OUT}_1^1, \text{OUT}_2^2, \text{IN}_3^2, \text{IN}_4^2, \text{OUT}_5^3, \text{IN}_6^3, \text{OUT}_6^3\}$.

- **Number of States and State Transitions :**

  Since there are 3 bits and 6 nodes, total number of states is 2^{2*6*3} which is 2^{36}. The number of transitions during an analysis is bounded by the sum of the lengths of all non-overlapping chains which, for expression e_1, is 9. Even if the chains for all expressions are not enumerated, the upper bound on the number of transitions (for all expressions) is $2nr$ which is 36.

□

4.4.2 Correctness of the Generic Algorithm

Correctness is established by showing that the solution obtained is acceptable. From the discussion in section 2.3, it is evident that the acceptability of a solution follows if it can be shown that a solution procedure converges on MFP. There are three steps in the proof :

(a) existence of a fixed point solution,

(b) convergence on a fixed point, and

(c) maximality of the fixed point.

Step (a) follows from monotonicity and the fact that every strictly descending chain in the lattice is of finite length. We prove (b) and (c) in this section. Recall that our solution procedure consists of two steps :

(i) *Initialisation* : Constructing TR_0.

(ii) *Propagation* : Constructing all chains for all properties in TR_0.

Lemma 4.7 : *A property p changes to BOT iff either $p \in TR_0$ or $Seed\star(p) \cap TR_0 \neq \emptyset$.*
Proof : Follows directly from the solution procedure. □

Lemma 4.8 : *Propagation converges on a fixed point.*
Proof : Termination is guaranteed by monotonicity. We show here that the propagation terminates on a fixed point. Recall that a fixed point is defined as the fixed point of equations 3.2 – 3.3. We can rewrite the equations in a more abstract form as

$$Y = \bigsqcap h(X) \bigsqcap \text{CONST}$$

where $\bigsqcap h(X)$ represents the meet of the edge/node flow functions.

Assume that after the algorithm terminates, the resulting assignment Y is not a fixed point, i.e. if $Z = \bigsqcap h(X) \bigsqcap \text{CONST}$, then $Y \neq Z$. Let X^i denote the i^{th} bit of X. Then some Y^i and Z^i have different values.

(a) Y^i = TOP, Z^i = BOT.

Z^i could be BOT if

(i) X^i = BOT or,

(ii) X^i = TOP but $\mathcal{B}_h^i(X^i)$ = BOT or,

(iii) X^i = TOP, $\mathcal{B}_h^i(X^i)$ = TOP but CONST^i = BOT.

In case (i) $X^i \in Seed(Y^i)$. In cases (ii) and (iii) $Y^i \in TR_0$. By lemma 4.7, Y^i must be BOT in all the cases which leads to a contradiction.

(b) Y^i = BOT, Z^i = TOP.

If Z^i is TOP, Y^i can not have a seed nor does it belong to TR_0. From lemma 4.7, Y^i cannot be BOT.

□

Theorem 4.2 : *The fixed point obtained by the solution procedure is the maximum fixed point.*
Proof : This can be proved by induction on states. Let $S_0, S_1, S_2, \ldots, S_f$ be the sequence of states during an analysis. Let FP be some fixed point, and let the assignment in state S_j be AS$_j$.

For the basis case, we know that TR_0 computation involves only constants and hence, all the properties which are in TR_0 would necessarily be BOT in any fixed point solution. In other words, *all* the bits which are BOT in AS$_0$ are necessarily BOT in FP; the bits which are TOP in AS$_0$ may or may not be TOP in FP. Thus,

$$\forall\, i \in N \;:\; \text{FP}(i) \;\sqsubseteq\; \text{AS}_0(i)$$

Assume that the containment holds for some state S_k. Let $S_k \xrightarrow{p} S_{k+1}$. This is possible only if $Seed(p) \cap TR_k \neq \emptyset$. Since some seed of p is BOT in FP (by the virtue of being in TR_k), p must be BOT in FP. Hence,

$$\forall\, i \in N \;:\; \text{FP}(i) \;\sqsubseteq\; \text{AS}_{k+1}(i)$$

Since the containment holds for $k+1$, by induction it holds for the final state S_f i.e., FP is contained in AS$_f$. Since this holds for any FP, all fixed points are contained in AS$_f$ which itself is a fixed point (from lemma 4.8). Hence AS$_f$ is the maximum fixed point. □

Since MFP is same as MOP for all bit vector problems, the algorithm computes the MOP solution for all bit vector problems.

4.5 Looking Back

Transition from chapter 3 to this chapter is a transition from theory to practice. This transition is backed up by a detailed analysis of performance and a proof of correctness of the proposed algorithm. Generality and adaptability are two significant features of the algorithm. These advantages follow almost automatically since the algorithm is based on a generalised theory and belongs to the iterative class of methods. This is so by design, and not by accident, though!

Chapter 5

An Efficient Solution Procedure for MRA

This was the merchant who sold pills that had been invented to quench the thirst. You need only swallow one pill a week, and you would feel no need of anything to drink.
"Why are you selling those ?" asked the little prince.
"Because they save a tremendous amount of time," said the merchant. "Computations have been made by experts. With these pills, you save fifty-three minutes in every week."

This chapter reviews some approaches of solving the partial redundancy elimination algorithms, discusses their limitations and presents an efficient solution procedure for MRA which is an adaptation of the generic algorithm. Section 5.3.2 compares its performance with the round robin iterative method for MRA.

5.1 Speedy Solution of MRA-Class of Algorithms

The classical elimination methods are not directly applicable to the solution of bidirectional data flow problems [24], hence they have been solved conventionally using the round robin iterative approach. The observable complexity of this approach is reported to be 5 or fewer iterations for most programs [34, 35, 45]. The first (strict) bound on the number of iterations is derived in [23] which is discussed in chapter 6. In this section we discuss some other approaches which solve MRA *indirectly* in the sense that they solve *equivalent* problems but fail to solve MRA directly. These approaches will also be discussed in chapter 6.

5.1.1 The Edge Placement Approach

The forward dependency of MRA arises due to safety considerations, and the desire to place a hoisted computation strictly in a node of the original program flow graph. If the latter condition is relaxed, it is possible to eliminate the forward dependencies. In such an event, a computation which is hoisted out of node i, but which can not be placed in a predecessor node j due to safety considerations, is placed along the edge (i,j). A special *synthetic node* is introduced in the program flow graph to accommodate such computations. A complete description of such an algorithm, called the *edge placement algorithm* (EPA), and its solution aspects is contained in [17]. This approach also has some specific advantages when used for Load-Store placement in register assignment [20]. Other related work can be found in [21, 12, 58, 27].

Elimination of the forward dependencies in this manner reduces the primary code placement problem to a *unidirectional* problem which has lower solution complexities. However, this approach faces the following difficulties :

- The approach may perform poorly due to *code proliferation* [17]. To inhibit such proliferation, it becomes necessary to introduce an additional profitability criterion, which re-introduces forward dependencies. [17] suggests a fast solution technique for the data flows involved, however the approach continues to suffer from the drawback mentioned below.

- Introduction of synthetic nodes in the program introduces run time overheads (upto 2 additional branch instructions per synthetic node [17]).

5.1.2 The Edge-Splitting Approach

The edge-splitting approach *a priori* splits edges which run from a *fork* node (i.e. a node with more that one successor) to a *join* node (i.e. a node with more that one predecessor). The placement problem is then solved as a backward unidirectional problem, followed by a forward *correction* pass over the program (note that the forward correction is *not* a data flow problem, it is simply one pass over the program) [25]. This approach suffers from the following drawbacks :

- It is more expensive than solving the primary data flow in EPA, which is simply a backward unidirectional data flow problem.

- It requires an additional edge-splitting pass, which amounts to introducing synthetic nodes along edges which are to be split.
- Its solution is different from the MRA solution, in the sense that expressions may be needlessly put along edges, whereas they would have been put into program nodes by MRA. This problem is not faced by the edge placement approach.
- Some of the synthetic nodes introduced due to edge splitting may be redundant. These have to be removed.

5.2 Adapting the Generic Algorithm for MRA

The generic algorithm can be easily adapted for any bit vector data flow problem. The modifications involve omission/replication of some computations and are governed by the flow functions of a data flow framework, which can be derived from the data flow equations syntactically without requiring any knowledge of the underlying problem.

Figures 5.1 and 5.2 contain a straightforward adaptation for MRA assuming the *Boundaryinfo* values to be $\overline{\mathrm{F}}$. Since there is no forward node flow in MRA the lines corresponding to f^f are omitted from the generic algorithm. Besides, there is no need to use separate bit vectors for CONST_OUT properties since they are TOP.

5.3 Empirical Performance of the Algorithm

The algorithm was implemented and integrated in the optimiser of an optimising compiler developed at IIT Bombay. Extensive experiments were carried out with a a set of scientific programs written in Fortran.

5.3.1 Speeding Up the Algorithm

Empirical profiling revealed that even without any heuristic for list organisation, the proposed algorithm performed much better in terms of the number of bit vector operations than the round robin method which is how MRA has been solved traditionally. However, regardless of the heuristic, it fared poorly in terms of actual time initially. This was due to the ‘ overheads associated with :

1. Function calls : Round robin method does not make any function calls.

```
procedure mra ()
{     init ()
      settle ()
}
procedure init ()
{     for each word w
           for each node i
           {     if i ∈ entry(G) then
                       PPIN_i = F̄
                 else
                       PPIN_i = PAVIN_i·( ANTLOC_i + TRANSP_i)
                 if any property in PPIN_i is F then    /⋆ it belongs to TR_0 ⋆/
                       Insert < i, in(i) > in LIST_w
                 if i ∈ exit(G) then
                       PPOUT_i = F̄
                 else
                       PPOUT_i = T̄
                 if any property in PPOUT_i is F then /⋆ it belongs to TR_0 ⋆/
                       Insert < i, out(i) > in LIST_w
           }
}
procedure settle ()
{     for each word w
      {     while ∃ an entry <node, program_point> in LIST_w
                 Delete <node, program_point> from LIST_w
                 if program_point = in(node) then
                       propagate_in (node, w)
                 else propagate_out (node, w)
      }
}
```

Figure 5.1: A Straightforward Adaptation for MRA

2. Worklist maintenance : Round robin method does not require any worklist.

The performance of the algorithm was improved by changes at three levels :

procedure *propagate_in* (i, w)
$\{$ **for** all $j \in pred\,(i)$
$\{$ $\text{PPOUT}_j = \text{PPOUT}_j \cdot \text{PPIN}_i$ /⋆ refinement using $g^b_{(j,i)}$ ⋆/
if any property in PPOUT_j becomes **F then**
Insert $< j, out\,(j) >$ in LIST_w if not already present
$\}$
$\}$
procedure *propagate_out* (i, w)
$\{$ $\text{PPIN}_i = \text{PPIN}_i \cdot (\text{ANTLOC}_i + \text{PPOUT}_i)$ /⋆ refinement using f^b_i ⋆/
if any property in PPIN_i becomes **F then**
Insert $< i, in(i) >$ in LIST_w if not already present
for all $k \in succ\,(i)$
$\{$ $\text{PPIN}_k = \text{PPIN}_k \cdot (\text{AVOUT}_i + \text{PPOUT}_i)$ /⋆ refinement using $g^f_{(i,k)}$ ⋆/
if any property in PPIN_k becomes **F then**
Insert $< k, in(k) >$ in LIST_w if not already present
$\}$
$\}$

Figure 5.2: A Straightforward Adaptation for MRA (contd. from Figure 5.1)

1. As a first optimisation, the function calls were removed from the algorithm by in-line expansion. The difference was remarkable. Still, the round robin method out-performed the algorithm in some list organisations.

2. As a second optimisation, the lists were implemented using arrays with the indices serving as pointers. This was possible since for a given word the number of nodes in the list can never exceed n (i.e. total number of nodes).

3. The third level optimisations concerned the heuristics for list organisation.

5.3.2 A Comparison of Different Heuristics

The experimental results are contained in appendix B. Each heuristic is compared with the round robin method in terms of both time and bitwise operations.

1. FIFO :

   The simplest heuristic for list organisation is the *First in first out* strategy. Though, the number of bit vector operations reduces (average speed-up factor is 1.58), it remains slower than the round robin method in terms of time; the overheads associated with the list manipulation more than nullify the gains in terms of bit vector operations.

2. MBOT :

   Section 4.2.3 suggests propagating the influence of the node with *Maximum number of BOT* properties. It indeed saves on bit vector operations and the average speed up factor is 2.84 (in fact it is less than 2 in only one case). However, the added complexity of maintaining a sorted list makes this heuristic slower than FIFO.

3. PORD :

   Section 4.3.2 suggests maintaining the list of nodes in *Postorder* for MRA since the information flow is predominantly backwards in MRA. It can be verified that gains in bit vector operations are comparable to the gains made by MBOT. In terms of time it is much faster than MBOT, though still slower than round robin method.

   Intuitively, this heuristic tries to combine the best of both worlds : a preferred order of traversal of round robin method and the requirement driven processing of worklist method.

4. IPLM :

   Our final heuristic is an *Improved postorder list management* scheme which tries to eliminate the redundant list manipulations that are made by PORD.

   Apart from maintaining a list, we maintain a global array PROCESS where the entry PROCESS[i] indicates whether node i needs processing or not. Let the properties at program point u (node i) influence the properties at program point v (node j). If the postorder number of j is higher than the postorder number of i, PROCESS[i] is set to **T**. It is inserted in the list only when its postorder number is lower than the postorder number of i. Whenever there is a node in the list, it is processed first. If the list is empty (which is what happens most of the times), the nodes which need processing are processed in postorder.

   This heuristic minimises the overheads of list organisation and the savings in bit operations truly get reflected in the time requirements. It can be easily verified that this heuristic results in superb gains both in bit operations as well as time.

5.4 Concluding Remarks

The transition from chapter 3 to this chapter has been a smooth transition from theoretical insights to practical gains. The algorithm for MRA presented in this chapter is far more superior than the traditional approach of the round robin iterative analysis. For compilers making heavy use of MRA in the optimisation phase, this algorithm offers obvious advantages. Unlike other efforts, this algorithm provides solutions of MRA rather than of algorithms which are *deemed equivalents* of MRA.

Chapter 6

The Width of a Graph

When a mystery is too overpowering, one dare not disobey. Absurd as it might seem to me, a thousand miles away from any human habitation and in danger of death, I took out of my pocket a sheet of paper and my fountain pen. But then ... I told the little chap (a little crossly, too) that I did not know how to draw. He answered me "That does not matter. Draw me a sheep ..."

Though chapter 4 shows that the bidirectional problems are no more complex than the unidirectional problems, certain complexity issues, viz. the bound on the number of iterations in the round robin data flow analysis of bidirectional flows, remain outside the purview of chapter 4.

This chapter relates the information flow paths to the complexity of data flow analysis by defining the notion of the *width* of a graph for a data flow problem. The width is shown to bound the number of iterations required for round-robin iterative data flow analysis. This notion is uniformly applicable to unidirectional and bidirectional flows and provides a more accurate bound than the traditional notion of the depth of a graph. More importantly, width provides the first (strict) bound on the round-robin analysis of bidirectional flows. Other applications include explanation of isolated results in efficient solution techniques and motivation of new techniques for bidirectional flows. In particular, we discuss edge-splitting, edge placement and develop a feasibility criterion for decomposition of a bidirectional flow into a sequence of unidirectional flows. Among the efficient methods for the solution of bidirectional flows are the method of alternating iterations, and an interval analysis based elimination method for MRA.

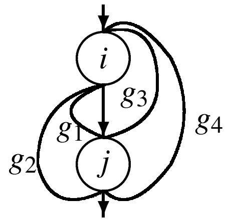


Figure 6.1: General edge flow functions

6.1 General Data Flow Problems

The concepts in this chapter are applicable to a wider class of problems than the one covered by previous chapters. This generality is in terms of :

- The confluence operator : It is possible to analyse the complexity of non-singular problems too, using the notions defined in this chapter despite the fact that the *ifp*'s for such problems cannot be defined precisely without a knowledge of the semantics of a data flow problem.[1]
- The edge flow functions : In general, the information flow along/against an edge (i, j) need not be between $out(i)$ and $in(j)$ *only*; other combinations involving $in(i)$ and $out(j)$ too are possible.

Non-singular problems are already defined in section 3.2.2. Apart from MMRA, we consider the Edge Placement Algorithm [17] (EPA, for short) which is another non-singular data flow problem.

6.1.1 General Flow Functions

A complete list of possible flow functions in a data flow problem, is :

(i) Information flows *within* a node, i.e. between the entry and exit of node i : Represented by *node flow* functions f_i.

(ii) Information flows along edge (i, j) : Represented by *edge flow* functions $g_{(i,j)}$. (Figure 6.1)

$g_{1(i,j)}$: Information flow between the exit of i and the entry of j.

[1]In this chapter we relax the condition of a single confluence operator in the definition of a data flow framework.

Problem	Function types	Class $\mathcal{C}$
Reaching Def.	f^f, g_1^f	$< \{\text{OUT}\}, \{f^f, g^f(1000)\} >$
Live Variables	f^b, g_1^b	$< \{\text{IN}\}, \{f^b, g^b(1000)\} >$
MRA	f^b, g_1^b, g_1^f	$< \{\text{IN}\}, \{f^b, g^f(1000), g^b(1000)\} >$
LSIA	f^f, g_1^f, g_1^b	$< \{\text{OUT}\}, \{f^f, g^b(1000), g^f(1000)\} >$
MMRA	f^b, g_1^b, g_1^f, g_3^f	$< \{\text{IN}\}, \{f^b, g_\Pi^f(1000), g_\Sigma^f(0010), g_\Pi^b(1000)\} >$
EPA	f^b, g_1^b, g_3^f	$< \{\text{IN}\}, \{f^b, g_\Sigma^f(0010), g_\Pi^b(1000)\} >$
CHSA	f^f, f^b, g_1^b, g_1^f	$< \{\text{OUT}\}, \{f^f, f^b, g^f(1000), g^b(1000)\} >$

Table 6.1: Classification of data flow problems considered in this thesis

$g_{2(i,j)}$: Information flow between the exit of i and the exit of j.

$g_{3(i,j)}$: Information flow between the entry of i and the entry of j.

$g_{4(i,j)}$: Information flow between the entry of i and the exit of j.

As in chapter 3, the flow functions are superscripted by f or b to indicate whether the flow is in the forward or the backward direction. We refer to the functions by their type names f, g_1, g_2, g_3 and g_4 respectively with an appropriate superscript f or b. Table 6.1 lists the function types for the data flow problems defined in the appendix.

6.1.2 Classification of Data Flow Problems

We classify data flow problems according to the nature of the information flow paths. A class $\mathcal{C}$ is a tuple $<${*path origin*}, {*flow function types*}$>$ where :

(i) Path origin is a program point indicated by IN/OUT.

(ii) The node flow function types are represented by f^f/f^b. Edge flow functions are represented by $g(b_1b_2b_3b_4)$ where b_i is 1 if the corresponding flow function is $\neq \top$. For non-singular data flows, the function name is subscripted by the operator.

Table 6.1 contains the classification of the problems referred in the thesis. Note the dualism among the classes with respect to the path origin and the function types. For simplicity, we will name the classes by their representative data flow problems. Hence we will use the names appearing in the first column of Table 6.1 as the class names.

$$
\begin{aligned}
\mathrm{PPIN}_i &= \mathrm{PAVIN}_i \cdot (\,\mathrm{ANTLOC}_i + \mathrm{TRANSP}_i \cdot \mathrm{PPOUT}_i) \\
&\quad \cdot \sum_{p \,\in\, pred(i)} (\,\mathrm{PPIN}_p \cdot \neg\mathrm{ANTLOC}_p + \mathrm{AVOUT}_p) \\
\mathrm{PPOUT}_i &= \prod_{s \,\in\, succ(i)} (\,\mathrm{PPIN}_s)
\end{aligned}
$$

Figure 6.2: EPA equations.

Problem	Function types	Information flow paths
Reaching Def.	f^f, g_1^f	T_e^{f+}
Live Variables	f^b, g_1^b	T_e^{b+}
MRA	f^b, g_1^b, g_1^f	$(T_e^{b^+}(T_e^f \mid \varepsilon))^+$
LSIA	f^f, g_1^f, g_1^b	$(T_e^{f^+}(T_e^b \mid \varepsilon))^+$
MMRA	f^b, g_1^b, g_1^f, g_3^f	$(T_e^b \mid T_e^f)^+$
EPA	f^b, g_1^b, g_3^f	$(T_e^{b^+} \mid T_e^f)T_e^{f^*}(T_e^b \mid \varepsilon)$
CHSA	f^f, f^b, g_1^b, g_1^f	$T_e^b(T_e^b \mid T_e^f)^*$

Table 6.2: Information flow paths of some data flow problems.

The significance of classes lies in the fact that for a given graph, all problems in a class possess similar *ifp*'s and hence obey the same complexity bound. However, using the knowledge of a data flow problem it may be possible to develop a more specific definition of its *ifp*'s, and hence a more specific bound for it.

Example 6.1 : Consider the MMRA and EPA problems. MMRA, and other members of its class, have *ifp*'s which can be represented by the regular expression $(T_e^b \mid T_e^f)^+$. The *ifp*'s of the EPA class of problems are represented by the same regular expression. However, the *ifp*'s of the EPA data flow problem can be more precisely represented by $(T_e^{b^+} \mid T_e^f)(T_e^{f^*}T_e^b \mid T_e^{f^*})$. This is due to the nature of the EPA data flow (Figure 6.2), which restricts the backward flow resulting from the PPIN property turning BOT due to the g_Σ^f function merely to the OUT property of the predecessors (since the IN property of the predecessors is already BOT). Thus, we can have at most one backward edge traversal after a forward edge traversal. □

δ_f	:	Traversal along a forward edge in direction δ
δ_b	:	Traversal along a back edge in direction δ
δ_f^-	:	Traversal along a forward edge in direction δ^-
δ_b^-	:	Traversal along a back edge in direction δ^-
δ_G	:	Traversal over the graph in direction δ
δ_G^-	:	Traversal over the graph in direction δ^-

Figure 6.3: Generic notation for various traversals.

6.2 The Width of a Graph

We extend the notation of edge traversals to the traversals over the graphs. Thus T_G^f indicates a graph traversal in reverse postorder while T_G^b indicates graph traversal in postorder.[2] One traversal over the graph implies one iteration of the round-robin analysis. To simplify the presentation, we represent the forward and backward directions generically by δ. For example, if δ is the forward direction, then a T^f is replaced by δ while a T^b is replaced by δ^-. Thus $\delta_G = T_G^f$ if the graph traversal visits the nodes of a graph in a reverse postorder. Figure 6.3 summarises the generic notation.

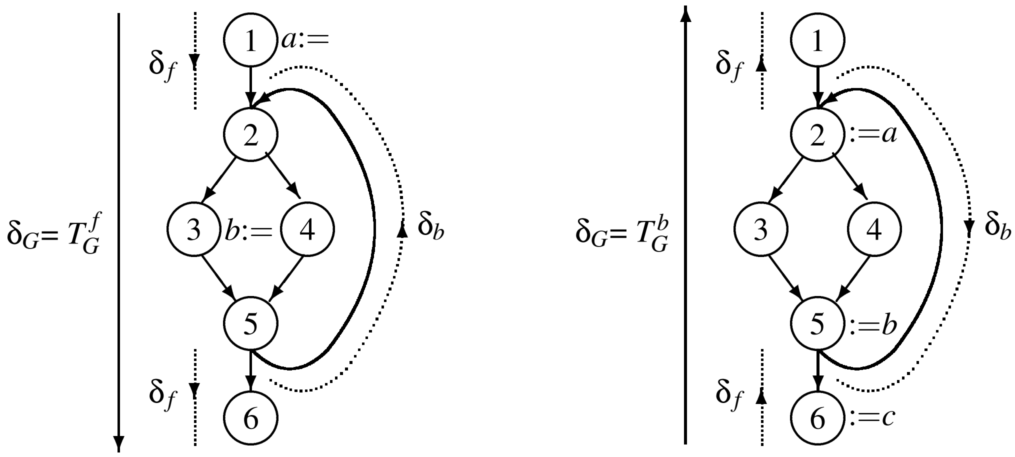
A graph traversal cannot realize the effect of all kinds of edge traversals. The following definition captures the relationship between edge and graph traversals.

Definition 6.1 : **Conforming and Non-conforming edge traversals**
For a δ_G traversal, δ_f and δ_b^- edge traversals are conforming edge traversals while δ_f^- and δ_b are non-conforming edge traversals.

A graph traversal realizes the effect of conforming edge-traversals, but fails to realize the effect of non-conforming edge traversals. This is illustrated in the following example.

Example 6.2 : Consider program flow graphs in Figure 6.4. For the reaching definitions analysis, the graph is traversed in reverse postorder. The fact that the definition of a in node 1 reaches all other nodes (via T_f^f edge traversals) is known in the first iteration over the graph. Similarly, the definition of b in node 3 is known to reach node 5 (T_f^f traversal) in the same iteration. However, the fact that this definition also reaches node 2 along the back edge (T_b^f traversal) is known only in the next iteration. Thus an T_b^f edge traversal is non-conforming whereas an T_f^f traversal is conforming.

[2] For a node i, T_G^f visits $in(i)$ followed by $out(i)$ while T_G^b visits $out(i)$ followed by $in(i)$.



Reaching Definitions Analysis   Live Variables Analysis

Figure 6.4: Conforming and non-conforming edge traversals

Analogous comments hold for the graph for live variable analysis with T^b replacing T^f. Assuming that the graph is traversed in postorder, the fact that b is live at $out(1)$ is known in the first iteration but its liveness at $out(5)$ is known in the next iteration only. □

An *ifp* consists of conforming and non-conforming edge-traversals.

Definition 6.2 : **Span**

A span is a maximal sequence of conforming edge traversals in an ifp.

Spans are separated by a non-conforming edge traversal and vice-versa. Thus, two successive non-conforming edge traversals have a *null span* between them. Further, an information flow path may begin and/or end with a null span.

The information along a span can be propagated in one δ_G traversal; the same graph traversal also realizes the information flow along the preceding non-conforming edge traversal.

Definition 6.3 : **Segment**

A segment is a maximal sequence of edge traversals in the same direction.

Successive T_e^f's constitute a *forward* segment while successive T_e^b's constitute a *backward* segment. A segment may be *bounded* or *unbounded*.

Example 6.3 : Consider a T_G^f graph traversal and an *ifp*

$$\overbrace{\underbrace{T_f^f\, T_f^f\, T_f^f}\;\underline{T_b^f}}\;\underbrace{\overbrace{T_b^b}\;\;\overbrace{T_f^f}}\;\overbrace{\underline{T_f^b}\;\underline{T_f^b}}$$

The underbraces denote the (non-null) spans, overbraces denote the segments, while underscores denote the non-conforming edge traversals. Note that there is a null span between the two successive T_f^b's. □

Example 6.4 : From Table 3.2, it is clear that the *ifp*'s of unidirectional data flow problems consist of a single unbounded segment. MRA and CHSA have *unbounded* backward segments. CHSA has unbounded forward segments too, while MRA has a bounded forward segment consisting of a single edge traversal. □

Definition 6.4 : **Information preserving path**
An ifp is an information preserving path (ipp) if all flow functions in the ifp are identity functions.

Definition 6.5 : **Clustered flow functions**
The edge flow functions of the type g^f (g^b) are said to be clustered *if the information flow is identical for all out-edges (in-edges) of a node.*

For an *ifp* $< u, v, \rho >$, let $length(\rho)$ and $width(\rho)$ denote the total number of edge flow functions and the number of edge flow functions along non-conforming edge traversals, respectively.

Definition 6.6 : **Bypassed information flow path**
An ifp $< p, q, \rho_1 >$ is said to be bypassed by $< p, q, \rho_2 >$ if the edge functions of p are clustered, and

(i) either ρ_2 is an ipp or $length(\rho_2)$ = 1, and

(ii) $width(\rho_2) < width(\rho_1)$.

Intuitively, $< p, q, \rho_1 >$ is bypassed by $< p, q, \rho_2 >$ if the same information is guaranteed to flow along ρ_2 and $length(\rho_2) < length(\rho_1)$.

In practical data flow problems, bypassing usually occurs due to $length(\rho_2) = 1$.

Definition 6.7 : **Width**
Width w of a graph G for a class of data flow problems with respect to a traversal δ_G is the maximum number of non-conforming edge traversals along an ifp, no part of which is bypassed.

If we represent the number of spans by s then $s = w + 1$ for the width determining path.

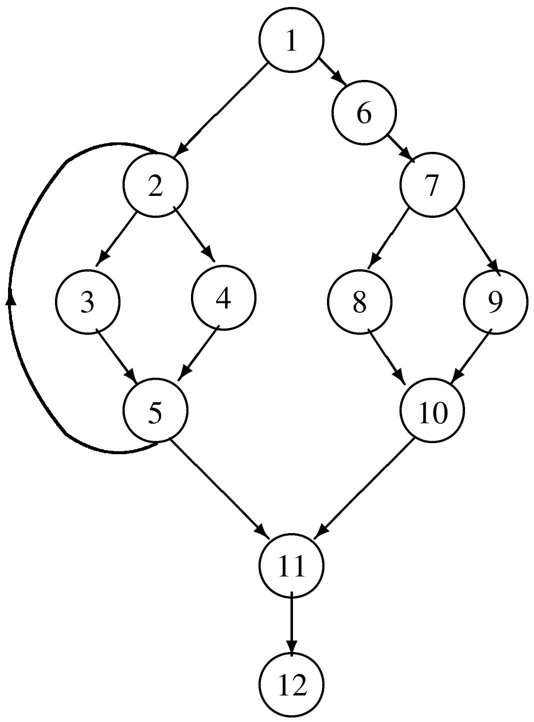
Theorem 6.1 : $w+1$ *iterations are sufficient for the round robin algorithm to converge on a fixed point.*

Proof : The information flow can be initiated only after the IN and OUT properties of all nodes are computed to determine the information originating within the nodes (i.e. the information represented by TR_0). This can be achieved in the first iteration. The same iteration also realizes the propagation of information along a non-null span (if any) at the beginning of an *ifp*. However, every non-conforming edge traversal, and the span that follows it, requires a separate iteration. Thus, $w+1$ iterations are sufficient for information propagation along the width determining path.

Now consider an *ifp* $< p, q, \rho_1 >$ such that $width(\rho_1) > w$. This is possible only if a section ρ' of ρ_1 is bypassed by another *ifp* ρ''. Let $width(\rho')$ be w', $width(\rho'')$ be w'' and $width(\rho_1)$ be w_1. Then $(w_1 - w') + w'' \leq w$. Again $w+1$ iterations suffice for information to propagate from p to q. $\square$

Note that the bound $w+1$ is a static prediction. For a particular instance of a data flow problem, the number of iterations could be less as the behaviour of the non-preserving *ifp*'s (i.e. *ifp*'s containing functions of the form $h(X) = A + \neg B \cdot X$) is governed by the constants A and B for that instance.

Example 6.5 : Consider the following graph



We choose $\delta_G = T_G^b$, for MRA, EPA and MMRA. T_b^b and T_f^f are the non-conforming edge traversals. The width determining paths for these problems are :

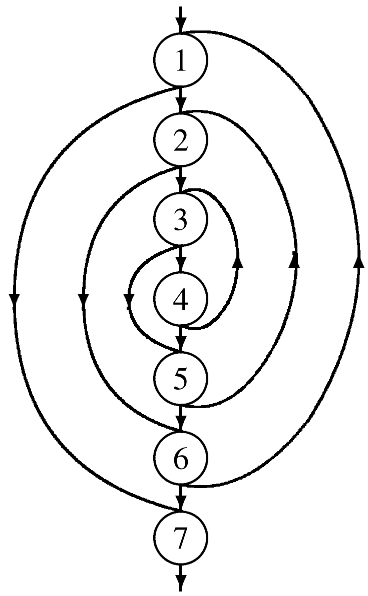
- MRA : $(6,1,2,5,11,10,9,7,8) = T_f^b T_f^f T_b^b T_f^f T_f^b T_f^b T_f^b T_f^f$
- EPA : $(1,6,7,8,10,11,12) = T_f^f T_f^f T_f^f T_f^f T_f^f T_f^f$
- MMRA : $(1,6,7,8,10,11,12) = T_f^f T_f^f T_f^f T_f^f T_f^f T_f^f$

The widths for these problems are 4, 6, and 6, respectively. □

6.3 The Width and the Depth

Depth (d) is defined as the maximum number of back edges along any acyclic path [2]. To use the notion of width for unidirectional flows, choose δ as the natural direction of the flow in the problem. There are no flows along δ_f^- and δ_b^- edge traversals, and the only non-conforming edge traversal is δ_b. Since the width considers only those paths which do not have bypassed fragments, $w \leq d$. Thus, width provides a tighter bound on the number of iterations.

Example 6.6 : Consider a *spiral graph* [7] whose depth increases linearly with the nesting depth.[3]



Here $d = 3$, while $w = 1$ for a unidirectional problem with clustered edge flow functions since every part of an *ifp* beginning on a back edge is bypassed, *viz.* path $(5,2,6)$ is bypassed

[3]Spiral structures result from **repeat** … **until** loops with multiple exits.

by the path $(5,6)$. This explains the observation that the number of iterations remains constant even as the size of the spiral graph grows. □

Further, the notion of depth assumes a fixed pattern for information flow governed by the directed paths in the flow graph, hence it is only applicable to unidirectional data flow problems.

6.4 Efficiency of Data Flow Analysis

The notions of bounded and unbounded segments yield several insights when applied to the efficiency of data flow analysis. In particular, the generalised theory :

- motivates efficient solution techniques *viz.* interval analysis technique for MRA [23], and the method of alternating iterations,
- explains several known results in bidirectional flows.

6.4.1 Choice of Direction in graph traversal

Consider a data flow problem whose *ifp*'s have unbounded segments in one direction and bounded segments in the other direction. Recall that $w = \#\delta_b + \#\delta_f^-$, i.e. width has contributions from the back edges in the δ segments and forward edges in the δ^- segments. Typically, forward edges outnumber the back edges. Hence, $\#\delta_f^-$ is likely to be smaller when the segments in the δ^- direction are bounded rather than unbounded. Hence the appropriate direction for graph traversal is the one that makes the bounded segments lie in the δ^- direction, and unbounded segments, in the δ direction.

Example 6.7 : MRA has unbounded backward segments but bounded forward segments, hence backward graph traversal would require fewer iterations than forward traversal. For unidirectional problems, the favoured direction of traversal is trivially the direction of the data flow. □

Alternating Iterations

For problems with unbounded segments in both directions, alternating the direction of graph traversal between successive iterations can effectively reduce the solution complexity. This can be explained as follows : A segment in the δ direction may consist of a number of spans separated by non-conforming edges, which may themselves form sizable spans in the δ^- direction. Since the effect of a span in the δ direction is incorporated by a single iteration in the δ direction, alternating the direction of graph traversal between successive iterations would yield better results. Thus, the alternating iterations approach is clearly warranted in the case of CHSA.

Let s_ρ be the number of non-null spans along an *ifp* ρ. The width of ρ for alternating iterations is defined as follows : $width_a(\rho) = 2s_\rho + c$, where $c = 1$ if ρ ends with a non-null span, else $c = 0$. The number of iterations for the method of alternating iterations is then $w_a + 1$ where w_a is defined analogous to w, *viz.* $w_a = \max(width_a(\rho))\ \forall\rho$, where ρ is an *ifp*, no part of which is bypassed.

Note that this is advantageous only if there are many null spans. i.e. if the number of non-null spans is small. In particular if $s_\rho > s/2$ then $w_a \geq w$ and alternating iterations may actually require a higher number iterations for convergence on MFP. The requirement that there should be many null spans implies that there should be many successive non-conforming edge traversals for the method of alternating iterations to be beneficial. Such a possibility is enhanced by the existence of unbounded segments in δ^- direction. This is in consonance with the fact that if we perform round robin analysis for available expressions with postorder traversal over the graph (i.e. T_G^b), then we need more iterations than the number of iterations required when we alternate the direction of graph traversal between successive iterations. In the former case, we traverse the graph against the direction of information flow in all the iterations; in the latter case we traverse the graph against the direction of flow in alternate iterations only.

6.4.2 Reducing Complexity by Reducing Width

When a data flow problem has bounded segments in one direction, and unbounded segments in the other direction, complexity of the data flow analysis can be reduced by attempting to *truncate* the information flow paths. Consider an *ifp* $\rho = (\cdots, \bar{e}_i, \bar{e}_{i+1}, \bar{e}_{i+2})$, where edge $\bar{e}_{i+1}$ constitutes a bounded segment of length 1, i.e. $\bar{e}_i$ and $\bar{e}_{i+2}$ belong to the segments in the opposite direction. Truncation can be effected by transforming the program flow graph or the data flow equations so as to terminate each *ifp* analogous to ρ *before* it reaches the edge $\bar{e}_{i+2}$. Effectively, ρ is split into two *ifp*'s ρ_1 and ρ_2. Since $width(\rho_1), width(\rho_2) \leq width(\rho)$, this could reduce the width. In this section, we present three transformations based on this approach.

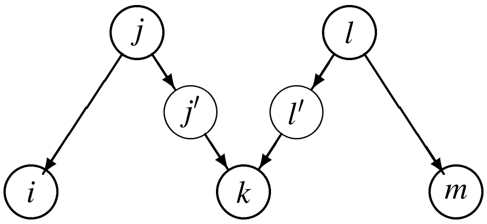


Figure 6.5: Edge splitting

Edge Splitting

An edge which runs from a *branch node* (i.e. a node with more than one successor) to a *join node* (i.e. a node with more than one predecessor) is called a *critical edge*. It has been reported [25] that when such an edge is split by inserting a new node, the solution complexity of MRA is reduced. The following lemma captures the influence of the edge splitting graph transformation on the solution complexity of MRA.

Lemma 6.1 : *Following edge splitting, MRA can be solved with the complexity of a unidirectional problem.*

Proof : Following section 6.4.1, we choose as $\delta_G = T_G^b$ for MRA. Consider the information flow path from node i to node m ($\rho = (i, j, k, l, m) = T_f^b T_f^f T_f^b T_f^f$) in figure 6.5. Before edge splitting, it would have contained two δ_f^- (i.e. T_f^f) edge traversals. With the insertion of j', ρ is split into two *ifp*'s $\rho_1 = (i, j, j')$ and $\rho_2 = (k, l', l, m)$, which contain a single δ_f^- traversal each. (This is because a $\rho' = (i, j, j', k, l', l, m)$ is not an *ifp* for MRA since two successive δ_f^- traversals (j, j') and (j', k) cannot exist in *ifp*'s for MRA). In general, edge splitting restricts the number of δ_f^- edge traversals along any information path to ≤ 1 in the transformed graph.[4] Thus $w \leq w' + 1$, where w' is the width of the graph for a unidirectional data flow problem. □

As a result of edge splitting, the MRA *ifp*'s become $T_e^{b^+}(T_e^f \mid \varepsilon)$ for the transformed graph, instead of the original $(T_e^{b^+}(T_e^f \mid \varepsilon))^+$.

Edge Placement

The technique of *edge placement* eliminates a partial redundancy of an expression e in node i, which cannot be safely hoisted into a predecessor j, by creating a synthetic node along the edge (j, i) and hoisting e into it [17]. Unlike edge splitting, however, a synthetic node is conceptual; it does not participate in data flow analysis. It becomes real only when a computation

[4] < 1 in the original graph.

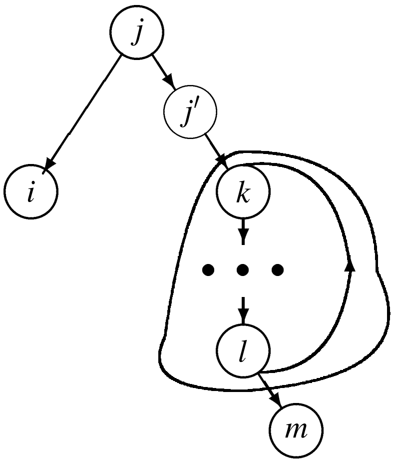


Figure 6.6: Interval based solution for MRA

is inserted in it during optimisation phase following the data flow analysis.

Use of edge placement in MRA results in elimination of the Π term from the PPIN$_i$ equation (eq. 1.1 of figure 6.2). It thus transforms the data flow, rather than the flow graph, to achieve the same effect as edge splitting, *viz.* restricting the number of δ_f^- traversals along any *ifp* to < 1. Solution efficiency vis-a-vis MRA is guaranteed by the fact that the resulting data flow is simply (backwards) unidirectional in nature.

Interval Pre-headers

[24] mentions that an interval based elimination method can not be extended to MRA. The notion of width can be used to justify this, and to motivate an edge splitting graph transformation which enables an elimination approach to MRA solution. Consider a forward edge (j,k) terminating on the interval header (figure 6.6). The information flowing forward along this edge can flow (backwards) along the latching edge (k,l) of the interval to reach the exit edge (l,m) of the interval. Thus, information can flow forward 'through' an interval, as well as back through it due to the mostly backward nature of MRA.

However, if we introduce an interval pre-header by splitting every inter-interval edge (i.e. every edge in a reduced graph), we will effectively truncate an information flow path before it reaches an interval-header along a forward edge, *viz.* the path $\rho = (i, j, k, l, m)$ would be truncated to $\rho_1 = (i, j, j')$. This restricts the information flow between two intervals to a unidirectional flow. Thus the bidirectional problem is partitioned into smaller sub-problems related to each other by a unidirectional data flow. This makes the classical interval analysis approach feasible for MRA.

6.5 Decomposing Bidirectional Flows into Unidirectional Flows

Decomposition of a bidirectional data flow problem into a sequence of unidirectional problems (i.e. solving a bidirectional problem as a sequence of cascaded unidirectional problems) is motivated by the desire to reduce the amount of work or to improve the understandability of the data flow involved. Prior work on decomposition has been *ad hoc* and/or directed at specific bidirectional data flow problems [17, 25, 40]. In this section we provide a condition for the decomposability of a bidirectional data flow problem.

Observation 6.1 : *For a program graph G, $w^u \leq w^b$ where w^u and w^b are the widths of G for arbitrary unidirectional and bidirectional data flows with respect to δ_G such that δ_G is along the natural direction of data flow for the unidirectional problem.* □

Lemma 6.2 : *It is feasible to decompose a bidirectional data flow problem into a sequence of unidirectional data flow problems if and only if the number of segments in every information flow path for the data flow problem is bounded by a constant.*

Proof : Let δ be the direction of the first segment in an information flow path. Information propagation along the segment can be realized by a unidirectional data flow problem which has δ as its natural direction of flow. Information flow along the following segment would require a unidirectional problem in the opposite direction, etc. Thus, the number of unidirectional problems required will equal the number of segments, which should be bounded by a constant for the decomposition to be feasible. Further, the order of solving the unidirectional problems will have to be the same as the order of segments in the information flow paths. □

Corollary 6.1 : *MRA cannot be solved by cascaded unidirectional problems.*[5]

The *ifp*'s of MRA have the form $(T_e^{b+}(T_e^f \mid \varepsilon))^+$, thus they can have an unbounded number of forward and backward segments. Similar statements hold for the LSIA and CHSA problems.□

In the following, we explain two decompositions reported in literature, and motivate a third one.

Corollary 6.2 : *It is possible to decompose MRA if edge splitting is performed.*

Following section 6.4.2, the information flow paths in the resulting program flow graph can be characterised by the regular expression $T_e^{b+}(T_e^f \mid \varepsilon)$. Since the number of segments can at most be 2, it is possible to solve MRA by cascaded unidirectional problems. Further,

[5]unless edge splitting is performed.

since the second (i.e. the forward) segment has a length ≤ 1, it is possible to solve MRA on a graph in which critical edges have been split as a backward problem followed by a *forward correction* [25].□

Corollary 6.3 : *EPA can be decomposed into cascaded unidirectional problems.*

The decomposition of the Edge Placement Algorithm(EPA) [17] is feasible since its information flow paths are characterised by $(T_e^{b^+} \mid T_e^f)T_e^{f^*}(T_e^b \mid \varepsilon)$ (refer table 6.2). The number of segments in an *ifp* can at most be 3, hence the condition of lemma 6.2 is satisfied. A decomposition for EPA is presented in [17, 24].□

Corollary 6.4 : *It is possible to decompose MMRA if edge splitting is performed.*

The *ifp*'s of MMRA incorporate two distinct effects :

(i) The mostly backward propagation of MRA which leads to *ifp* sections described by the regular expression $(T_e^{b^+}(T_e^f \mid \varepsilon))^+$, and

(ii) The forward propagation of EPA which leads to *ifp* sections described by the regular expression $T_e^{f^*}(T_e^b \mid \varepsilon)$.

Edge splitting truncates the *ifp* sections of the first kind to simply $T_e^{b^+}(T_e^f \mid \varepsilon)$ in the transformed flow graph, and $T_e^{b^+}$ in the original graph, respectively (refer lemma 6.1). The resulting *ifp*'s are described by the regular expression $(T_e^{b^+} \mid T_e^f)T_e^{f^*}(T_e^b \mid \varepsilon)$ in both the transformed as well as the original graphs. Since this is the same as the regular expression for EPA in corollary 6.3, the decomposability of MMRA follows. The resulting decomposition is described in [26].□

This result in not surprising since EPA differs from MMRA only in the use of the edge placement technique (refer appendix) which has already been shown to achieve an effect equivalent to edge splitting in the original program flow graph (refer section 6.4.2).

Corollary 6.5 : *It is not possible to decompose CHSA even if edge splitting is performed.*

Since both forward and backward segments are unbounded for CHSA, edge splitting does not truncate any *ifp* and the *ifp* pattern remains $T_e^f(T_e^b \mid T_e^f)^*$. Hence the number of segments remains unbounded.□

6.6 Results and Conclusions

The results of analysing a suit of scientific programs written in Fortran-77 are presented in appendix C. The following observations can be made concerning these results :

(i) Very good correlation is seen between the width and the number of iterations for the problems which have bounded segments in the δ^- direction (*viz.* MRA).

(ii) For EPA and MMRA, the predicted and observed widths for backward iterations differed by very large margins. This can be attributed to the following :

  (a) Width is a static prediction. Many of the long *ifp*'s traced during width computation may not be realized during data flow analysis. This fact is well accepted by practitioners of data flow analysis.

  (b) The *ifp*'s of EPA and MMRA have unbounded δ^- segments. Hence, the computed widths are very large. This accentuates the effect mentioned in (a) above. A similar effect is observed in the case of unidirectional problems having unbounded δ^- segments. For example, the problem of available expressions has unbounded δ^- segments when solved using backward iterations. Its width is therefore the number of edges along the longest acyclic forward path in the program. In practice, information does not propagate so much.

(iii) For problems with unbounded segments in both directions (*viz.* EPA and MMRA), the method of alternating iterations scored over the backward direction of graph traversal by a large margin in a total of 11 cases. In 10 cases, it performed only marginally worse than backward traversal. On the whole, alternating iterations should be the favoured solution method for EPA and MMRA. When compared to MRA, the additional complexities of data flow in EPA and MMRA do not appear to make them very expensive in practice. (Note that MRA is also known to take up to 5 iterations in practice [45]).

(iv) Despite observation (ii), observation (iii) above demonstrates the applicability of the concept of an information flow path to non-singular data flows as well, thereby vindicating the use of *ifp*'s as the basis of a study of the complexity of data flow analysis.

6.7 Looking Back

Classical data flow analysis uses two key features : *properties of graph structures* and *patterns of information flow*. Different methods blend and use these features in different ways. Elimination methods use graph regions to divide the data flow problems into smaller subproblems, while round-robin methods follow postorder or reverse postorder for graph traversal. Worklist versions do not use properties of the graphs but follow the data flow pattern dynamically.

These features have traditionally been used in isolation. We combine them to evolve the notions of the width and the related concepts which provide valuable insights in the complexity and efficiency of data flow analysis. Again, these results are uniformly applicable to unidirectional and bidirectional data flow analysis.

Part II

Incremental Data Flow Analysis

Chapter 7

Approaches to Incremental Data Flow Analysis

"Men," said the little prince, "set out on their way in express trains, but they do not know what they are looking for. Then they rush about, and get excited, and turn round and round ..."
And he added : "It is not worth the trouble."

When a program undergoes change during development or compilation (i.e. during the optimisation phase), the data flow information can be updated either by repeating the original exhaustive analysis, or by using incremental techniques that attempt to reuse the old information and recompute only the information affected by the change. Presuming that a change typically has a localised effect, or at least does not affect all of the program's data flow, incremental algorithms are more cost-effective than the exhaustive algorithms.

This chapter surveys the traditional approaches to incremental data flow analysis. The problems with the traditional paradigm are discussed and a modified paradigm is proposed which paves the way for an algorithm-independent theoretical discussion of incremental data flow analysis in the succeeding chapters.

7.1 A Paradigm for Incremental Computation

Let $\mathcal{P}$ be a problem, $\mathcal{A}$, an algorithm to solve $\mathcal{P}$, and $\mathcal{S}$, the resulting solution. Then, their interrelationship can be described by $\mathcal{S} = \mathcal{A}(\mathcal{P})$. Let $\mathcal{P}$ be modified such that the new $\mathcal{P}$, denoted $\mathcal{P}'$, is $\mathcal{P}' = \mathcal{P} + \Delta\mathcal{P}$.[1] The corresponding solution, $\mathcal{S}' = \mathcal{A}(\mathcal{P}')$, is $\mathcal{S}' = \mathcal{S} + \Delta\mathcal{S}$. Computing $\mathcal{S}'$

[1]The operation $+$ should be taken to mean "combination" i.e. "$\Delta\mathcal{P}$ *combines* with $\mathcal{P}$ to give $\mathcal{P}'$."

using $\mathcal{A}$ amounts to *exhaustive computation*. Instead, if it is possible to devise an algorithm $\mathcal{A}^I$ such that $\mathcal{A}^I(\mathcal{S},\Delta\mathcal{P})$ computes $\Delta\mathcal{S}$ and combines it with $\mathcal{S}$ to give $\mathcal{S}'$, then the computation $\mathcal{S}' = \mathcal{A}^I(\mathcal{S},\Delta\mathcal{P})$ is an *incremental computation*.

7.2 Traditional Approaches to Incremental Data Flow Analysis

Incremental data flow analysis has received a considerable attention lately. There is a plethora of algorithms in the literature, both elimination [8, 11, 54, 55, 57], as well as iterative [13, 14, 30, 49]. [43] falls into both the categories while [63] falls into neither. A comparison or various approaches is contained in [9].

7.2.1 Iterative Methods

The early efforts in iterative incremental methods were based on the technique of *restarting iteration* [13, 14, 30] which really is as simple as the name suggests : Given the set of changes and the old solution, this technique computes the new solution by propagating the changes through the affected area of the flow graph. Unaffected areas of the graph are not examined as would be required by an exhaustive recomputation.

Before restarting iteration, the solution at the immediately affected nodes is altered. Given these adjusted solutions at the immediately affected nodes and the old solution at other nodes, iterative analysis is restarted and repeats till a fixed point is reached. It is easy to see that this fixed point is not always the MFP [9]. Since any change can only be towards $\perp$, the properties are either changed to BOT or remain same.[2] As a result of function changes, a property may very well change from BOT to TOP. The influence of such a change can obviously not be propagated to other nodes; the properties at other nodes will remain BOT. Thus the calculated solution at a node cannot be higher than the value that it assumes before the iteration is started. Given an arbitrary set of changes to the data flow functions, it is possible for the new MFP solution at a node to be higher than previous MFP solution (and higher than in the restart configuration). In such a case, the incrementally computed fixed point will be lower than the (desired) fixed point. [4] presents a general condition under which restarting iteration results in the same solution as an exhaustive iterative analysis; however, this result is a sufficient

[2]TOP is the value of a bit in the bit vector representing the $\top$ element of the lattice while BOT is the value of a bit in the bit vector representing the $\perp$ element of the lattice. See chapter 3 or 8 for the details.

condition, not a necessary one. Counter examples to restarting iteration are also presented.

Pollock and Soffa [49] present an incremental iterative algorithm which yields the maximum fixed point after a program change. This algorithm is a combination of two different techniques : one for changes where restarting iteration will yield precise results and another where it will not. The latter involves reinitialising lattice values at nodes so that those facts potentially affected by the program change assume their original initialised value TOP. Then iteration is started from those nodes where direct changes are observed and continues until a new fixed point is obtained. Although Pollock and Soffa explain their algorithm in terms of the classical union and intersection problems, it is clearly of a wider utility. It handles all changes to a flow graph, including *structural changes* (i.e. edge and node insertions and deletions), although those involving node changes are not explicitly outlined.

Since it is developed in the context of editing, the method heavily depends on an exhaustive case analysis of each change and its possible implication. This approach makes it a little difficult to apply this method to new data flow problems.

7.2.2 Elimination Methods

Modelling the data flow problem by a system of linear equations, Ryder developed methods for incremental data flow analysis [54, 57, 55] including Allen-Cocke interval analysis [3], Tarjan interval analysis [59], and the Hecht-Ullman T1-T2 technique. The incremental update algorithms consist of two phases, namely recalculating all coefficients and constants in all data flow equations that are affected by the program change, and then recalculating all affected solutions. The original work was limited to handle only non-structural program change; however, recently the work was extended to handle structural changes [11] using an immediate dominator decomposition of the flow graph. Based on the elimination methods, the work is not easily applicable in an environment where program changes may result in irreducible flow graphs.

Burke reformulates interval analysis so that it can be applied to any monotone data flow problem including the nonfast problems of flow-insensitive inter-procedural analysis for exhaustive as well as incremental computation [8]. With a single update, the incremental algorithm can accommodate any sequence of program changes that do not alter the structure of program call graph; other structural changes are accommodated in the algorithm. The paper also presents an incremental algorithm for alias analysis that obtains the exact solution as computed by an exhaustive algorithm.

Among other elimination based methods, Rosen has developed an elimination technique based on finding the *flow cover* of a flow graph; he has shown that, in principle, his technique is applicable in the presence of small changes to the flow graph [50].

Though the iterative techniques are simpler to understand and code than the elimination techniques, the limitations of these techniques have not been explained. In contrast, the elimination methods are more formal and are usually proven correct; also their breadth of applicability is clearly specified.

7.2.3 Other Methods

Marlowe and Ryder's hybrid incremental technique [44] uses an elimination-like flow graph decomposition, a data flow solution factorisation on component regions, and solution on regions by fixed point iteration. More specifically, the main idea is to factor the data flow solution on strongly connected components of the flow graph into local and external parts, solving for local parts by iteration and propagating these effects on the condensation of the flow graph to obtain the entire data flow solution. The incremental hybrid algorithm re-performs those algorithm steps affected by the program changes. The algorithm handles all changes for the common distributive intra-procedural and inter-procedural data flow problems and can deal with irreducibilities.

Zadeck's incremental technique [63] is applicable to data flow problems that are *partitionable* into a series of independent problems called *clusters*. This technique is called the *partitioned variable technique* (PVT, for short) since a different cluster must be solved for each variable in the program. This avoids the added complexity of cycles in the flow graph. Focussing on the update of a single cluster, the flow graph is manipulated according to information contained within each block. The graph is split at each block that stops the propagation of variable's information chain. This ensures that any cycle in the resulting graph will have the property that information reachable from *any* block in the cycle is reachable from *every* block in the cycle since the cycle can be executed any number of times. Thus the strongly connected regions of the flow graph can be collapsed to form a directed acyclic graph. Information is propagated globally on the return from a depth first traversal starting at the blocks that end information chains. The incremental algorithm calculates changes to affected clusters and performs a reinitialised recomputation on them. It handles all program changes (including structural changes) for partitionable problems.

7.3 Limitations of the Traditional Approaches

7.3.1 The Cause

Almost all traditional approaches to incremental data flow analysis are based on the paradigm defined in section 7.1 :

> An algorithm for incremental data flow analysis ($\mathcal{A}^I$) takes old solution ($\mathcal{S}$) and the changes in the old instance of a data flow problem ($\Delta\mathcal{P}$) as input, manipulates the old solution ($\mathcal{S}$) and produces the new solution ($\mathcal{S}'$).

The key feature of the traditional approaches is that they *modify* the old solution by applying some technique. The expected change in the old solution is not defined explicitly.

We modify the traditional paradigm to define :

$$\left.\begin{array}{rcl} \Delta\mathcal{S} & = & \mathcal{A}^I(\mathcal{S},\Delta\mathcal{P}) \\ \mathcal{S}' & = & \mathcal{S}+\Delta\mathcal{S} \end{array}\right\} \tag{7.1}$$

Paradigm 7.1 breaks up the task into two steps by *explicitly* defining the computation of $\Delta\mathcal{S}$ which is later combined with $\mathcal{S}$ to produce $\mathcal{S}'$. The traditional paradigm mixes the two issues by defining $\Delta\mathcal{S}$ *implicitly*. No algorithm for incremental data flow analysis defines exactly what constitutes the *incremental* part of an overall solution — all of them rely on the definition of the *final solution* which is directly carried over from exhaustive methods (and the corresponding theory).

It must be admitted that the lack of a concrete definition $\Delta\mathcal{S}$ is an outcome of the primary concern of incremental computation — achieving efficiency. It is only too natural to devise incremental methods which would directly manipulate $\mathcal{S}$. Unfortunately, the formal abstractions which form the basis of such methods remain ad hoc and restrictive.

Apart from efficiency considerations, the other reason due to which the need of defining $\Delta\mathcal{S}$ was not felt, has been the widely held belief that incremental computation is an offshoot of exhaustive computation [8, 43, 57, 63]. This is best exemplified by the "rule of thumb" suggested by Barry Rosen in his pioneering paper on theorising incremental computation [53] :

> "For sake of clarity and reliability, it is good to arrive at incremental algorithms indirectly. Begin with an exhaustive algorithm ALG and argue for (perhaps even prove) its correctness. Convert ALG for incremental use by some systematic changes (not necessarily the ones that we have studied) that preserve correctness, and tidy up the result."

It can be easily verified from section 7.2 that almost all methods are developed as refinements of the corresponding exhaustive methods.

7.3.2 The Consequences

The formal abstractions of the solutions of a data flow problem (*viz.* the notions of MFP/MOP solutions), are pure mathematical concepts not restricted to any particular algorithm. While the same definitions are used for incremental data flow analysis, the incremental algorithms are devised with a goal of actually computing something much less than these complete solutions.[3] So what exactly is an incremental algorithm expected to compute? The traditional approaches do not attempt to provide a general answer independent of any particular algorithm. Consequently, two important issues in theory and practice of data flow analysis are adversely affected :

- Generality.

  The approaches remain ad hoc. No theoretical explanation of incremental data flow analysis is possible. All explanations remain specific to a particular algorithm. It is clear from section 7.2 that the literature is abundant with *techniques* for incremental data flow analysis; there is little or no *theory*.[4]

  Though some common trends can certainly be seen, they do not in any way help to argue about the applicability of an algorithm to a new data flow problem.

  - One of the most common trends is handling of some kind of a region, usually a strongly connected region, in a special way [8, 10, 42, 11, 55, 57, 63]. This is done in elimination as well as iterative algorithms (*viz.* [49]). Such algorithms cannot be possibly extended to bidirectional data flow problems *viz.* MRA since in such problems, the self-dependence of properties may arise even without the existence of a strongly connected region.
  - Another common trend is *change/implication classification* which heavily depends on a comprehensive case analysis[5] which in turn cannot be performed without a

[3] If they are not devised with this goal, they are not incremental.

[4] This should be contrasted with the exhaustive data flow analysis. Chapter 2 is dominated by theory, and not by methods.

[5] Its origin lies in the fact that most incremental algorithms were conceived in the context of interactive programming environments (or simple syntax-directed editors).

thorough knowledge of what is being solved. Thus, extending these algorithms [8, 43, 49, 63] to new problems is likely to be error-prone.

Despite some common trends, the approaches remain isolated, and the insights developed in one approach may not help much in other approaches.

- Correctness.

  The incremental part of the solution cannot be predicted theoretically, it is known only retrospectively. Any attempt to define $\Delta\mathcal{S}$ in the traditional approaches results in *algorithmic* or *operational* definitions. Though some elimination techniques show the correctness formally [57], most others (presumably following the "rule of thumb" mentioned in section 7.3.1), assume that correctness is preserved as one systematically modifies an exhaustive algorithm to create an incremental version. Sometimes a case for correctness is made through an appeal to intuition by discussing several examples. Often, showing correctness remains a matter of empirical testing.

7.4 Towards a Functional Abstraction

We conclude this chapter by another quote from Barry Rosen on his efforts of theorising incremental data flow analysis in [53] :

> "Better verbalizations can be expected, as theoreticians realize that exhaustive analysis is not the only kind needing careful mathematical treatment."

This goal cannot be achieved through the traditional paradigms. What is desired is a *functional abstraction* (totally devoid of any procedural or algorithmic element) of the process of incremental data flow analysis. This can only be achieved if a formal mechanism is devised which answers the following question : What *change* should be expected in the solution, following a change in an instance of **D**? Importantly, it should be possible to answer this question

(i) without requiring any algorithm, and

(ii) without the need of knowing the resulting MFP solution since computing the MFP solution is the very goal of answering this question.

This part of the thesis tries to provide such a mechanism.

Chapter 8

Background

My drawing was not a picture of a hat. It was a picture of a boa constrictor digesting an elephant. But since the grown-ups were not able to understand it, I made another drawing : I drew the inside of the boa constrictor, so that the grown-ups could see it clearly. They always need to have things explained.

In order to make the two parts as independent as possible, this chapter reviews those concepts of part I which are used in our treatment of incremental data flow analysis in chapter 9. Not all concepts defined in the generalised theory are used in this part and those which are used, are scattered throughout part I. This chapter puts them all in one place and prepares a cohesive background. Some of the concepts and notations have been altered to suit to the work presented; section 8.6 lists these changes.

8.1 Data Flow Frameworks

A *data flow framework* is defined as a triple **D** = $<\mathcal{L}, \sqcap, \mathcal{F} \cup \mathcal{G}>$. Elements in lattice $\mathcal{L}$ represent the information associated with the entry/exit of a basic block. Thus, each element in $\mathcal{L}$ represents a set of information associated with the entry/exit of a basic block. $\sqcap$ is the set union (alternatively, boolean SUM denoted by Σ) or intersection (alternatively, boolean PRODUCT denoted by Π) operation which determines the way the global information is combined when it reaches a basic block. $\mathcal{F}$ is the set of *node flow functions* while $\mathcal{G}$ is the set of *edge flow functions*.

An *instance* **I** of a data flow framework **D** is defined by the ordered pair **I** $=< G, M >$.

- G is the *control flow graph* $G =< N, E, entry(G), exit(G) >$ where N is the set of nodes

$\propto(p)$	:	The set of properties corresponding to p.
$\Omega(p)$	:	The set of functions influencing the value of p.
$\mathcal{D}(p)$	:	The set of properties which p depends on for its value.
$\mathcal{P}p(p)$	:	Program point of p.
$\mathcal{N}(p)$	:	The set of neighbouring properties of p (i.e. the properties which are influenced by p).
$\mathcal{N}^{-1}(p)$	:	Inverse of $\mathcal{N}(p)$ (i.e. the properties which influence p).[a]

[a]The notation $\mathcal{N}$ and $\mathcal{N}^{-1}$ is also extended to program points.

Figure 8.1: Various entities for a property p.

(i.e. basic blocks), E is the set of edges, and $entry(G)$ and $exit(G)$ denote the (non-null) sets of graph entry and exit nodes, i.e. nodes with zero in-degree and zero out-degree, respectively.

A *program point* refers to the entry/exit of a basic block. For a basic block i, its entry and exit points are denoted by $in(i)$ and $out(i)$ respectively. Program point u is a *neighbour* of program point v if u and v are adjacent in the underlying undirected graph of G and the information at u is influenced by the information at v. Thus $in(j)$ is a neighbour of $out(i)$ where $j \in succ(i)$, and $out(j)$ is a neighbour $in(j)$ for a forward data flow problem.

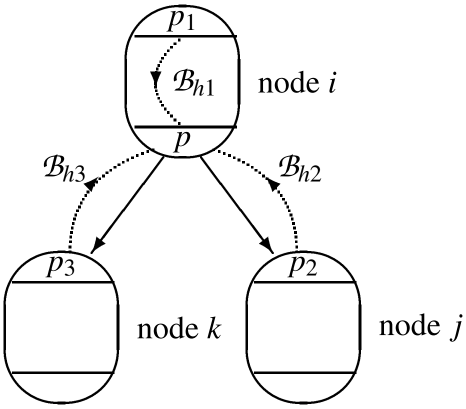
- $M \equiv < M_{\mathcal{F}}, M_{\mathcal{G}} >$ such that $M_{\mathcal{F}} : N \rightarrow \mathcal{F}$ maps the nodes to the node flow functions and $M_{\mathcal{G}} : E \rightarrow \mathcal{G}$ maps the edges to the edge flow functions.

  In our treatment, we drop this mapping and directly subscript the functions by program points.[1]

8.2 Data Flow Properties

Data flow properties represent the information to be gathered during data flow analysis. When the sets of information are implemented as bit vectors, each bit represents a data flow property

[1]This is different from part I where the functions are subscripted by nodes and edges rather than program points. Thus, a flow function corresponding to node i is denoted by f_i while a flow function corresponding to edge (i, j) is denoted by $g_{(i,j)}$ in part 1.4.3. Use of program point facilitates a uniform treatment of the edge and node flow functions in that f_i is denoted by $h_{(u,v)}$ where u and v are the end points of node i while $g_{(i,j)}$ is denoted by $g_{(v,w)}$ where v and w are the end points of the edge (i, j).



$\mathcal{P}p(p) = out(i),\;\; \mathcal{P}p(p_1) = in(i)$
$\mathcal{P}p(p_2) = in(j),\;\; \mathcal{P}p(p_3) = in(k)$
$\mathcal{N}^{-1}(p) = \{p_1, p_2, p_3\}$
$p \in \mathcal{N}(p_1),\; p \in \mathcal{N}(p_2),\; p \in \mathcal{N}(p_3)$
$\mathcal{N}^{-1}(out(i)) = \{in(i), in(j), in(k)\}$
$out(i) \in \mathcal{N}(in(i))$
$p \leftarrow \mathcal{B}_{h1}(p_1),\;\; \mathcal{B}_{h1} \in \mathcal{F}$
$p \leftarrow \mathcal{B}_{h2}(p_2),\;\; \mathcal{B}_{h2} \in \mathcal{G}$
$p \leftarrow \mathcal{B}_{h3}(p_3),\;\; \mathcal{B}_{h3} \in \mathcal{G}$
$\Omega(p) = \{\mathcal{B}_{h1}, \mathcal{B}_{h2}, \mathcal{B}_{h3}\}$

Figure 8.2: Several functions may influence a property.

which may either be true (denoted **T**) or false (denoted **F**). There is one bit vector for the entry and one for the exit of each node. The lattice elements $\top$ and $\bot$ are "all bits true" (denoted $\overline{\mathrm{T}}$) and "all bits false" (denoted $\overline{\mathrm{F}}$) , or vice-versa, depending on $\sqcap$. When $\sqcap$ is Π, $\top$ is $\overline{\mathrm{T}}$ while $\bot$ is $\overline{\mathrm{F}}$; the situation is exactly opposite in when $\sqcap$ is Σ. TOP is the value of an individual property in a bit vector representing the $\top$ element of the lattice, while BOT is the value of an individual property in a bit vector representing the $\bot$ element of the lattice.

The data flow properties associated with program points $in(i)$ and $out(i)$ are denoted by IN_i and OUT_i respectively. The program point for a property p is denoted by $\mathcal{P}p(p)$. Two properties belonging to different program points are called *corresponding* properties if they represent information about the same entity, *viz.* the same variable or the same expression. By definition, the relation of correspondence is reflexive, i.e. a property corresponds to itself. A property p associated with program point u is a *neighbour* of a corresponding property p' associated with a program point v, if u is a neighbour of v.

8.3 Flow Functions

Let a property p be influenced by the value of a neighbouring property p' through a flow function h. This fact is denoted by $p \leftarrow h(p')$. Clearly, $p \in \mathcal{N}(p')$ and $p' \in \mathcal{N}^{-1}(p)$. Whenever the program points of the two properties are required along with function, we write h as $h_{(u',u)}$ where $\mathcal{P}p(p)$ is u and $\mathcal{P}p(p')$ is u'. The notation describing neighbourhood is also extended to the corresponding program points u and u'. Analogous remarks hold for the *bit functions* of h, denoted by $\mathcal{B}_h$. As shown in Figure 8.2, a property p may have several functions influencing it. We define $\Omega(p)$ as the set of functions influencing the property p. Thus,

$\mathcal{B}_h$	$\mathcal{B}_h(\text{TOP})$	$\mathcal{B}_h(\text{BOT})$	Behaviour
raise	TOP	TOP	Result is always TOP.
lower	BOT	BOT	Result is always BOT.
propagate	TOP	BOT	Result is same as the function argument.
negate	BOT	TOP	Negates the function argument.

Table 8.1: Four possibilities for a bit function

$\Omega(p) = \{\mathcal{B}_h \mid p \leftarrow \mathcal{B}_h(p')\}$

A bit function $\mathcal{B}_h(u, u')$ has several attributes :

1. *Direction* : A bit function may be a forward flow function or a backward flow function.

2. *Association* : A bit function may be associated with an edge or a node. This attribute is required due to the presence of self loops to distinguish between the flow between the entry/exit of a node through the node and along a self loop, if any.

3. *Type* : Since a bit may have two values (i.e. TOP and BOT) there may be at most four different possibilities for a bit function [25] (Table 8.1) out of which the *negate* function is forbidden in monotone data flow frameworks.

In our treatment of incremental data flow analysis, we will be referring to a bit function and its type interchangeably. Thus, while we may write $\Omega(p) = \{\mathcal{B}_{h1}, \mathcal{B}_{h2}, \mathcal{B}_{h3}\}$ in Figure 8.2, we may also write $\Omega(p) = \{propagate, propagate, propagate\}$ if the three bit functions are propagate.

8.4 The Flow of Information

Information flows from a program point u to a program point v when a property at u, on becoming BOT, causes the corresponding property at v to become BOT.

Definition 8.1 : Information flow path
An information flow path (ifp) is a sequence of program points along which information can flow during data flow analysis.

It is easy to see from Table 8.1 that *propagate* is the only function for which the function value depends on the argument. Thus we say that *p depends* on the argument property for its value. This notion of dependence forms a fundamental concept in the generalised theory and will be used extensively in this work since incremental data flow analysis involves computing the influence of a change on the rest of the graph. This is achieved by finding out the properties which depend on the changed property.

Definition 8.2 : **Dependence**
A property p depends on a property p' if p' is capable of changing p to BOT. The dependence relation of p is denoted by $\mathcal{D}(p)$ where,

$\mathcal{D}(p) = \{p' \mid p \leftarrow \mathcal{B}_h(p'),\ \mathcal{B}_h \equiv$ *propagate* and *lower* $\notin \Omega(\text{p})\}$

If the condition *lower* $\notin \Omega(p)$ is violated, p will always be BOT and hence would not depend on any property. In such a case $\mathcal{D}(p) = \emptyset$.

Definition 8.3 : **Chain**[2]
A chain ch from a property p to property p' is a sequence $p_0 \xrightarrow{\mathcal{T}_1} p_1 \xrightarrow{\mathcal{T}_2} p_2 \cdots \xrightarrow{\mathcal{T}_n} p_n$ such that $p_0 \equiv p$, $p_n \equiv p'$, and $p_{i-1} \in \mathcal{D}(p_i)$, $0 < i \leq n$. $\mathcal{T}_i$ indicates the association of function h with either an edge or a node for $p_i \leftarrow h(p_{i-1})$.

In terms of program points, $p_i \leftarrow h_{(u,v)}(p_{i-1})$ and $h_{(u,v)} \equiv$ *propagate* where $u = \mathcal{P}p(p_{i-1})$ and $v = \mathcal{P}p(p_i)$. Further, $\mathcal{T}_i$ is f if h is a node flow function, g if h is an edge flow function. Note that $p_{i-1} \in \mathcal{D}(p_i)$ implies that *lower* $\notin \mathcal{D}(p_i)$.

Since the set $\mathcal{D}(p)$ for a p may contain multiple elements, multiple chains may exist between two properties. Hence unlike part I, CH(p, p') denotes a set of chains rather than one chain.

Observation 8.1 : *If $\exists p$ such that $CH(p, p') \neq \emptyset$ then $p = BOT \Rightarrow p' = BOT$.* □

Observation 8.2 : *If $\exists p$ such that $CH(p, p') \neq \emptyset$ then $p = TOP \not\Rightarrow p' = TOP$.* □

Though the chains are acyclic, it is quite likely that p_0 may depend on p_n. A chain gives rise to a *cyclic dependence* if $p_n \in \mathcal{D}(p_0)$.

Observation 8.3 : *All properties in a chain which gives rise to a cyclic dependence, must have the same value in a fixed point solution.* □

Observation 8.3 must be contrasted with observations 8.1 and 8.2. If any p_i in a chain is BOT, all properties p_j, $i \leq j \leq n$ must be BOT in any fixed point solution since all these properties depend on p_i. If the chain gives rise to a cyclic dependence, all other properties p_k, $0 \leq k < i$ must also be BOT. If the chain does not give rise to a cyclic dependence, all properties p_k, $0 \leq k < j$ must be TOP where $j \leq i$ and p_j is the first BOT property. It follows that no property can be TOP in a chain giving rise to a cyclic dependence unless all other properties are TOP.

[2]Not to be confused with the lattice chains.

8.5 Performing Exhaustive Data Flow Analysis

Recall that the *Initial Trigger Set*, denoted TR_0, contains the properties whose initial values are BOT due to computations located within a node/along an edge. TR_0 can also be defined as :

$$\begin{aligned} Bprops &= \{p \mid \text{ either } \mathcal{P}p(p) = in(i) \text{ and } i \in entry(G), \text{ or} \\ &\quad\ \mathcal{P}p(p) = out(i), i \in exit(G)\} \\ TR_0 &= \{p \mid \text{ either } lower \in \Omega(p), \text{ or} \\ &\quad\ p \in Bprops \text{ and } p = \text{BOT}\} \end{aligned} \tag{8.1}$$

Observation 8.4 : $p \in TR_0 - Bprops \Rightarrow \mathcal{D}(p) = \emptyset$. □

The values of properties can only change from TOP to BOT and not the other way round. Thus the properties whose initial values are BOT, do not change any further but instead tend to influence other corresponding properties. Hence exhaustive data flow analysis is performed by

(i) constructing TR_0, and

(ii) constructing chains for the properties in TR_0.

The MFP solution of a data flow problem can thus be defined as follows :

$$\forall p,\ p = \text{BOT } \textit{iff } p \in TR_0 \text{ or } \exists p' \in TR_0 \text{ such that } \ \text{CH}(p', p) \neq \emptyset \tag{8.2}$$

This definition will be used later to show the correctness of incremental data flow analysis. In particular, it will be shown later that the old MFP solution, when updated incrementally to reflect the changes, satisfies the above condition.

8.6 Refinement of the Notions from the Generalised Theory

The discussion in this section can perhaps be appreciated better in retrospect. We nevertheless present it at the outset to avoid any confusion.

There is a subtle difference between exhaustive and incremental analysis : The discussion of incremental analysis tries to captures specificities of an instance of a data flow framework which are ignored by the discussion of exhaustive analysis. Thus the notions in incremental analysis are more refined than the notions in exhaustive analysis.

It is the above subtle difference that warrants the following alterations :

1. Unlike the generalised theory,the entire discussion of incremental analysis is in terms of program points rather than nodes/edges. When the discussion is in terms of the notion of program points, the entry and exit of a node are represented by distinct program points so the distinction between the entry and exit points of a node, comes about inherently.

2. In the exhaustive analysis, the flow functions are in terms of nodes and edges (distinct sets $\mathcal{G}$ and $\mathcal{F}$). This evolved in contrast to the traditional notion of *transfer functions* which may be associated with the edges or the nodes.

   In incremental analysis since the attribute of the *association* of a function (with either a node or an edge) is introduced, it does not matter much if the function is in $\mathcal{G}$ or in $\mathcal{F}$ so long as the attribute association is correctly defined.

3. The discussion of exhaustive analysis is in terms of information flow paths which can be determined from the data flow equations. In incremental analysis we talk in terms of chains which capture the information flow more precisely since they are governed by the individual functions (i.e. if the functions are *propagate*, *raise*, or *lower*). In exhaustive analysis, we distinguish between only existing and non-existing functions (this information is derived from data flow equations) without looking at the constants. Hence, the discussion of exhaustive analysis involves only the *propagate* and non-*propagate* functions whereas the discussion of incremental analysis divides the class of non-*propagate* functions further into *raise* and *lower* functions.

   This finer distinction is denoted by the attribute *type* of a function. Note that in exhaustive analysis, the attribute *typename* denotes what is called the *association* of a function in incremental data flow analysis.

4. The notion of *dependence* has been defined in the context of the state transition model for exhaustive data flow analysis. Since incremental data flow analysis distinguishes between *lower* and *raise* functions, the definition of dependence changes slightly and the condition that the property should not be influenced by a *lower* function, is added.[3]

5. The notion of *chain* is defined in terms of the notion of the *seed* of a property in exhaustive data flow analysis. Since, there exists a unique seed of a property during any analysis, there exists a unique chain from a property p to a property p' in exhaustive analysis. In incremental data flow analysis, a chain is defined in terms of the notion

[3] Besides, a more compact notation is used for brevity since the discussion of incremental data flow analysis involves complicated rigourous statements.

of *dependence*. Since a property depends on multiple properties, multiple chains may exist from a property p to a property p' in incremental data flow analysis. Hence the set $\text{CH}(p, p')$ denotes a unique chain in exhaustive analysis but a set of chains in incremental analysis. Due to the multiplicity of chains, it becomes imperative to distinguish between chains and hence the attribute of the *association* of a function with either an edge or a node is included.[4]

We chose the option of not changing the notations and concepts in the generalised theory since it already is a part of the published literature [23, 38].

[4]In exhaustive analysis, this distinction is required in the case of *information flow paths*.

Chapter 9

A Functional Model for Incremental Data Flow Analysis

So I tossed off this drawing. And I threw out an explanation with it.
"This is only his box. The sheep you asked for is inside."
I was very surprised to see a light break over the face of my young judge :
"That is exactly the way I wanted it !

The fundamental concepts of the generalised theory have been presented with a lot of intuitive explanations in Part I of the thesis. This chapter presents the material much more formally. The rigour used in this chapter is not redundant — it provides a formalism to define the notion of incremental solution concretely. More detailed insights into the process of incremental data flow analysis are contained in chapter 12 which proves correctness of the model.

Equation pair 7.1 defines a high level abstraction of the proposed paradigm; we use a more specific notation in the context of incremental data flow analysis. Section 9.2.1 defines the notation using which we differentiate between the entities of the old instance of a data flow framework (for which the MFP solution is available) and the new instance (for which the solution needs to be computed incrementally). The model is presented in an evolutionary style; additional notation is introduced as and when required.

Section 9.1 develops the fundamental concepts of incremental data flow analysis intuitively. This section also provides two examples of incremental data flow analysis which are later used to illustrate the application of the proposed functional model. Section 9.2.2 defines a formalism for classifying the changes that may take place in a data flow problem and discusses their influence on TR_0 and chains. Section 9.2.3 defines the change in the old solution and shows how the old solution may be updated to reflect the changes in the previous instance of a data flow problem. Section 9.2.4 summarises the model. Section 9.3 computes the incremental solutions of the examples in section 9.1 using the proposed functional model. Section 9.4 shows how the proposed model handles the cyclic dependences and suggests extensions to handle bidirectional data flows, multiple function changes, and structural changes.

9.1 Motivation

This section contains an intuitive discussion of incremental data flow analysis. Concepts evolved in this section are used in section 9.2.3 to formally define the concepts used in the proposed functional model for incremental data flow analysis. For the simplicity of exposition, available expressions analysis is used for motivating the concepts.

Let IN_i^l and OUT_i^l be the properties for the expression e_l at the program points $in(i)$ and $out(i)$ respectively. Assuming that the interprocedural information is not available, the data flow equations for available expressions analysis are :

$$\begin{aligned}
\text{IN}_i^l &= \begin{cases} \mathbf{F} & \text{if } i \in entry(G) \\ \prod\limits_{p \in pred(i)} \text{OUT}_p^l & \text{otherwise} \end{cases} \\
\text{OUT}_i^l &= \text{Comp}_i^l + \text{Transp}_i^l \cdot \text{IN}_i^l
\end{aligned}$$

The flow functions involved in available expressions analysis are of the following form :

– Node flow functions $f \in \mathcal{F} : f^f(X) \;=\; \text{Comp} + \text{Transp} \cdot X,\;\; f^b(X) \;=\; \top$

– Edge flow functions $g \in \mathcal{G} : g^f(X) \;=\; X,\;\; g^b(X) \;=\; \top$

Since $\sqcap$ is Π, TOP is **T** and BOT is **F**.

As a consequence of some change in a node,

(i) some properties may change from **T** to **F**, i.e. from TOP to BOT,

(ii) some properties may change from **F** to **T**, i.e. from BOT to TOP, and

(iii) some properties may remain the same.

These changes may take place *locally* (i.e., the properties undergoing a change may be the properties associated with the node in which the original change has taken place), or *globally* (i.e., the properties may be associated with some other node). Global changes can be found by incorporating the effect of local changes over the rest of the graph. It is easy to see that

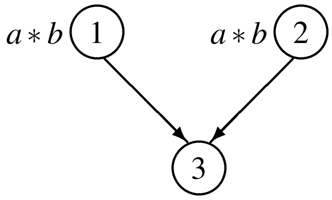


Figure 9.1: Incorporating TOP to BOT and BOT to TOP changes

the work involved in incremental data flow analysis would mostly be centered around finding global changes. The rest of this section discusses global changes only.

The notions discussed in the following two subsections are essentially the same as the *Any Path Effect* and the *All Paths Effect*, respectively, defined in [49]. However, their categorisation is based on an adhoc reasoning of a change and its implication. Thus in some cases, a change which belongs to a particular category in our approach, may belong to a different category in their approach. We, on the other hand use these notions consistently and develop a model which handles all the cases uniformly.

9.1.1 Incorporating TOP to BOT changes

Consider figure 9.1 which represents (a part of) a program flow graph. In this case OUT_1^1, OUT_2^1, IN_3^1, and OUT_3^1 are TOP (i.e. **T**). Let an assignment to variable "*a*" be inserted after the computation of expression "$a*b$" in node 1. After this change, OUT_1^1 becomes BOT (i.e. **F**) while OUT_2^1 remains TOP. IN_3^1 also becomes BOT since $\text{IN}_3^1 = \text{OUT}_1^1 \sqcap \text{OUT}_2^1$. The important point to note is that the value of IN_3^1 is determined by the value of OUT_1^1 alone in such a case.[1] Thus, the effect of TOP to BOT changes can be incorporated directly by using the notion of *propagation* as defined in section 4.2.2.

9.1.2 Incorporating BOT to TOP changes

Note that if the change in the value of OUT_1^1 is from BOT to TOP, the value of IN_3^1 cannot be determined directly since now it depends on the value of OUT_2^1 also, which is not the case for a TOP to BOT change in OUT_1^1. All that we can infer about IN_3^1 in the case of a BOT to TOP change in OUT_1^1 is that it *may* be TOP. Its actual value will be determined by the value of OUT_2^1.

[1] Refer to the *SBVP* property in section 4.1.

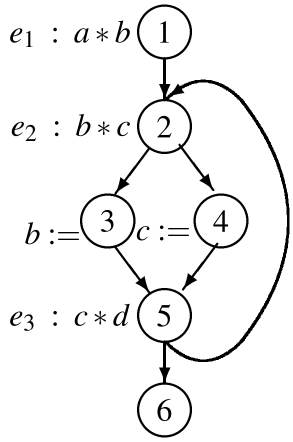


Figure 9.2: Program flow graph for incremental data flow analysis of available expressions.

Thus, incorporating the effect of BOT to TOP changes requires more processing which can be summarised in the following two steps.

1. Identify the properties which *may* become TOP.

2. From the properties identified in the above step, find out the properties which must remain BOT due to the effect of some other property.

9.1.3 Examples of Incremental Data Flow Analysis

We consider two examples of incremental data flow analysis where the new solutions are computed intuitively. The same solutions are then computed formally using the proposed model in section 9.3.

We use the program flow graph in Figure 9.2. For convenience, we define the universal set of properties :

$$U = \{\text{IN}_i^l \mid 1 \leq l \leq 3,\ 1 \leq i \leq 6\} \cup \{\text{OUT}_i^l \mid 1 \leq l \leq 3,\ 1 \leq i \leq 6\}$$

The MFP solution for this instance of available expressions analysis is contained in Table 9.1.

Example 9.1 : Assume that the assignment to c in node 4 is deleted. Since expressions e_2 and e_3 become available at the exit of node 4, some properties which are **F** may now become **T**. We calculate the new solution in two steps :

Node	Expression e_l					
	e_1		e_2		e_3	
	IN_i^l	OUT_i^l	IN_i^l	OUT_i^l	IN_i^l	OUT_i^l
1	**F**	**T**	**F**	**F**	**F**	**F**
2	**F**	**F**	**F**	**T**	**F**	**F**
3	**F**	**F**	**T**	**F**	**F**	**F**
4	**F**	**F**	**T**	**F**	**F**	**F**
5	**F**	**F**	**F**	**F**	**F**	**T**
6	**F**	**F**	**F**	**F**	**T**	**T**

Table 9.1: MFP solution for program flow graph in Figure 9.2

1. The properties which were **F** and *may* become **T** due to this change are :

   $\{\mathrm{OUT}_4^2, \mathrm{OUT}_4^3, \mathrm{IN}_5^2, \mathrm{IN}_5^3, \mathrm{OUT}_5^2, \mathrm{IN}_2^2, \mathrm{IN}_6^2, \mathrm{OUT}_6^2\}$

2. From the above properties, the properties which must remain **F** are :

   (a) Properties for expression e_2 : IN_5^2 cannot become **T** since OUT_3^2 is **F**.
   Hence $\mathrm{IN}_5^2, \mathrm{OUT}_5^2, \mathrm{IN}_6^2, \mathrm{OUT}_6^2, \mathrm{IN}_2^2$ must remain **F**.

   (b) Properties for expression e_3 : OUT_4^3 cannot become **T** since IN_4^3 is **F**. Thus, IN_5^3 must also be **F**.

Thus, in the final solution, only OUT_4^2 becomes **T**. No other property changes. □

Example 9.2 : Consider another change in the program of previous example. Let the expression e_3 in node 5 be deleted. The new solution computed in the previous example now becomes old solution for this example and we need to find out the changes in that solution due to deletion of expression e_3. Since the expression ceases to be available at the exit of node 5, some properties which were **T**, will now become **F**. In particular, the following properties will become **F** : $\{\mathrm{OUT}_5^3, \mathrm{IN}_6^3, \mathrm{OUT}_6^3\}$. □

Thus, the final solution after incorporating the effect of the above two changes is contained in Table 9.2.

Node	Expression e_l					
	e_1		e_2		e_3	
	IN	OUT	IN	OUT	IN	OUT
1	**F**	**T**	**F**	**F**	**F**	**F**
2	**F**	**F**	**F**	**T**	**F**	**F**
3	**F**	**F**	**T**	**F**	**F**	**F**
4	**F**	**F**	**T**	**T**	**F**	**F**
5	**F**	**F**	**F**	**F**	**F**	**F**
6	**F**	**F**	**F**	**F**	**F**	**F**

Table 9.2: New MFP solution for program flow graph in Figure 9.2

9.2 A Functional Model for Incremental Data Flow Analysis

9.2.1 Preliminaries

Let $\pi(x)$ and $\sigma(x)$ denote a partition of the set x and a subset in the partition $\pi(x)$, respectively. We use a subscript to denote a specific partition and a superscript to denote a specific subset within a partition. Thus, a partition π_μ consisting of k subsets is :

$$\pi_\mu(x) \quad = \quad \{\sigma^1_\mu(x), \sigma^2_\mu(x), \ldots, \sigma^k_\mu(x)\}$$

Let $\mathbf{I} = < G, M >$ represent the instance for which incremental data flow analysis is to be performed.[2] Let x be an entity in instance $\mathbf{I}$. $old\,(x)$ represents the corresponding entity in the old instance which itself it denoted by $old\,(\mathbf{I})$. In particular, $old\,(x)$ denotes :

- old chain x if x is a chain,
- old value of x if x is a property,
- old function x if x is a function,
- old solution x if x is a solution,
- old set x if x is a set.

[2]Recall that incremental data flow analysis can be performed only if the solution of the previous analysis is available. Thus, whenever we talk about performing incremental data flow analysis for an instance of a data flow framework, we will be referring to both the old and the current instance.

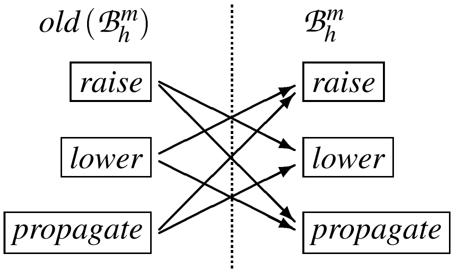


Figure 9.3: Change in the function $\mathcal{B}_h^m$, for $p \leftarrow \mathcal{B}_h^m(p')$.

Thus, $old\,(\mathbf{I}) =< old\,(G), old\,(M) >$ represents the old instance of $\mathbf{D}$. We assume that $old\,(G) \equiv G$, i.e. we restrict ourselves to the changes in M (alternatively, in $\mathcal{F}$ and $\mathcal{G}$), only. Extensions to structural changes (i.e. changes in G) will be discussed later. We also presume that there is no change in $\mathcal{L}$, i.e. no bit is added to, or deleted from, the bit vector representing the data flow properties.

Let the MFP solution for the instance $\mathbf{I}$ be S. Then, $old\,(S)$ represents the MFP solution for $old\,(\mathbf{I})$. The goal of incremental data flow analysis is to compute S from $old\,(S)$.

9.2.2 Change in Data Flow Problem

9.2.2.1 Change in Flow Functions

Consider a function $h^m \in \mathcal{F} \cup \mathcal{G}$ which is modified. Let $\mathcal{B}_h^m$ be a modified bit function of h^m, i.e. $\exists\, x \in \{\text{TOP}, \text{BOT}\}$ such that $[old\,(\mathcal{B}_h^m)]\,(x) \neq \mathcal{B}_h^m(x)$. $\partial\mathcal{B}_h^m$ denotes the change $old\,(\mathcal{B}_h^m) \twoheadrightarrow \mathcal{B}_h^m$. Figure 9.3 summarises the six possible changes. We abbreviate the functions *raise*, *lower*, and *propagate* by r, l, and p[3], respectively, to define Δ, the set containing these changes.

$$\Delta \;=\; \{r \twoheadrightarrow l\ ,\ r \twoheadrightarrow p\ ,\ l \twoheadrightarrow r\ ,\ l \twoheadrightarrow p\ ,\ p \twoheadrightarrow r\ ,\ p \twoheadrightarrow l\ \}$$

Δ forms the base set for the discussion of incremental data flow analysis. In particular, we define three partitions of Δ which will be used later[4] :

$$\pi_\alpha(\Delta) \;=\; \{\sigma_\alpha^1(\Delta), \sigma_\alpha^2(\Delta), \sigma_\alpha^3(\Delta)\}$$

[3]Use of p as an abbreviation for *propagate* is an unfortunate choice since it overloads p which also denotes a property. However, the context is sufficient to remove the ambiguity : p denotes the bit function *propagate* only when it appears in conjunction with the arrow "$\twoheadrightarrow$" which denotes a change. In all other cases, p denotes a property.

[4]Though the order of elements in a set does not matter, we define an ordering on the subsets in a partition to allow for positional correspondence.

$$
\begin{aligned}
&= \{\{r \rightarrowtriangle l\,,\, p \rightarrowtriangle l\}, \{l \rightarrowtriangle r\,,\, l \rightarrowtriangle p\}, \{p \rightarrowtriangle r\,,\, r \rightarrowtriangle p\}\} && (9.1)\\
\pi_\beta(\Delta) &= \{\sigma^1_\beta(\Delta), \sigma^2_\beta(\Delta)\}\\
&= \{\{r \rightarrowtriangle l\,,\, r \rightarrowtriangle p\,,\, p \rightarrowtriangle l\}, \{l \rightarrowtriangle r\,,\, l \rightarrowtriangle p\,,\, p \rightarrowtriangle r\}\} && (9.2)\\
\pi_\gamma(\Delta) &= \{\sigma^1_\gamma(\Delta), \sigma^2_\gamma(\Delta), \sigma^3_\gamma(\Delta)\}\\
&= \{\{p \rightarrowtriangle l\,,\, p \rightarrowtriangle r\}, \{l \rightarrowtriangle p\,,\, r \rightarrowtriangle p\}, \{l \rightarrowtriangle r\,,\, r \rightarrowtriangle l\}\} && (9.3)
\end{aligned}
$$

(i) $\pi_\alpha(\Delta)$ uses the *lower* function to partition Δ. $\sigma^1_\alpha(\Delta)$ contains the changes in which the new function is *lower*, $\sigma^2_\alpha(\Delta)$ contains the changes in which the old function is *lower*, and $\sigma^3_\alpha(\Delta)$ contains the changes in which *lower* function is not involved.

(ii) $\pi_\beta(\Delta)$ uses the possible value of the bit function $\mathcal{B}^m_h$ to partition Δ. $\sigma^1_\beta(\Delta)$ contains the changes which may result in the new function computing BOT while the old function may have computed TOP. Similarly, $\sigma^2_\beta(\Delta)$ contains the changes which may result in the new function computing TOP while the old function would have computed BOT.

(iii) $\pi_\gamma(\Delta)$ uses the *propagate* function to partition Δ. $\sigma^1_\gamma(\Delta)$ contains the changes in which old function is *propagate*, $\sigma^2_\gamma(\Delta)$ contains the changes in which the new function is *propagate*, and $\sigma^3_\gamma(\Delta)$ contains the changes in which *propagate* function is not involved.

Observation 9.1 :
$$
\begin{aligned}
\sigma^1_\beta(\Delta) &= \sigma^1_\alpha(\Delta) \cup \{r \rightarrowtriangle p\}\\
\sigma^2_\beta(\Delta) &= \sigma^2_\alpha(\Delta) \cup \{p \rightarrowtriangle r\}
\end{aligned}
$$

□

9.2.2.2 Influence of Function Change

We know from section 8.5 that the MFP solution is defined in terms of the chains originating from the properties which are in TR_0. Thus we need to define TR_0 and chains for **I**.

Influence of Function Change on Initial Trigger Set

In order to find the change in TR_0 due to a change in h^m, we define

$$
\begin{aligned}
\mathcal{TR}^+ &= \{p \mid p \leftarrow \mathcal{B}^m_h(p') \text{ s.t. } \mathcal{B}^m_h \equiv \mathit{lower}\} && (9.4)\\
\mathcal{TR}^- &= \{p \mid p \leftarrow \mathcal{B}^m_h(p') \text{ s.t. } \mathit{old}\,(\mathcal{B}^m_h) \equiv \mathit{lower} \text{ and } \mathit{lower} \notin \Omega(p)\} && (9.5)
\end{aligned}
$$

TR_0 corresponding to instance **I** is defined as follows :

$$TR_0 = \mathit{old}\,(TR_0) \cup \mathcal{TR}^+ - \mathcal{TR}^- \qquad (9.6)$$

where the operations are performed left to right.

Observation 9.2 : $\mathcal{TR}^+ \cap \mathcal{TR}^- = \emptyset$. □

Observation 9.3 :
$$\begin{aligned} \mathcal{TR}^+ &= \{p \mid p \leftarrow \mathcal{B}_h^m(p') \textit{ s.t. } \partial\mathcal{B}_h^m \in \sigma_\alpha^1(\Delta)\} \\ \mathcal{TR}^- &= \{p \mid p \leftarrow \mathcal{B}_h^m(p') \textit{ s.t. } \partial\mathcal{B}_h^m \in \sigma_\alpha^2(\Delta) \text{and } \textit{lower} \notin \Omega(\text{p})\} \end{aligned}$$
□

Observation 9.4 : $\forall p,\ p \leftarrow \mathcal{B}_h^m(p'),\ \partial\mathcal{B}_h^m \in \sigma_\alpha^3(\Delta) \Rightarrow p \in TR_0$ *iff* $p \in old\,(TR_0)$. □

Influence of Function Change on Chains

As a consequence of a function change, some chains may become non-existent while some new chains may come into existence. In order to capture this influence formally, we define the following notions.[5]

- A chain *ch involves* a tuple $< p_1, p_2, \mathcal{B}_h >$ if :

  (i) $p_2 \leftarrow \mathcal{B}_h(p_1)$ and $\mathcal{B}_h \equiv$ *propagate* and *lower* $\notin \Omega(p_2)$.

  (ii) $p_1 \xrightarrow{\mathcal{T}} p_2$ appears in *ch* where $\mathcal{T}$ indicates the association of $\mathcal{B}_h$.

- A chain *ch includes* a property p if *ch* involves a tuple $< p', p, \mathcal{B}_h >$. Thus, by definition, the first property is not included in a chain.

- A chain *ch* existing in *old* ($\mathbf{I}$) *becomes non-existent* in $\mathbf{I}$ if function $\mathcal{B}_h$ of a tuple $< p_1, p_2, \mathcal{B}_h >$ involved in *ch* becomes either *lower* or *raise* in $\mathbf{I}$.

- Consider a tuple $< p_1, p_2, \mathcal{B}_h >$ such that $p_1 \in \mathcal{N}^{-1}(p_2)$ and $p_2 \leftarrow \mathcal{B}_h(p_1)$. A chain $p_1 \xrightarrow{\mathcal{T}} p_2$ is *created* in $\mathbf{I}$ if *old* ($\mathcal{B}_h$) $\not\equiv$ *propagate*, $\mathcal{B}_h \equiv$ *propagate*, and *lower* $\notin \Omega(p_2)$.

  More precisely, one chain involving the tuple $< p_1, p_2, \mathcal{B}_h >$ will be created and added to every set CH(p_1', p_2') where p_1' and p_2' are defined thus,

  – if p_1 is the last property of a chain ch_1 which exists in $\mathbf{I}$, then p_1' is any property in ch_1 else p_1' is p_1.

  – if p_2 is the first property of a chain ch_2 which exists in $\mathbf{I}$, then p_2' is any property in ch_2 else p_2' is p_2.

[5]This part of this sections is required mostly for the correctness proofs. However, it is included here for sake of completeness of the definition of the model.

- Finally, the following chains exist in **I** :
  - The chains which exist in *old* (**I**) and do not become non-existent in **I**.
  - The chains which are created in **I**.

Let $p \leftarrow \mathcal{B}_h^m(p')$. It is clear that only the chains which include the property p may change due to the change $\partial\mathcal{B}_h^m$. All other chains remain unaffected. Also, if $lower \in \Omega(p) - \{\mathcal{B}_h^m\}$, no chain is affected. Thus, only if $lower \notin \Omega(p) - \{\mathcal{B}_h^m\}$, the following chains for the properties corresponding to p may change :

1. chains involving $< p', p, \mathcal{B}_h^m >$.
2. chains involving $< p'', p, \mathcal{B}_h >$ for $p'' \in \mathcal{N}^{-1}(p), p'' \not\equiv p'$ (i.e. $\mathcal{B}_h \in \Omega(p) - \{\mathcal{B}_h^m\}$)if
   (a) $\mathcal{B}_h \equiv old\,(\mathcal{B}_h) \equiv propagate$, and
   (b) $\partial\mathcal{B}_h^m \in \sigma_\alpha^1(\Delta) \cup \sigma_\alpha^2(\Delta)$.

We consider the two classes separately and enumerate each possible case and its influence on the chains in the class :

1. Chains involving $< p', p, \mathcal{B}_h^m >$: We use the partition $\pi_\gamma(\Delta)$ to enumerate the distinct cases.
   (a) $\partial\mathcal{B}_h^m \in \sigma_\gamma^1(\Delta)$.
   In these cases, *old* $(\mathcal{B}_h^m)$ is *propagate*. Thus, these chains existed in *old* (**I**) but become non-existent in **I**.
   (b) $\partial\mathcal{B}_h^m \in \sigma_\gamma^2(\Delta)$.
   In these cases $\mathcal{B}_h^m$ is *propagate*. Since *old* $(\mathcal{B}_h^m)$ is not *propagate*, no such chains existed in *old* (**I**). They are created in **I**.
   (c) $\partial\mathcal{B}_h^m \in \sigma_\gamma^3(\Delta)$.
   In these cases, *propagate* function is not involved. Thus such chains neither existed nor are created.
2. Chains involving $< p'', p, \mathcal{B}_h >$: If there exists a *lower* function in $\Omega(p)$i, then p cannot be included in a chain. Since $\mathcal{B}_h^m$ is the only function in $\Omega(p)$ which changes, these chains may be influenced only if either $\mathcal{B}_h^m$ or *old* $(\mathcal{B}_h^m)$ is *lower*.[6] Hence we use the partition $\pi_\alpha(\Delta)$ to enumerate the distinct cases.

[6]Note that we are considering those chains which do not involve $< p', p, \mathcal{B}_h^m >$, but $< p'', p, \mathcal{B}_h >$.

(a) $\partial\mathcal{B}_h^m \in \sigma_\alpha^1(\Delta)$.

Since $\mathcal{B}_h^m$ is *lower*, $p \in TR_0$. Hence chains involving $< p'', p, \mathcal{B}_h >$ cannot exist in **I**. However, since $\mathcal{B}_h$ is *propagate* and $lower \notin old\,(\Omega(p))$, such chains existed in $old\,(\mathbf{I})$.

(b) $\partial\mathcal{B}_h^m \in \sigma_\alpha^2(\Delta)$.

Since $old\,(\mathcal{B}_h^m)$ is *lower*, $p \in old\,(TR_0)$. Hence chains involving $< p'', p, \mathcal{B}_h >$ did not exist in $old\,(\mathbf{I})$. However, since $\mathcal{B}_h$ is *propagate* and $lower \notin \Omega(p)$ in **I**, such chains are created in **I**.

(c) $\partial\mathcal{B}_h^m \in \sigma_\alpha^3(\Delta)$.

Since *lower* function is not involved, the chains are not influenced in any way.

9.2.3 Updating the Old Solution

9.2.3.1 Local change

The *local change* caused by changes in h^m is denoted by $\mathcal{LC} = < \mathcal{B2T}, \mathcal{T2B} >$. Sets $\mathcal{B2T}$ and $\mathcal{T2B}$ contain the BOT-to-TOP and TOP-to-BOT changes respectively.

$$\begin{aligned} \mathcal{B2T} &= \{p \mid p \leftarrow \mathcal{B}_h^m(p'),\ \mathcal{B}_h^m(\text{TOP}) = \text{TOP}, [old\,(\mathcal{B}_h^m)]\,(old\,(p')) = \text{BOT}, \\ &\quad \text{and } lower \notin (\Omega(p) - \{\mathcal{B}_h^m\})\} \qquad (9.7) \\ \mathcal{T2B} &= \{p \mid p \leftarrow \mathcal{B}_h^m(p'),\ \mathcal{B}_h^m(\text{BOT}) = \text{BOT},\ \text{and } old\,(p) = \text{TOP and} \\ &\quad old\,(p') = \text{BOT if } \mathcal{B}_h^m \equiv propagate\} \qquad (9.8) \end{aligned}$$

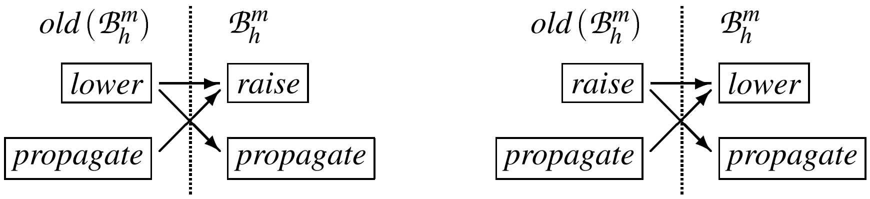
Note the asymmetry in the definition of $\mathcal{B2T}$ and $\mathcal{T2B}$. In the case of $\mathcal{T2B}$, it is enough to know if the old value of p is TOP. However, in the case of $\mathcal{B2T}$, p could have been BOT due to any other function; since $\mathcal{B}_h^m$ is the only function that is changing, p cannot be included in $\mathcal{B2T}$ unless it was BOT in $old\,(\mathbf{I})$ due to the influence of $old\,(\mathcal{B}_h^m)$.

Observation 9.5 : $\mathcal{B2T} \cap \mathcal{T2B} = \emptyset$. □

Observation 9.6 :

$$\begin{aligned} \mathcal{B2T} &= \{p \mid p \leftarrow \mathcal{B}_h^m(p'),\ \partial\mathcal{B}_h^m \in \sigma_\beta^2(\Delta), [old\,(\mathcal{B}_h^m)]\,(old\,(p')) = \text{BOT},\ \text{and} \\ &\quad lower \notin \Omega(\text{p})\} \\ \mathcal{T2B} &= \{p \mid p \leftarrow \mathcal{B}_h^m(p'),\ \partial\mathcal{B}_h^m \in \sigma_\beta^1(\Delta),\ old\,(p) = \text{TOP},\ \text{and} \\ &\quad old\,(p') = \text{BOT if } \mathcal{B}_h^m \equiv propagate\} \end{aligned}$$

□



$(a)\ p \in \mathcal{B}2\mathcal{T}$ $(a)\ p \in \mathcal{T}2\mathcal{B}$

Figure 9.4: Change in the function $\mathcal{B}_h^m$, for $p \leftarrow \mathcal{B}_h^m(p')$.

9.2.3.2 Global change

The global change, $\mathcal{GC} = <\mathcal{B}2\mathcal{T}\star, \mathcal{T}2\mathcal{B}\star>$ is determined by incorporating the transitive influence of properties in $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ sets.

- The influence of $\mathcal{T}2\mathcal{B}$ is computed by defining the set $\mathcal{GCB}(p)$ which contains the properties which depend on p, and which should change to BOT.

$$\mathcal{GCB}(p) = \begin{cases} \{p\} \cup \{p'' \mid \text{CH}(p, p'') \neq \emptyset \text{ and } old\,(p'') = \text{TOP}\} & \text{if } p \in \mathcal{T}2\mathcal{B} \\ \emptyset & \text{if } p \notin \mathcal{T}2\mathcal{B} \end{cases} \tag{9.9}$$

$$\mathcal{T}2\mathcal{B}\star = \bigcup_{p \,\in\, \mathcal{T}2\mathcal{B}} \mathcal{GCB}(p) \tag{9.10}$$

- As noted in section 9.1, the influence of $\mathcal{B}2\mathcal{T}$ cannot be computed directly since some properties which depend on the properties in $\mathcal{B}2\mathcal{T}$, may actually remain BOT due to the influence of some other properties. For a $p \in \mathcal{B}2\mathcal{T}$, let $\mathcal{GCT}(p)$ denote the set of properties which may change to TOP. The properties which do not change to TOP are contained in $\mathcal{NCT}(p)$.

$$\mathcal{GCT}(p) = \begin{cases} \{p\} \cup \{p'' \mid \text{CH}(p, p'') \neq \emptyset \text{ and } old\,(p'') = \text{BOT}\} & \text{if } p \in \mathcal{B}2\mathcal{T} \\ \emptyset & \text{if } p \notin \mathcal{B}2\mathcal{T} \end{cases} \tag{9.11}$$

$$\mathcal{NCT}(p) = \{p'' \mid p'' \in \mathcal{GCT}(p) \text{ and } \exists p''' \in TR_0 \text{ such that } \text{CH}(p''', p'') \neq \emptyset\} \tag{9.12}$$

$$\mathcal{B}2\mathcal{T}\star = \bigcup_{p \,\in\, \mathcal{B}2\mathcal{T}} (\mathcal{GCT}(p) - \mathcal{NCT}(p)) \tag{9.13}$$

9.2.3.3 Incremental Computation of S

Any solution of a bit vector data flow problem can be represented by a tuple consisting of two mutually exclusive and complementary sets : one containing the properties which are

TOP and the other containing the properties which are BOT. Let $S = <\mathcal{VT}, \mathcal{VB}>$ and $old\,(S) = <old\,(\mathcal{VT})\,, old\,(\mathcal{VB})>$.

Let $\otimes$ be a binary operation *updates*, defined on tuples consisting of two disjoint sets such that :

$$\begin{aligned}
\eta \otimes \xi &= <\eta_1, \eta_2> \otimes <\xi_1, \xi_2> \\
&= <\xi_1 - \eta_2 \cup \eta_1, \xi_2 - \eta_1 \cup \eta_2>
\end{aligned}$$

The set operations $-$ and $\cup$ are performed left to right, i.e. $\xi_1 - \eta_2 \cup \eta_1 = (\xi_1 - \eta_2) \cup \eta_1$. S is computed by updating $old\,(S)$ with $\mathcal{GC}$:

$$\begin{aligned}
S &= \mathcal{GC} \otimes old\,(S) \\
&= <\mathcal{B2T}\star, \mathcal{T2B}\star> \otimes <old\,(\mathcal{VT})\,, old\,(\mathcal{VB})> \\
&= <old\,(\mathcal{VT}) - \mathcal{T2B}\star \cup \mathcal{B2T}\star\,,\ old\,(\mathcal{VB}) - \mathcal{B2T}\star \cup \mathcal{T2B}\star> \qquad (9.14)
\end{aligned}$$

9.2.4 Summarising the Model

Figure 9.5 summaries the proposed model for incremental data flow analysis. The desired incremental solution, $\Delta\mathcal{S}$, is represented by the global change ($\mathcal{GC}$), which in turn consists of two sets : $\mathcal{B2T}\star$ and $\mathcal{T2B}\star$. These sets are defined in terms of the sets $\mathcal{GCT}(p)$ and $\mathcal{NCT}(p)$ for each property $p \in \mathcal{B2T}$ and the set $\mathcal{GCB}(p)$ for each property $p \in \mathcal{T2B}$. Computation of $\mathcal{GCT}(p)$, $\mathcal{NCT}(p)$, and $\mathcal{GCB}(p)$ requires the set TR_0, which is defined in terms of $old\,(TR_0)$, $\mathcal{TR}^-$, and $\mathcal{TR}^+$. The first level sets ($\mathcal{B2T}$, $\mathcal{T2B}$, $\mathcal{TR}^-$, $\mathcal{TR}^+$) are defined in terms of the function changes. Note that we have not shown the old sets except where the old solution is updated by $\Delta\mathcal{S}$.

We call this a functional model since all definitions are pure mathematical definitions totally devoid of any procedural element; it is easy to see that computing the incremental solution defined by this model is independent of any particular technique. Section 9.3 illustrates this through two examples of incremental data flow analysis where S is obtained by computing the sets from their fundamental definitions.

9.3 Examples Revisited

We use the examples in section 9.1.3 and compute the incremental solutions using the functional model.

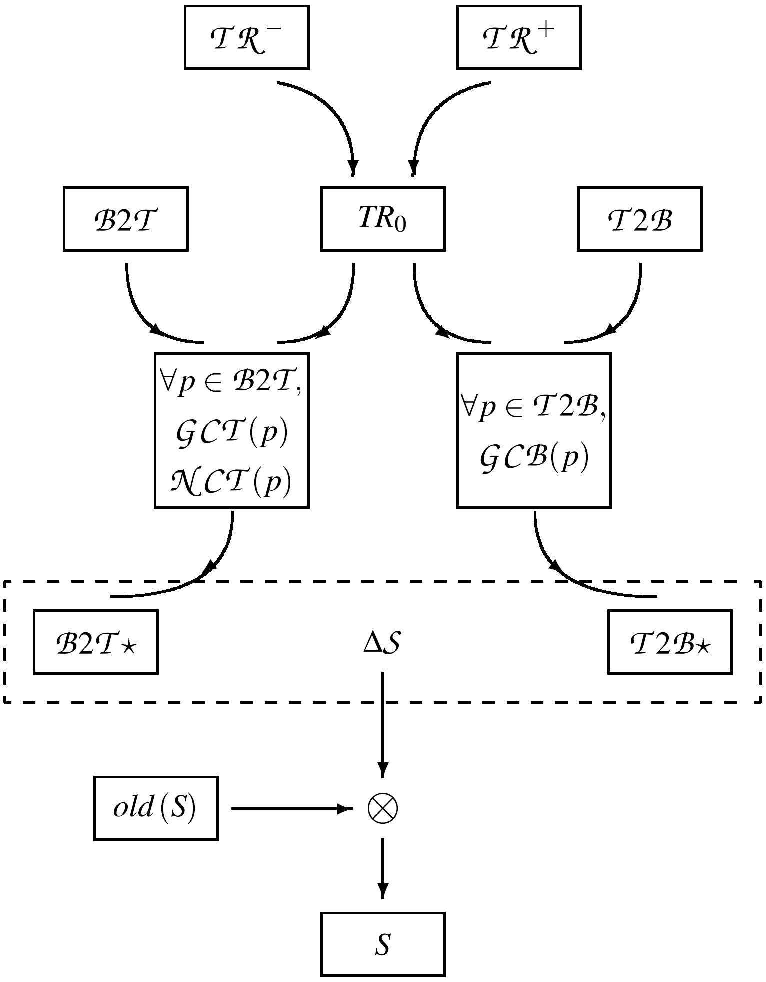


Figure 9.5: A functional model for incremental data flow analysis.

Example 9.3 : We consider example 9.1 in which the assignment to c in node 4 is deleted.

- $old(\mathbf{I}) =< old(G), old(M) >$:

  Since old(Comp) and old(Transp) are **F** for e_1 and e_2 in node 3 and for e_2 and e_3 in node 4,

$$old(TR_0) = \{\text{IN}_1^1, \text{IN}_1^2, \text{IN}_1^3, \text{OUT}_3^1, \text{OUT}_3^2, \text{OUT}_4^2, \text{OUT}_4^3\}$$

  The MFP solution for $old(\mathbf{I})$ is $old(S) =< old(\mathcal{VT}), old(\mathcal{VB}) >$ where

$$\begin{aligned} old(\mathcal{VT}) &= \{\text{OUT}_1^1, \text{OUT}_2^2, \text{IN}_3^2, \text{IN}_4^2, \text{OUT}_5^3, \text{IN}_6^3, \text{OUT}_6^3\} \\ old(\mathcal{VB}) &= U - old(\mathcal{VT}) \end{aligned}$$

- $\mathbf{I} =< G, M >=< old(G), M >$:

  The deletion of assignment to c in node 4 modifies the bit functions for expressions e_2 and e_3.

  – Preliminary observations :

    1. Function change : Since available expressions analysis is an intersection problem, TOP is **T** and BOT is **F**. The following are the constant properties and the functions associated with node 4 :

	Transp		Comp		$\mathcal{B}_h^m(x)$		$\mathcal{B}_h^m$
	e_2	e_3	e_2	e_3	e_2	e_3	(for both e_2 and e_3)
old	**F**	**F**	**F**	**F**	**F**	**F**	*lower*
new	**T**	**T**	**F**	**F**	x	x	*propagate*

       Thus, $\partial\mathcal{B}_h^m \equiv l \twoheadrightarrow p$ for both the expressions. The affected properties are : OUT_4^2 and OUT_4^3.

    2. Sets containing the influencing functions are :

$$\Omega(\text{OUT}_4^2) \;=\; \Omega(\text{OUT}_4^3) \;=\; \{propagate\}$$

    3. Sets containing the corresponding properties are :

$$\begin{aligned} \propto(\text{OUT}_4^2) &= \{\text{OUT}_i^2 \mid 1 \le i \le 6\} \cup \{\text{IN}_i^2 \mid 1 \le i \le 6\} \\ \propto(\text{OUT}_4^3) &= \{\text{OUT}_i^3 \mid 1 \le i \le 6\} \cup \{\text{IN}_i^3 \mid 1 \le i \le 6\} \end{aligned}$$

  – TR_0 computation :

    Since $old(\mathcal{B}_h^m) \equiv lower$, $lower \notin \Omega(\text{OUT}_4^2)$ and $lower \notin \Omega(\text{OUT}_4^3)$,

$$\begin{aligned} \mathcal{TR}^+ &= \emptyset \\ \mathcal{TR}^- &= \{\text{OUT}_4^2, \text{OUT}_4^3\} \\ TR_0 &= old(TR_0) - \mathcal{TR}^- \;=\; \{\text{IN}_1^1, \text{IN}_1^2, \text{IN}_1^3, \text{OUT}_3^1, \text{OUT}_3^2\} \end{aligned}$$

  – Computing the new MFP :

    1. Computing $\mathcal{LC} =< \mathcal{B2T}, \mathcal{T2B} >$

$$\begin{aligned} \mathcal{B2T} &= \{\text{OUT}_4^2, \text{OUT}_4^3\} \\ \mathcal{T2B} &= \emptyset \end{aligned}$$

    2. Computing $\mathcal{GC} =< \mathcal{B2T}\star, \mathcal{T2B}\star >$:

       Since $\mathcal{T2B}$ is $\emptyset$, $\mathcal{T2B}\star$ is $\emptyset$.

We need to identify chains starting from the properties in $\mathcal{B2T}$. It is easy to see that the chains exist between OUT_4^2 and the following properties : $\text{IN}_5^2, \text{OUT}_5^2, \text{IN}_2^2, \text{IN}_6^2, \text{OUT}_6^2$ (i.e. there exists an information flow path along which all flow functions are *propagate*). It can be verified that all of them are in $old(\mathcal{VB})$ (i.e. all of them are BOT in $old(S)$). Of these, some properties transitively depend on OUT_3^2 (i.e. chains exist between OUT_3^2 and these properties) which is in TR_0. Thus,

$$\begin{aligned}
\mathcal{GCT}(\text{OUT}_4^2) &= \{\text{OUT}_4^2, \text{IN}_5^2, \text{OUT}_5^2, \text{IN}_2^2, \text{IN}_6^2, \text{OUT}_6^2\} \\
\mathcal{NCT}(\text{OUT}_4^2) &= \{\text{IN}_5^2, \text{OUT}_5^2, \text{IN}_2^2, \text{IN}_6^2, \text{OUT}_6^2\}
\end{aligned}$$

Similarly, $\mathcal{GCT}(\text{OUT}_4^3) = \{\text{OUT}_4^3, \text{IN}_5^3\}$. However, both the properties transitively depend on IN_1^3 (i.e. chains exist between IN_1^3 and these properties) which is **F** ($\text{IN}_1^3 \in TR_0$). Hence, $\mathcal{NCT}(\text{OUT}_4^3)$ is $\{\text{OUT}_4^3, \text{IN}_5^3\}$. Thus,

$$\begin{aligned}
\mathcal{B2T}\star &= \bigcup_{p \,\in\, \mathcal{B2T}} (\mathcal{GCT}(p) - \mathcal{NCT}(p)) \\
&= (\mathcal{GCT}(\text{OUT}_4^2) - \mathcal{NCT}(\text{OUT}_4^2)) \cup \\
&\quad\; (\mathcal{GCT}(\text{OUT}_4^3) - \mathcal{NCT}(\text{OUT}_4^3)) \\
&= \{\text{OUT}_4^2\}
\end{aligned}$$

Thus the only property that changes is OUT_4^2. All other properties retain their old values.

3. Computing $S = < \mathcal{VT}, \mathcal{VB} >$:

$$\begin{aligned}
\mathcal{VT} &= old(\mathcal{VT}) - \mathcal{T2B}\star \cup \mathcal{B2T}\star \\
&= old(\mathcal{VT}) \cup \mathcal{B2T}\star \\
&= \{\text{OUT}_1^1, \text{OUT}_2^2, \text{IN}_3^2, \text{IN}_4^2, \text{OUT}_5^3, \text{IN}_6^3, \text{OUT}_6^3\} \cup \{\text{OUT}_4^2\} \\
&= \{\text{OUT}_1^1, \text{OUT}_2^2, \text{IN}_3^2, \text{OUT}_4^2, \text{IN}_4^2, \text{OUT}_5^3, \text{IN}_6^3, \text{OUT}_6^3\} \qquad (9.15) \\
\mathcal{VB} &= old(\mathcal{VB}) - \mathcal{B2T}\star \cup \mathcal{T2B}\star \\
&= old(\mathcal{VB}) - \mathcal{B2T}\star \\
&= U - old(\mathcal{VT}) - \mathcal{B2T}\star \\
&= U - (old(\mathcal{VT}) \cup \mathcal{B2T}\star) \\
&= U - \mathcal{VT}
\end{aligned}$$

□

Example 9.4 : Now we consider example 9.2 in which the expression e_3 in node 5 is deleted. The MFP solution S defined in the previous example now becomes $old(S)$ for this example and $\mathcal{VT}$ defined in (9.15) becomes $old\,(\mathcal{VT})$.

- Function change :

	Transp_5^3	Comp_5^3	$\mathcal{B}_h^m(x)$	$\mathcal{B}_h^m$
old	**T**	**T**	**T**	*raise*
new	**T**	**F**	x	*propagate*

  Thus, $\partial\mathcal{B}_h^m \equiv r \dashrightarrow p$.

- TR_0 computation :

  Since *lower* function is not involved in the change, $\mathcal{TR}^+ = \mathcal{TR}^- = \emptyset$ and TR_0 is the same as $old\,(TR_0)$.

- Computing the new MFP :

  1. Computing $\mathcal{LC} =< \mathcal{B2T}, \mathcal{T2B} >$

     It is clear that $\mathcal{B2T} = \emptyset$.

     It can be verified from (9.15) that $old\,(p') = old\,(\text{IN}_5^3) = \text{BOT}$ (since IN_5^3 is not in $old\,(\mathcal{VT})$). Hence from equation 9.8,

     $$\mathcal{T2B} \;=\; \{\text{OUT}_5^3\}$$

  2. Computing $\mathcal{GC} =< \mathcal{B2T}\star, \mathcal{T2B}\star >$

     Since $\mathcal{B2T}$ is $\emptyset$, $\mathcal{B2T}\star$ is $\emptyset$.

     There is no definition of either c or d in the program for this instance and the properties which depend on OUT_5^3 through some chain are : $\text{IN}_6^3, \text{OUT}_6^3, \text{IN}_2^3, \text{OUT}_2^3$, $\text{IN}_3^3, \text{OUT}_3^3, \text{IN}_4^3$, OUT_4^3, and IN_5^3. However, most of these properties are BOT in $old\,(S)$. From (9.15), the only properties which are TOP are : IN_6^3 and OUT_6^3. Thus, from equation 9.9,

     $$\begin{aligned} \mathcal{GCB}(\text{OUT}_5^3) &= \{\text{OUT}_5^3, \text{IN}_6^3, \text{OUT}_6^3\} \\ \mathcal{T2B}\star &= \mathcal{GCB}(\text{OUT}_5^3) \;=\; \{\text{OUT}_5^3, \text{IN}_6^3, \text{OUT}_6^3\} \end{aligned}$$

  3. Computing $S =< \mathcal{VT}, \mathcal{VB} >$:

     $$\begin{aligned} \mathcal{VT} &= old\,(\mathcal{VT}) - \mathcal{T2B}\star \cup \mathcal{B2T}\star \\ &= old\,(\mathcal{VT}) - \mathcal{T2B}\star \end{aligned}$$

$$
\begin{aligned}
&= \{\mathrm{OUT}_1^1, \mathrm{OUT}_2^2, \mathrm{IN}_3^2, \mathrm{OUT}_4^2, \mathrm{IN}_4^2, \mathrm{OUT}_5^3, \mathrm{IN}_6^3, \mathrm{OUT}_6^3\} - \\
&\quad \{\mathrm{OUT}_5^3, \mathrm{IN}_6^3, \mathrm{OUT}_6^3\} \\
&= \{\mathrm{OUT}_1^1, \mathrm{OUT}_2^2, \mathrm{IN}_3^2, \mathrm{OUT}_4^2, \mathrm{IN}_4^2\} \\
\mathcal{VB} &= U - \mathcal{VT}
\end{aligned}
$$

□

9.4 Miscellaneous Issues in Incremental Data Flow Analysis

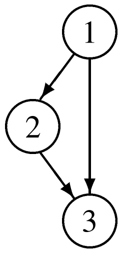
This section discusses several issues which were not addressed till now for the sake of simplicity. As this discussion reveals, the proposed model is elegant in that it handles many seemingly complicated situations very naturally, without requiring any special treatment. This is possible only because the concepts and the definitions of the model are based on sound theoretical foundations are not ad hoc.

9.4.1 Handling Bidirectional Data Flows

The functional model of incremental data flow analysis derives its basis from the fundamental principles of the generalised theory of data flow analysis. Thus, it inherently distinguishes between the entry and exit points of a node thereby handling unidirectional and bidirectional data flows uniformly. Though it goes further and develops additional concepts and principles to theorise incremental data flow analysis, it does not make any assumption/proposition which contradicts the notions involved in bidirectional data flows. However, the following three issues specific to bidirectional data flows may seem to require special treatment.

1. All known unidirectional problems have only one flow function associated with a node and all flow functions associated with edges are identity functions. Changes in a node do not affect edge flow functions in any way but may cause the node flow function to change. The important point to note is that at most one flow function may change due to a change in a node.

   Bidirectional problems may, however, have more than one function associated with a node. Besides, the edge flow functions may be of the form $h(X) = C_1 + C_2 \cdot X$ where C_1 and C_2 could change due to a change in the node. Thus, multiple flow functions could change simultaneously due to a change in a single node. It is shown in section 9.4.3 that such a situation can be handled by the model very easily. In fact, the changes



For the chain $ch \equiv \text{PPIN}^l_3 \xrightarrow{g^b} \text{PPOUT}^l_2 \xrightarrow{f^b} \text{PPIN}^l_2 \xrightarrow{g^b} \text{PPOUT}^l_1$,

$\text{PPOUT}^l_1 \in \mathcal{D}(\text{PPIN}^l_3)$, i.e. PPIN^l_3 depends on PPOUT^l_1 through g^f

Figure 9.6: Cyclic dependence in MRA in the absence of a strongly connected component

need not be restricted to a single node; changes in several nodes can also be processed simultaneously.

Example 9.5 : Consider node 2 in the program flow graph of Figure 9.2. If expression $b*c$ is deleted, then ANTLOC^l_2 becomes **F** hence f^b_2 for e_2 (which is $\text{ANTLOC}^l_2 + \text{TRANSP}^l_2 \cdot X$), changes. Similarly, since AVOUT^l_2 becomes **F**, the edge flow function $g^f_{(2,3)}$ for e_2 (which is $\text{AVOUT}^l_2 + X$), also changes. Thus, due to a single change, multiple flow functions for the same property could change. In the case of available expressions analysis, only f^f_2 would have changed due to the removal of $b*c$ from node 2. □

2. Due to the presence of flows in both the directions, cyclic dependences may arise in bidirectional flows even in the absence of a strongly connected component. Figure 9.6 contains such an example for MRA. It is shown in section 9.4.2 that the definitions in the functional model handle the cyclic dependences automatically without requiring any special treatment. Thus, neither the unidirectional nor the bidirectional data flow analysis requires any special provision for handling the cyclic dependences.

3. The CONST_IN/CONST_OUT properties (i.e. the constant properties associated with the entry/exit of a node) are $\top$ for all known unidirectional problems. They may be non-$\top$ for bidirectional problems (see section 3.3). Thus a property could be in TR_0 due to the influence of CONST_IN/CONST_OUT and computing new TR_0 from flow functions alone may not suffice.

Example 9.6 : For MRA, CONST_IN_i is PAVIN_i. The initial value of PPIN^l_i is BOT if PAVIN^l_i is BOT. Thus if PAVIN^l_i changes, this change is not a change in any flow function, and yet the value of PPIN^l_i and its membership in TR_0 is affected. □

Handling this situation is really very easy. Recall that the data flow equations can be written in an abstract form as :

$$X(u) = \bigsqcap_{v \in \mathcal{N}^{-1}(u)} h(X(v)) \sqcap \text{CONST}(u)$$

where $\text{CONST}(u)$ is either CONST_IN_i or CONST_OUT_i depending upon whether u is $in(i)$ or $out(i)$. Let the flow function h be $h(X) = C_1 + C_2 \cdot X$. Then the data flow equation can be re-written as

$$X(u) = \bigsqcap_{v \in \mathcal{N}^{-1}(u)} h'(X(v))$$

where h' is defined as $h'(X) = (\text{CONST}(u) \sqcap C_1) + (\text{CONST}(u) \sqcap C_2) \cdot X$. Effectively, the constant property which was "outside" is "pulled inside" the flow function[7], and a change in a constant property can be treated as a change in a flow function. No additional provision is required for handling constant properties.

9.4.2 Handling Cyclic Dependences

Consider an influence $p \leftarrow \mathcal{B}_h^m(p')$. Let there be an information flow path from p to p'. Then, $p' \leftarrow \mathcal{B}_h(p)$ where $\mathcal{B}_h$ represents the composition of all flow functions along the information flow path from p to p'. It is easy to see that :

$$p \leftarrow \mathcal{B}_h^m(p') \text{ and } p' \leftarrow \mathcal{B}_h(p) \Rightarrow p \leftarrow \mathcal{B}_h^m(\mathcal{B}_h(p))$$

Traditionally, such a situation is handled in two steps :

1. Identify the existence of an information flow path from p to p'.

   This involves identifying a strongly connected component in the graph. This approach is error prone in case of bidirectional problems where information flow paths do not necessarily follow the graph theoretic paths and a cyclic dependence may exist even in the absence of a strongly connected component.

2. Find the influence of $\mathcal{B}_h(p)$ to determine the final value of p.

   This involves traversing the strongly connected component.

In this section, we show that these two steps are redundant in the proposed model.

[7]This needs to be done for the node flow functions only.

Observation 9.7 : *For an influence* $p \leftarrow \mathcal{B}_h^m(p')$. *There is no need to identify strongly connected components unless* $CH(p, p') \neq \emptyset$. $\square$

Even if an information flow path along a strongly connected component exists, p cannot influence p' unless (i) all the functions along the information flow path are *propagate*, and (ii) no property along the path is influenced by a *lower* function. Further, unless $\mathcal{B}_h^m$ is *propagate*, p does not depend on p' (i.e. the value of p' does not influence p). Hence we conclude that there is no need to identify and traverse the strongly connected component unless $\mathcal{B}_h^m \equiv$ *propagate*, and a chain from p to p' exists.

From observation 8.3, all properties in a chain giving rise to a cyclic dependence must necessarily have the same value in a fixed point solution. The functional model ensures this without requiring any special treatment of strongly connected components.

Lemma 9.1 : *Consider an influence* $p \leftarrow \mathcal{B}_h^m(p')$ *and a* $ch \in CH\,(p, p')$ *such that ch gives rise to a cyclic dependence. Then, all properties in ch have the same value in the solution computed by the functional model.*

Proof : Let $old\,(ch) \equiv p_1 \xrightarrow{\mathcal{T}_1} p_1 \xrightarrow{\mathcal{T}_2} p_2 \cdots \xrightarrow{\mathcal{T}_n} p_n$ where $p_1 \equiv p$ and $p_2 \equiv p'$. Since ch does not include p, we know that $ch \equiv old\,(ch)$. Let p_i be the first BOT property in $old\,(ch)$. Then, all properties from p_i to p_n (both inclusive) must necessarily be BOT and all properties from p_1 to p_{i-1} (both inclusive) must necessarily be TOP in $old\,(\mathbf{I})$.

Since $\mathcal{B}_h^m$ is *propagate*, $\partial\mathcal{B}_h^m \in \sigma_\gamma^2(\Delta)$. We consider all cases in the following :

1. $old\,(p) =$ BOT. By the above argument, $old\,(p')$ must be BOT. The two changes in $\sigma_\gamma^2(\Delta)$ are,

   (a) $\partial\mathcal{B}_h^m \equiv l \twoheadrightarrow p$: From equation 9.7, $p \in \mathcal{B}2\mathcal{T}$. Since ch exists in $\mathbf{I}$, every p_i is contained in $\mathcal{GCT}(p)$. Two possibilities exist for these properties :

      i. $\exists\, p'' \in TR_0$ such that $\mathrm{CH}(p'', p_j) \neq \emptyset$. In such a case, since a cyclic dependence exists, a chain exists from p'' to every p_i (via p_j). Thus every p_i is in $\mathcal{NCT}(p)$ and its value is BOT.

      ii. $\nexists\, p'' \in TR_0$ such that $\mathrm{CH}(p'', p_j) \neq \emptyset$ for any p_j. Then, all properties are in $\mathcal{GCT}(p) - \mathcal{NCT}(p)$ and all of them are TOP.

   (b) $\partial\mathcal{B}_h^m \equiv r \twoheadrightarrow p$: Since $old\,(p)$ is BOT, $p \notin \mathcal{T}2\mathcal{B}$. Hence the value of p remains BOT and no property changes.

2. $old\,(p) =$ TOP. We consider the two cases in $\sigma_\gamma^2(\Delta)$,

$$\begin{array}{lcl} \mathcal{TR_I}^+(h^m) & : & \mathcal{TR}^+ \text{ set for a function } h^m \in \mathcal{H}^m. \\ \mathcal{TR_I}^-(h^m) & : & \mathcal{TR}^- \text{ set for a function } h^m \in \mathcal{H}^m. \\ \mathcal{T2B}_I(h^m) & : & \mathcal{T2B} \text{ set for a function } h^m \in \mathcal{H}^m. \\ \mathcal{B2T}_I(h^m) & : & \mathcal{B2T} \text{ set for a function } h^m \in \mathcal{H}^m. \end{array}$$

Figure 9.7: Intermediate sets corresponding to h

(a) $\partial\mathcal{B}_h^m \equiv l \nrightarrow p$: Since $old(\mathcal{B}_h^m)$ is *lower*, $old(p)$ cannot be TOP and hence this possibility is ruled out.

(b) $\partial\mathcal{B}_h^m \equiv r \nrightarrow p$: This is very much possible and we need to consider $old(p')$ for further analysis.

   i. $old(p') =$ BOT : In this case, $p \in \mathcal{T2B}$. Let p_i be the first BOT property in $old(ch)$. Then, the old values of all properties from p_1 to p_{i-1} must be TOP. All these properties will be contained in $\mathcal{GCB}(p)$. Since all properties from p_i to p_n retain their old values, once again all properties in *ch* are BOT.

   ii. $old(p') =$ TOP : In this case, p is neither in $\mathcal{T2B}$ nor in $\mathcal{B2T}$, and $\mathcal{GCB}(p) = \mathcal{GCT}(p) = \emptyset$. However, since $old(p')$ is TOP, it implies that all the properties in *ch* are TOP. Now, the value of p remains TOP, and once again, all the properties in *ch* have the same value.

Thus, in each case all properties in *ch* have the same value. This is achieved by the functional model automatically without the need of knowing about the presence of a strongly connected component. □

It is shown in chapter 12 that the resulting solution is the MFP solution.

9.4.3 Handling Multiple Function Changes

The proposed model is easily extended to multiple function changes. Instead of processing a single modified bit vector function h^m, we define a set $\mathcal{H}^m$ to contain all functions which have changed. For each set in the proposed model, we define a corresponding set for each function $h^m \in \mathcal{H}^m$ (see Figure 9.7). Each of these sets is defined much in the same way like the original set.

9.4.3.1 Computing TR_0

In the presence of multiple function changes, $\mathcal{TR}^+$ and $\mathcal{TR}^-$ are defined as follows :

$$
\begin{aligned}
\mathcal{TR}_I{}^+(h^m) &= \{p \mid p \leftarrow \mathcal{B}_h^m(p') \text{ s.t. } \mathcal{B}_h^m \equiv \mathit{lower},\ h^m \in \mathcal{H}^m\} \\
\mathcal{TR}_I{}^-(h^m) &= \{p \mid p \leftarrow \mathcal{B}_h^m(p') \text{ s.t. } \mathit{old}\,(\mathcal{B}_h^m) \equiv \mathit{lower},\ h^m \in \mathcal{H}^m \text{ and } \mathit{lower} \notin \Omega(p)\} \\
\mathcal{TR}^+ &= \bigcup_{h^m \,\in\, \mathcal{H}^m} \mathcal{TR}_I{}^+(h^m) \\
\mathcal{TR}^- &= \bigcup_{h^m \,\in\, \mathcal{H}^m} \mathcal{TR}_I{}^-(h^m)
\end{aligned}
$$

TR_0 definition is same as in the case of a single function change :

$$TR_0 = \mathit{old}\,(TR_0) \cup \mathcal{TR}^+ - \mathcal{TR}^-$$

where the operations are performed left to right.

9.4.3.2 Local change

Recall that local change $\mathcal{LC}$ is defined as $\mathcal{LC} =< \mathcal{B2T}, \mathcal{T2B} >$. The sets $\mathcal{B2T}$ and $\mathcal{T2B}$ are defined in terms of $\mathcal{B2T}_I(h)$ and $\mathcal{T2B}_I(h)$ sets for individual functions in $\mathcal{H}^m$.

$$
\begin{aligned}
\mathcal{B2T}_I(h^m) &= \{p \mid p \leftarrow \mathcal{B}_h^m(p'),\ \mathcal{B}_h^m(\text{TOP}) = \text{TOP}, [\mathit{old}\,(\mathcal{B}_h^m)]\,(\mathit{old}\,(p')) = \text{BOT} \\
&\qquad \text{and } \mathit{lower} \notin \Omega(p), h^m \in \mathcal{H}^m\} \\
\mathcal{T2B}_I(h^m) &= \{p \mid p \leftarrow \mathcal{B}_h^m(p'),\ \mathcal{B}_h^m(\text{BOT}) = \text{BOT and } \mathit{old}\,(p) = \text{TOP and} \\
&\qquad \mathit{old}\,(p') = \text{BOT if } \mathcal{B}_h^m \equiv \mathit{propagate}, h^m \in \mathcal{H}^m\}
\end{aligned}
$$

These definition are essentially the same as the original definitions (equations 9.7 – 9.8). The overall $\mathcal{B2T}$ and $\mathcal{T2B}$ sets are defined as follows :

$$
\begin{aligned}
\mathcal{B2T} &= \bigcup_{h^m \,\in\, \mathcal{H}^m} \mathcal{B2T}_I(h^m) \\
\mathcal{T2B} &= \bigcup_{h^m \,\in\, \mathcal{H}^m} \mathcal{T2B}_I(h^m)
\end{aligned}
$$

Effectively, local change computation accumulates the local changes all over the graph[8], i.e. the sets $\mathcal{B2T}$ and $\mathcal{T2B}$ may contain the properties at different program points unlike in the case of a single function change when they contain the properties at a single program point.

[8] Is it fair to call them *local*?

9.4.3.3 Global change

Having accumulated the influence of all bit vector functions which have changed, we compute the global change $\mathcal{GC}$ which is $< \mathcal{B}2\mathcal{T}\star, \mathcal{T}2\mathcal{B}\star >$ by computing $\mathcal{GCT}(p)$, $\mathcal{NCT}(p)$ sets for all properties in $\mathcal{B}2\mathcal{T}$ and $\mathcal{GCB}(p)$ for all properties in $\mathcal{T}2\mathcal{B}$. Interestingly, all definitions for global change computation remain same even if there are multiple function changes. This is possible since even though there could be conflicting influences on the value of a property, it is automatically taken care of by the definitions in the model.

It is shown in section 12.5 that the possibility of several functions changing together does not constrain any proposition of the model and there is no need to perform separate analysis for different changes; all changes can be handled simultaneously. This facilitates handling the structural changes because a structural change often implies changes in several functions.

9.4.4 Handling Structural Changes

Recall that a data flow framework is defined as a triple $\mathbf{D} = < \mathcal{L}, \sqcap, \mathcal{F} \cup \mathcal{G} >$ while its instance is defined by the ordered pair $\mathbf{I} =< G, M >$ where M maps the flow functions to the nodes/edges of the control flow graph. So far we have considered the changes in M (alternatively, in $\mathcal{F}$ or $\mathcal{G}$) only. Now we consider the other changes called the *structural changes*.

Two kinds of structural changes may take place in an instance of a data flow framework :

1. *Changes in the lattice $\mathcal{L}$.*

   This implies changing the structure of a lattice element, i.e. adding a new expression in the program or removing all computations of an existing expression from the program. Such a change implies that some bits are added to or deleted from each lattice element. Such a change needs to be processed using exhaustive analysis since old MFP solution for these bits is not available; it must be computed from scratch. Thus, this problem is beyond the purview of incremental data flow analysis.

2. *Changes in the graph G.*

   The following kinds of changes may take place in a graph :

   (a) *Deletion of an edge* :

   Consider an edge $e = (i, j)$. Then, possibly two flow functions $h(u, u')$ and $h'(u', u)$ have been deleted where $u \equiv out(i)$ and $u' \equiv in(j)$. We know that a non-existent function is essentially a constant function $\top$. Thus effectively, some functions may have changed to $\top$ and hence the properties may change from BOT to TOP. Thus, $\mathcal{B}2\mathcal{T}$ should be computed at u and u'. Global change can then be found by computing $\mathcal{GCT}$ and $\mathcal{NCT}$ sets.

(b) *Insertion of an edge* :

Let the inserted edge be $e = (i, j)$. Then, possibly two new flow functions $h(u, u')$ and $h'(u', u)$ may have been created where $u \equiv out(i)$ and $u' \equiv in(j)$. These functions did not exist in $old(\mathbf{I})$. In other words, the influence of these function was $\top$ in $old(\mathbf{I})$. Thus, the changes can only be from TOP to BOT which can be identified by computing $\mathcal{T}2\mathcal{B}$ at u and u'. Global change can then be found by computing $\mathcal{GCB}$ sets.

(c) *Deletion of a node* :

Deletion of a node may have the following two connotations :

i. All in/out edges of the node are deleted.
This basically is the mathematical notion deletion of a node. In this case, the desired change can be modelled in terms of the approach mentioned in step 2a. Note that the multiplicity of edges does not affect the complexity since the number of edges can be safely assumed to be bounded by a constant.[9]

ii. The in-edges of the node are joined with out-edges, i.e. every successor of the node becomes a successor of every predecessor of the node.
This is what the node deletion may practically imply. In this case the desired computation can be defined as in the next step.

(d) *Insertion of a node* :

Let node k be inserted by splitting an edge (i, j). For brevity, let u and v denote $out(i)$ and $in(j)$ respectively. Let u' and v' denote $in(k)$ and $out(k)$ respectively. Then, the following changes have taken place :

- $h(u, v)$ and $h(v, u)$ have become $\top$.
- The following functions which were non-existent (i.e. $\top$) in $old(\mathbf{I})$, have been added,
  - $h(u, u')$ and $h(u', u)$,
  - $h(u', v')$ and $h(v', u')$,
  - $h(v', v)$ and $h(v, v')$,

[9]This is in consonance with the assumption that $|\ E\ |$ is $O(|\ N\ |)$.

We need to find $\mathcal{T2B}$ and $\mathcal{B2T}$ sets for all these functions. The influence of some flow functions becoming $\top$ can be incorporated in the influence of the new functions. As a first step, compute (mutually consistent) IN_k and OUT_k. Then, compute the $\mathcal{T2B}$ and $\mathcal{B2T}$ sets as defined below :

- Local change at u.

$$\begin{aligned}\mathcal{T2B}(h(u',u)) &= \{p \mid p \text{ is TOP in } \mathit{old}(\mathrm{OUT}_i) \text{ and is BOT in} \\ &\quad [h(u',u)](\mathrm{IN}_k)\} \\ \mathcal{B2T}(h(u',u)) &= \{p \mid p \text{ is BOT in } [\mathit{old}(h(v,u))](\mathit{old}(\mathrm{IN}_j)), \\ &\quad \text{is TOP in } [h(u',u)](\mathrm{IN}_k), \text{ and } \mathit{lower} \notin \Omega(p)\}\end{aligned}$$

- Local change at u'.

$$\begin{aligned}\mathcal{T2B}(h(u,u')) &= \{p \mid p \text{ is BOT in } \mathrm{IN}_k \text{ and the corresponding} \\ &\quad \text{property is TOP in } \mathit{old}(\mathrm{IN}_j)\ \} \\ \mathcal{B2T}(h(u,u')) &= \{p \mid p \text{ is TOP in } \mathrm{IN}_k \text{ and the corresponding} \\ &\quad \text{property is BOT in } \mathit{old}(\mathrm{IN}_j)\ \}\end{aligned}$$

- Local change at v'.

$$\begin{aligned}\mathcal{T2B}(h(v,v')) &= \{p \mid p \text{ is BOT in } \mathrm{OUT}_k \text{ and the corresponding} \\ &\quad \text{property is TOP in } \mathit{old}(\mathrm{OUT}_i)\ \} \\ \mathcal{B2T}(h(v,v')) &= \{p \mid p \text{ is TOP in } \mathrm{OUT}_k \text{ and the corresponding} \\ &\quad \text{property is BOT in } \mathit{old}(\mathrm{OUT}_i)\ \}\end{aligned}$$

- Local change at v.

$$\begin{aligned}\mathcal{T2B}(h(v',v)) &= \{p \mid p \text{ is TOP in } \mathit{old}(\mathrm{IN}_j) \text{ and is BOT in} \\ &\quad [h(v',v)](\mathrm{OUT}_k)\} \\ \mathcal{B2T}(h(v',v)) &= \{p \mid p \text{ is BOT in } [\mathit{old}(h(u,v))](\mathit{old}(\mathrm{OUT}_i)), \\ &\quad \text{is TOP in } [h(v',v)](\mathrm{OUT}_k), \text{ and } \mathit{lower} \notin \Omega(p)\}\end{aligned}$$

9.5 Looking Back

It is easy to see that the proposed model is independent of any particular technique. This independence is achieved by clearly defining the incremental solution of a data flow problem. The advantage of this model is that since the correctness of this model is formally proved (chapter 12), the correctness of any technique of incremental data flow analysis automatically follows if it can be shown that it computes the sets defined in the model correctly. Another significant advantage is that no special treatment is required for strongly connected components. Besides, it is capable of handling bidirectional data flows, multiple function changes, and structural changes. Chapter 10 develops an iterative algorithm which is based on the proposed model; chapter 11 develops four variants of a wordwise algorithm.

Chapter 10

Performing Incremental Data Flow Analysis : The Bitwise Approach

I pointed out to the little prince that baobabs were not little bushes, but, on the contrary, trees as big as castles; ... But he made a wise comment : "Before they grow so big, the baobabs start out by being little."

This chapter presents an iterative algorithm for performing incremental data flow analysis according to the functional model proposed in chapter 9. This algorithm processes each property in the bit vector separately. For the sake of simplicity, we ignore the extensions for multiple function changes and structural changes and restrict ourselves to changes in one bit vector function.

We start from the basic definitions and develop the algorithm in an evolutionary style. The pseudo-code is written in Pascal-like language.

10.1 Preliminaries

We use the terms *sets* and *bit vectors* interchangeably. We assume that the bit function $\mathcal{B}_h^i$ is $\mathcal{B}_h^i(x) = C_1^i + C_2^i \cdot x$. We use the following notation for presenting the algorithm :

$Pos(p)$: Bit position of a property p in the bit vectors.
$X(u)$: Bit vector representing the properties at program point u.
$X^i(u)$: Property represented by the i^{th} bit at program point u.
$\mathcal{B}_h^i$: Bit function for the i^{th} property in a bit vector.

Let the modified bit vector function be h^m. The first task is to determine those bit functions of h^m which have changed. Let $h^m(Y) = C_1^m + C_2^m \cdot Y$ and H^m be the bit vector representing the bit functions which have changed.[1] Let $\oplus$ represent the boolean EX-OR operation. Then, $H^m = old\,(C_1^m) \oplus C_1^m + old\,(C_2^m) \oplus C_2^m$.

```
procedure retain_TOP_bits(x, y)
{     /* Identify those bits in y which are TOP in y and are T in x */
      if (⊓ is Π) then
            return x · y
      else if (⊓ is Σ) then
            return x · ¬y
}
procedure retain_BOT_bits(x, y)
{     /* Identify those bits in y which are BOT in y and are T in x */
      if (⊓ is Π) then
            return x · ¬y
      else if (⊓ is Σ) then
            return x · y
}
procedure make_bits_BOT(x, y)
{     /* If a bit is T in y, make it BOT in x */
      if (⊓ is Π) then
            return x · ¬y
      else if (⊓ is Σ) then
            return x + y
}
procedure make_bits_TOP(x, y)
{     /* If a bit is T in y, make it TOP in x */
      if (⊓ is Π) then
            return x + y
      else if (⊓ is Σ) then
            return x · ¬y
}
```

Figure 10.1: Some binary operations on lattice elements

Since the algorithm should handle both confluence operators (i.e. union and intersection), we abstract out the operator dependent computations. Figures 10.1 contains procedures for some binary operations on the lattice elements while Figure 10.2 contains some procedures to compute the sets of functions.

[1] Note that H^m is different from $\mathcal{H}^m$ defined in section 9.4.3.

```
procedure lower_functions(h)        /* h(X) = C1 + C2·X */
{     if (⊓ is Π) then
            return ¬C1 · ¬C2
      else if (⊓ is Σ) then
            return C1
}
procedure raise_functions(h)        /* h(X) = C1 + C2·X */
{     if (⊓ is Π) then
            return C1
      else if (⊓ is Σ) then
            return ¬C1 · ¬C2
}
procedure propagate_functions(h)    /* h(X) = C1 + C2·X */
{     return ¬C1 · C2
}
```

Figure 10.2: Computing sets of functions

10.2 Computing TR_0

TR_0 is implemented as a property vector, i.e. if a property associated with the program point u is in TR_0, the corresponding bit is **T** in the bit vector $TR_0(u)$. Thus, the sets $\mathcal{TR}^+$ and $\mathcal{TR}^-$ influence the bit vector $TR_0(u)$ where $u = \mathcal{P}p(p)$ such that $p \leftarrow \mathcal{B}_h^m(p')$. For a $p \in \mathcal{TR}^-$, there should be no *lower* function in $\Omega(p)$.

Presence of a *lower* function in $\Omega(p)$ influences the chains and and $\mathcal{B}2\mathcal{T}$ set. Thus, we need to identify if a property is being influenced by a *lower* function. Testing whether there exists a *lower* function can be carried out by simply testing for the membership of p in TR_0. This is precisely why the TR_0 computation is required.

10.3 Computing the Local Change

Procedure *find_local_change* in Figure 10.4 computes the local influence of the modifications in function $h^m : X(u) \leftarrow h^m(X(u'))$. Since $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ contain the properties which change, we need to know the influence of the properties at u' through $old\,(h^m)$ on the properties at u. Note that $\mathcal{B}_h^m(\mathrm{BOT}) = \mathrm{BOT}$ covers the *lower* as well as *propagate* functions. Procedure *find_local_change* identifies them separately since $\mathcal{T}2\mathcal{B}$ computation has an additional condi-

```
procedure compute_TR0(h^m, H^m, u)
{    TR+ := H^m · lower_functions(h^m)
     TR- := H^m · old_lower_functions(h^m)
     for each u'' ∈ N^-1(u)
     {    temp := [h(u'', u)](⊤)
          TR- := retain_TOP_bits(TR-, temp)
     }
     TR0(u) := old(TR0(u)) · ¬TR- + TR+
}
procedure old_lower_functions(h) /* h(X) = C1 + C2 · X */
{    if (⊓ is Π) then
          return ¬old(C1) · ¬old(C2)
     else if (⊓ is Σ) then
          return old(C1)
}
```

Figure 10.3: Computing TR_0

```
procedure find_local_change(h^m, H^m, u', u) /* h^m(X) = C1^m + C2^m · X */
{    oldinfluence := old(C1^m) + old(C2^m) · old(X(u'))
     temp := retain_BOT_bits(propagate_functions(h^m), old(X(u')))
     botinfluence := lower_functions(h^m) + temp
     topinfluence := raise_functions(h^m) + propagate_functions(h^m)
     B2T := H^m · retain_BOT_bits(topinfluence, oldinfluence) · ¬TR0(u)
     T2B := H^m · retain_TOP_bits(botinfluence, oldinfluence)
     X(u) := make_bits_BOT(old(X(u)), T2B)
     X(u) := make_bits_TOP(X(u), B2T)
}
```

Figure 10.4: Computing $\mathcal{LC} = < \mathcal{B}2\mathcal{T}, \mathcal{T}2\mathcal{B} >$

tion that the old value of p' should be BOT if $\mathcal{B}_h^m$ is *propagate*.

10.4 Computing the Global Change

In order to compute the global change, we need to know the properties which depend on the changed property. Figure 10.5 contains a procedure for tracing the chains of a property.

```
procedure causes_dependence(u, u', i)
{    /* [B_h^i(u, u')](x) = C_1^i + C_2^i · x */
     if ¬C_1^i · C_2^i = T then /* B_h^i ≡ propagate */
          if TR_0^i(u') = F then
               return T
     return F
}
```

Figure 10.5: Determining if $p \in \mathcal{D}(p')$ (i.e. if $X^i(u')$ depends on $X^i(u)$).

We compute $\mathcal{T2B}\star$ by constructing $\mathcal{GCB}(p)$ for each property p in $\mathcal{T2B}$. $\mathcal{B2T}\star$ is computed by constructing $\mathcal{GCT}(p)$ and $\mathcal{NCT}(p)$ for each property p in $\mathcal{B2T}$. The sets $\mathcal{GCB}(p)$ and $\mathcal{NCT}(p)$ are not constructed explicitly; instead, the graph is traversed and the value of a property is set to BOT. Since $\mathcal{NCT}(p)$ computation requires $\mathcal{GCT}(p)$, we remember the program points of the properties in $\mathcal{GCT}(p)$. The properties in $\mathcal{GCT}(p)$ are changed to TOP which may then be reset to BOT if they belong to $\mathcal{NCT}(p)$.

We consider the simpler case of $\mathcal{T2B}\star$ first.

10.4.1 Computing $\mathcal{T2B}\star$

From equation 9.9, for a $p \in \mathcal{T2B}$

$$\mathcal{GCB}(p) \quad = \quad \{p\} \cup \{p'' \mid \mathrm{CH}(p, p'') \neq \emptyset \text{ and } old\,(p'') = \mathrm{TOP}\}$$

Thus, $\mathcal{GCB}(p)$ can be computed in two steps :

1. $\mathcal{GCB}(p) = \{p\} \cup \{p'' \mid p \in \mathcal{D}(p'') \text{ and } old\,(p'') = \mathrm{TOP}\}$

2. Repeat until no more properties can be added to $\mathcal{GCB}(p)$

   $$\mathcal{GCB}(p) = \mathcal{GCB}(p) \cup \{p'' \mid \exists p' \in \mathcal{D}(p'') \cap \mathcal{GCB}(p) \text{ and } old\,(p'') = \mathrm{TOP}\}$$

Procedure *propagate_bot_influence* (Figure 10.6) computes $\mathcal{GCB}(p)$ recursively, where p is represented by $X^i(u)$.

```
procedure propagate_bot_influence(u,i)    /* i is Pos(p) and u is Pp(p) */
{     for each u' ∈ N(u)
      {     if ((causes_dependence (u,u',i) = T) and (X^i(u') = TOP)) then
            {     X^i(u) := BOT
                  propagate_bot_influence(u',i)
            }
      }
}
```

Figure 10.6: Computing $\mathcal{GCB}(p) - \{p\}$

10.4.2 Computing $\mathcal{B}2\mathcal{T}\star$

10.4.2.1 Computing $\mathcal{GCT}(p)$

From equation 9.11, for a $p \in \mathcal{B}2\mathcal{T}$

$$\mathcal{GCT}(p) \quad = \quad \{p\} \cup \{p'' \mid \text{ CH}(p, p'') \neq \emptyset \text{ and } old\,(p'') = \text{BOT}\}$$

Thus, $\mathcal{GCT}(p)$ also can be computed in two steps :

1. $\mathcal{GCT}(p) = \{p\} \cup \{p'' \mid p \in \mathcal{D}(p'') \text{ and } old\,(p'') = \text{BOT}\}$

2. Repeat until no more properties can be added to $\mathcal{GCT}(p)$.

   $$\mathcal{GCT}(p) = \mathcal{GCT}(p) \cup \{p'' \mid \exists p' \in \mathcal{D}(p'') \cap \mathcal{GCT}(p) \text{ and } old\,(p'') = \text{BOT}\}$$

The set $\mathcal{GCT}(p)$ is effectively represented by the *affected region* which is defined as :

$$\mathcal{AR}_p = \{\mathcal{P}p(p') \mid p' \in \mathcal{GCT}(p)\}. \tag{10.1}$$

The affected region represents the portion of the graph which needs to be processed for incorporating the effect of the changes in $\mathcal{B}2\mathcal{T}$. The recursive procedure which implements this definition is given in Figure 10.7, where p is represented by $X^i(u)$.

10.4.2.2 Computing $\mathcal{NCT}(p)$

From equation 9.12,

$$\mathcal{NCT}(p) \quad = \quad \{p'' \mid p'' \in \mathcal{GCT}(p) \text{ and } \exists\, p''' \in TR_0 \text{ s.t. CH}(p''', p'') \neq \emptyset\}$$

```
procedure find_global_change_to_top(u, i)   /* i is Pos(p) and u is Pp(p) */
{     AR_p := AR_p ∪ {u}
      for each u' ∈ N(u)
      {     if ((causes_dependence (u, u', i) = T) and (X^i(u') = BOT)) then
                  {     X^i(u') := TOP
                        find_global_change_to_top(u', i)
                  }
      }
}
```

Figure 10.7: Computing $\mathcal{GCT}(p)$ and $\mathcal{AR}_p$

```
procedure propagate_boundary_effect(u, i) /* i is Pos(p) and u is Pp(p) */
{     for each u ∈ AR_p
            for each u' ∈ N^{-1}(u)
                  if (u' ∉ AR_p) then                  /* Then u is in BR_p */
                        if (causes_dependence (u, u', i) = T) then
                              if ((X^i(u') = BOT) and (X^i(u) = TOP)) then
                              {     X^i(u) := BOT
                                    propagate_bot_influence(u, i)
                              }
}
```

Figure 10.8: Computing $\mathcal{NCT}(p)$

This equation can be rewritten as :

$$\mathcal{NCT}(p) \;=\; \{p'' \mid p'' \in \mathcal{GCT}(p) \text{ and } \exists p' \notin \mathcal{GCT}(p) \text{ s.t. } \mathrm{CH}(p', p'') \neq \emptyset, \text{ and } p' = \mathrm{BOT}\}$$

Thus, $\mathcal{NCT}(p)$ also can be computed in two steps :

1. $\mathcal{NCT}(p) = \{p'' \mid p'' \in \mathcal{GCT}(p) \text{ and } \exists p' \notin \mathcal{GCT}(p) \text{ s.t. } p' \in \mathcal{D}(p'') \text{ and } p' = \mathrm{BOT}\}$

2. Repeat until no more properties can be added to $\mathcal{NCT}(p)$.

   $\mathcal{NCT}(p) = \mathcal{NCT}(p) \cup \{p'' \mid p'' \in \mathcal{GCT}(p) \text{ and } \exists p' \in \mathcal{D}(p'') \cap \mathcal{NCT}(p)\}$

The first step effectively identifies the *boundary of affected region* which may be defined as :

$$\mathcal{BR}_p \;=\; \{\mathcal{P}p(p') \mid \mathcal{P}p(p') \in \mathcal{AR}_p \text{ and } \exists\, p'' \in \mathcal{N}^{-1}(p') \text{ s.t. } \mathcal{P}p(p'') \notin \mathcal{AR}_p\} \quad (10.2)$$

Procedure *propagate_boundary_effect* in Figure 10.8 computes $\mathcal{NCT}(p)$ by finding out those properties in $\mathcal{GCT}(p)$ which are neighbours of properties outside $\mathcal{GCT}(p)$. If a property outside $\mathcal{GCT}(p)$ is BOT and the property in $\mathcal{GCT}(p)$ depends on it, the latter is set to BOT and its influence is propagated to other nodes of the region.

10.5 A Generic Algorithm for Incremental Data Flow Analysis

procedure *incremental_dfa*(h^m, u', u)
$\{$ /$\star$ $X(u) \leftarrow h^m(X(u'))$ and $h^m(X) = C_1 + C_2 \cdot X$ $\star$/
$H^m := old(C_1)\cdot\neg C_1 + \neg old(C_1)\cdot C_1 + old(C_2)\cdot\neg C_2 + \neg old(C_2)\cdot C_2$
compute_TR$_0(h^m, H^m, u)$ /$\star$ TR_0 is globally available $\star$/
find_local_change(h^m, H^m, u', u) /$\star$ $\mathcal{T2B}$ and $\mathcal{B2T}$ are globally available $\star$/
for each $p \in \mathcal{T2B}$
$\{$ $i := Pos(p)$
propagate_bot_influence(u, i)
$\}$
for each $p \in \mathcal{B2T}$
$\{$ $\mathcal{AR}_p := \emptyset$ /$\star$ $\mathcal{AR}_p$ is globally available $\star$/
$i := Pos(p)$
find_global_change_to_top(u, i)
propagate_boundary_effect(u, i)
$\}$
$\}$

Figure 10.9: A generic algorithm for incremental data flow analysis

The main procedure which calls various procedures is given in Figure 10.9. Since the algorithm is a straight-forward implementation of the functional model of incremental data flow analysis, it inherits several advantages of the model.

1. It can be extended to handle multiple function changes without any significant increase in the amount of work.

2. It can handle structural changes with slight modification.

3. It is uniformly applicable to union as well as intersection problems.

4. It is uniformly applicable to unidirectional and bidirectional data flow problems.

   (a) This is the first algorithm for incremental data flow analysis of bidirectional data flow problems.

   (b) Most other techniques have to provide separate procedures for forward and backward data flow problems. In contrast this algorithm is uniformly applicable to forward and backward data flow problems.

5. Unlike most other approaches, this algorithms does not require the implementor to perform an exhaustive case analysis of changes and their possible implications. In this sense this a rather clean and elegant algorithm which can be very easily adapted to any (singular) bit vector data flow problem.

6. Since the algorithm faithfully implements the definitions of the entities constituting the functional model, its correctness automatically follows.

7. Unlike most other algorithms (elimination as well as iterative), this algorithm does not need to identify any strongly connected component.

10.6 Implementation Notes

The generic incremental data flow analysis algorithm proposed in this chapter has been implemented and thoroughly tested for both unidirectional and bidirectional data flows. The problem of available expressions analysis was chosen as a representative unidirectional data flow problem while MRA was chosen as a representative bidirectional data flow problem.

The implementation framework consisted of the global optimiser phase of the optimising FORTRAN compiler developed at IIT Bombay as a part of ongoing research in code optimisation. The following strategy was used for performing incremental data flow analysis :

> The global optimiser performs exhaustive analysis for a host of data flow problems including the problem of available expressions analysis and MRA. The resulting solutions are stored for a later use in incremental analysis. Global optimisation is performed using the Composite Hoisting and Strength Reduction Algorithm (see

section 1.2.3). This causes several (non-structural) changes in the program flow graph. The local properties are recomputed and data flow analysis is performed incrementally by retaining the changes in one node at a time. After computing the MFP solution by the proposed algorithm, exhaustive data flow analysis is performed independently using the round robin iterative method. The two solutions thus obtained are compared. If found identical, the process of incremental analysis is repeated by retaining the changes in the next node.

A test suite of 16 scientific FORTRAN programs was used for the purpose of analysis. These programs are from the same suite used for testing the generic exhaustive data flow analysis algorithm (proposed in chapter 4) and for making width measurements. The empirical results are contained in appendix D. Here, we record the following points concerning the implementation.

1. Procedures mentioned in figures 10.1 and 10.2 are actually implemented as inline computation since $\sqcap$ is known.

2. In practice there is no need to remember TR_0. As noted in section 10.2, TR_0 computation is required mainly to find out the presence of a *lower* function influencing a property.

   (a) In the case of available expressions analysis, the functions influencing AVIN_i^l can never be *lower*. AVOUT_i^l has only one function influencing it which is f_i^f. When a property at $out(i)$ is being included in $\mathcal{GCT}(p)$, there is no other function influencing it. Hence, the condition that there should be no *lower* function influencing it, is vacuously satisfied.

   (b) In MRA, the functions influencing PPOUT_i^l can never be *lower*. PPIN_i^l is influenced by two kinds of functions :

      - $g^f_{(j,i)}$, $j \in pred(i)$: This can never be *lower*.
      - f_i^b : This may be *lower*. As mentioned in section 9.4.1, the influence of PAVIN_i^l also is included in f_i^b.

      When a property is being included in $\mathcal{GCT}(p)$, the presence of a *lower* function is found out by the following computation.

$$props\_inTR_0 \quad = \quad \neg\text{ANTLOC}_i \cdot \neg\text{TRANSP}_i \; + \; \neg\text{PAVIN}_i$$

3. The local change computation refers to the old influence of the modified bit function (i.e. $old\,(\mathcal{B}_h^m(p'))$ for the purpose of computing $\mathcal{B2T}$. In the case of unidirectional data flow

problems, the edge flow functions are identity functions and only node flow functions can change. Since there is only one node flow function per node in unidirectional data flow problems, the old influence of such a function is same as the old value of the property that it influences. Thus, the implementation for available expressions analysis refers to $old(\text{AVOUT}_i)$ rather than to $old(\text{COMP}_i) + old(\text{TRANSP}_i) \cdot old(\text{AVIN}_i)$.

This must be contrasted with MRA where the old value of PPIN_i is not necessarily the same as the old influence of f_i^b.

4. Even though the definitions of $\mathcal{GCT}(p)$ and $\mathcal{NCT}(p)$ refer to $old(p'')$ for a property p'', in practice, we refer to current value of p'' where the old value of p'' is copied to the current value before the analysis begins. This allows us to use the same *propagate_bot_influence* procedure for $\mathcal{GCB}(p)$ and $\mathcal{NCT}(p)$ construction. This also helps in processing the multiple function changes : If a property p'' is in $\mathcal{GCT}(p)$ and $\mathcal{GCT}(p')$ for two properties p and p', then p'' is processed only once rather than twice since p'' is made TOP while constructing $\mathcal{GCT}(p)$ and can be safely ignored later while constructing $\mathcal{GCT}(p')$.

5. In the case of MRA, as noted in section 9.4.1, multiple bit vector functions may change due to changes in a single node. Thus we need to compute $\mathcal{T}2\mathcal{B}$ and $\mathcal{B}2\mathcal{T}$ sets for each changed function.

   (a) It is erroneous to process the changed functions separately. All $\mathcal{B}2\mathcal{T}$ and $\mathcal{T}2\mathcal{B}$ sets must be computed before finding out the global change. This is so since if we decide to process one function change fully, we are effectively ignoring the change in chains due to other functions which have changed.

   (b) In practice, a flow function is identified by the program points it is associated with. Let there be a self-loop around node i. Then, $\mathcal{B}2\mathcal{T}$ sets for f_i^b and $g_{(i,i)}^f$ are effectively referring to the properties at the same program point *viz.* $in(i)$. The two $\mathcal{B}2\mathcal{T}$ sets and the two $\mathcal{T}2\mathcal{B}$ are merged to give one $\mathcal{B}2\mathcal{T}$ set and one $\mathcal{T}2\mathcal{B}$ set per program point. A property is included in the resulting $\mathcal{B}2\mathcal{T}$set only if it is in either of the $\mathcal{B}2\mathcal{T}$ sets and is not in any of the $\mathcal{T}2\mathcal{B}$ sets.

10.7 Looking Back

The algorithm proposed in this chapter is a straight-forward iterative implementation of the functional model of incremental data flow analysis. In order to implement this algorithm, one does not require any detailed understanding of the data flows involved in the problem being solved. In fact our experience has been to the contrary. When we started implementing this algorithm, we implicitly made several assumptions mainly due to the familiarity of the data flows involved. However, we could get it working only after we carefully avoided all deviations from the algorithm. The fact that even small deviations proved erroneous, vindicates the belief that the whole process of incremental data flow analysis is a rather complicated process to visualise. On the other hand, a straight-forward implementation of the algorithm giving correct results can be achieved quickly (even for new data flow problems). This fact high-lights the strength and the generality of the algorithm. It would not be inappropriate to mention though, that the algorithm (being a straight-forward implementation of the functional model) is incidental; the real strength lies in the model. Any technique that correctly implements the definitions in the model is guaranteed to inherit this strength.

Chapter 11

Performing Incremental Data Flow Analysis : The Wordwise Approach

I raised the bucket to his lips. He drank, his eyes closed. It was as sweet as some special festival treat. This water was indeed a different thing from ordinary nourishment. Its sweetness was born of the walk under the stars, the song of the pulley, the effort of my arms. It was good for the heart, like a present.

The bitwise algorithm developed in chapter 10 is a faithful implementation of the functional model of data flow analysis and as such inherits the advantages mentioned in section 10.5. However, from a practical viewpoint it is preferable to be able to process several bits together. In this chapter we develop the wordwise approach in which all bits in a word are processed simultaneously. Like the previous chapter, we restrict ourselves to changes in only one bit vector function.

Section 11.1 investigates the issues which need to be addressed when many bits at a program point are processed simultaneously. Section 11.2 presents the algorithm while section 11.3 presents four variants of the algorithm. Section 11.4 analyses the complexity of the algorithm, while section 11.5 discusses some issues concerning the implementation.

11.1 Issues in Wordwise Incremental Data Flow Analysis

Consider the problem of available expressions analysis over the program flow graph in Figure 11.1. Let the properties at a program point be denoted by a tuple. In this case, since we have only two expressions, $\text{IN}_i = < \text{IN}_i^1, \text{IN}_i^2 >$ and $\text{OUT}_i = < \text{OUT}_i^1, \text{OUT}_i^2 >$. The MFP solution of this instance is $\text{IN}_i = < \mathbf{F}, \mathbf{F} >$ and $\text{OUT}_i = < \mathbf{F}, \mathbf{F} >$ for each node i.

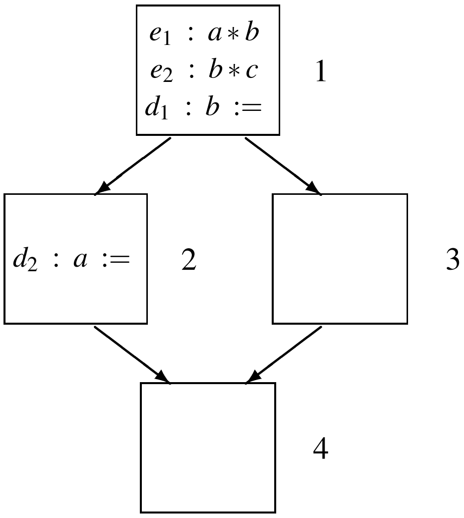


Figure 11.1: Motivating example for wordwise approach.

Let the assignment to b in node 1 be deleted. Both e_1 and e_2 now become available at the exit of node 1. Thus, $\text{OUT}_1 = <\mathbf{T}, \mathbf{T}>$. Since available expressions analysis is an intersection problem, TOP is **T** and BOT is **F**. Thus the above change is a case of BOT-to-TOPchange. We discuss in details, the steps involved in incorporating the effect of this change over the rest of the graph. Important observations which form the basis of the wordwise approach are highlighted in the following discussion.

Constructing the affected region

Two information flow paths (i.e. 1,2,4 and 1,3,4) need to be traversed for the purpose.

1. *Traversing the path 1,2,4*

   IN_2 is made $<\mathbf{T}, \mathbf{T}>$. However, OUT_2 cannot become $<\mathbf{T}, \mathbf{T}>$; OUT_2^1 remains **F** due to the presence an assignment to a in node 2. Thus, OUT_2 become $<\mathbf{F}, \mathbf{T}>$ and hence both IN_4 and OUT_4 become $<\mathbf{F}, \mathbf{T}>$. In other words, visit to node 4 is essential even though OUT_2^1 is **F**.

   At this stage, the affected region consists of the following program points : $out(1)$, $in(2)$, $out(2)$, $in(4)$, and $out(4)$.

2. *Traversing the path 1,3,4*

   Both IN_3 and OUT_3 are made $<\mathbf{T}, \mathbf{T}>$. When node 4 is reached, IN_4^1 is found to be **F**. Since OUT_3^1 has been made **T**, IN_4 is made $<\mathbf{T}, \mathbf{T}>$. Hence OUT_4 also becomes $<\mathbf{T}, \mathbf{T}>$.

Observation 11.1 : *Several properties are considered simultaneously for global change for constructing the affected region. While traversing a given information flow path, the effect of some properties becoming TOP may terminate at some program point — the information flow path may still have to be traversed further to incorporate the effect of some other properties becoming TOP.* □

Observation 11.2 : *A node may be visited several times due to the existence of multiple information flow paths reaching the node and the value of a property, which was not changed in an earlier traversal* (*viz.* IN_4^1 *while traversing the path 1,2,4*), *may have to be changed during a later traversal* (*along path 1,3,4 in this case*). □

This must be contrasted with the situation in the bitwise algorithm : While processing e_1, the traversal of path 1,2,4 would terminate at $out(2)$. Thus before path 1,3,4 is traversed, the affected region consists of $out(1)$, $in(2)$. Neither $out(2)$ nor $in(4)$ is in the affected region at this stage. Later when path 1,3,4 is traversed, $in(4)$ is visited for the first time and IN_4^1 is made **T**.

Finding the boundary and propagating the effect of its BOT properties

Since $in(1)$ is not in the affected region, $out(1)$ is a boundary point. Recomputation of OUT_1 results in $<\mathbf{T},\mathbf{T}>$. This is same as the previous value and no propagation is required. There is no other boundary point in the region. The process terminates and we get the following values :

Node	Expression e_l			
	e_1		e_2	
	IN_i^1	OUT_i^1	IN_i^2	OUT_i^2
1	**F**	**T**	**F**	**T**
2	**T**	**F**	**T**	**T**
3	**T**	**T**	**T**	**T**
4	**T**	**T**	**T**	**T**

Note that this is not a fixed point solution since OUT_2^1 and IN_4^1 have inconsistent values — OUT_2^1 is **F** while IN_4^1 is **T** in spite of 2 being a predecessor of 4.

Why do we get inconsistent values?

This happens because the values computed at multi-entry nodes may be different along different paths. A property may remain BOT (IN_4^1 in this case) while traversing one path (path

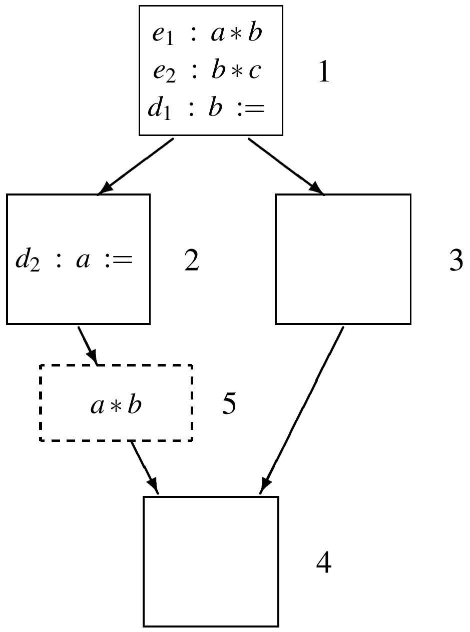


Figure 11.2: Another program flow graph for wordwise approach.

1,2,4 in this case) and it may be made TOP while traversing some other path (path 1,3,4 in this case).

A naive but erroneous solution

One may be tempted to conclude that IN_4^1 should not be made **T** when path 1,3,4 is traversed. Though it may give a correct result in this particular case, it is wrong to do so for the following reasons :

1. If path 1,3,4 is traversed first, IN_4 is made $< \mathbf{T}, \mathbf{T} >$ the first time node 4 is visited. Unless the path 1,2,4 is traversed, it is not possible to infer that IN_4^1 should not be made **T** when path 1,3,4 is traversed.

2. Consider the program flow graph in Figure 11.2 which is essentially the same as the flow graph of Figure 11.1 but for the existence of node 5 between nodes 2 and 4. The presence of expression $a*b$ in node 5 has the following consequences :

   When path 1,2,5,4 is traversed, IN_2 is made $< \mathbf{T}, \mathbf{T} >$ and OUT_2 is made $< \mathbf{F}, \mathbf{T} >$. At this stage we note the fact that the effect of OUT_1^1 becoming **T** has stopped at $out(2)$ and we need not consider the corresponding properties along the remaining path for a change. Thus we make $\mathrm{IN}_5 = <\mathbf{F},\mathbf{T}>$. OUT_5, which originally was $<\mathbf{T},\mathbf{F}>$, is made $<\mathbf{T},\mathbf{T}>$. IN_4 which originally was $<\mathbf{F},\mathbf{F}>$ is made $<\mathbf{F},\mathbf{T}>$.

Later when path 1,3,4 is traversed, IN_4^1 is made **T** as in the previous example. In this case though, it turns out to be correct. Not making it **T** would have been wrong since both OUT_5^1 and OUT_3^1 are **T** — OUT_5^1 was originally **T** and OUT_3^1 has been made **T** during the process of incremental analysis.

The first argument above shows that it may not be possible to identify a situation when a property should not be made **T** and the second argument shows that not making it **T** may be wrong in some cases. Thus, we conclude that the suggestion made by observation 11.2 is a valid suggestion and that the problem lies elsewhere.

Computing consistent values

Note that we get inconsistent values because we may make a property TOP when it should not be made TOP. This must be contrasted with the other possible cause of inconsistency *viz.* making a property BOT when it should have been TOP. The latter doesn't arise in our case and we err on the conservative side in that some properties are re-initialised to TOP. Thus if we are in a position to identify such properties, it would be possible to find out if their final values are actually BOT. This is same as identifying a boundary point and propagating the effect of its BOT properties, except that the program point may not be a boundary point of the affected region, topologically.[1]

Observation 11.3 : *The program point which is visited more than once while constructing the affected region must be treated as a boundary point even though, topologically, all its neighbours are in the affected region. Note that this needs to be done only for those program points u for which the condition* $\mid \mathcal{N}^{-1}(u) \mid > 1$ *holds.* □

In the above example, only the program points with multiple entries need to be remembered for propagation.

Thus, a revisit to $in(4)$ must be remembered while a revisit to $in(2)$ can be safely ignored since such a revisit cannot happen unless $in(1)$ is revisited.

With this modification, IN_4 will be recalculated. It becomes $<\mathbf{F},\mathbf{T}>$ in the case of program flow graph in Figure 11.1 and hence OUT_4 will also be made $<\mathbf{F},\mathbf{T}>$. In the case of program flow graph in Figure 11.2, IN_4 remains $<\mathbf{T},\mathbf{T}>$ and no propagation is required. The resulting solutions are MFP solutions.

[1]i.e. all program points in $\mathcal{N}^{-1}(u)$ may be in the affected region.

We conclude this section by developing one more insight. Let the program point for which $\mathcal{B}2\mathcal{T}$ is computed and the affected region is constructed ($out(1)$ in the above example), be called the *header*.

Observation 11.4 : *The header* (u) *is considered a boundary point regardless of whether it is revisited and/or whether* $\mid \mathcal{N}^{-1}(u) \mid > 1$. $\Box$

Let h^m be $h^m(u', u)$. Consider a p in $\mathcal{B}2\mathcal{T}$ represented by $X^i(u)$. Let the corresponding property at u' be p'. Thus p' is represented by $X^i(u')$. After constructing the affected region, following possibilities exist for the status of u' and the value $X^i(u')$.

1. u' is in the affected region.

   This implies that some $X^j(u')$ corresponding to a property $X^j(u) \in \mathcal{B}2\mathcal{T}$, has been made TOP. Two possibilities arise :

   (a) $j = i$, i.e. $X^i(u')$ is made TOP.

   In this case $X^i(u')$ cannot change the value of $X^i(u)$ ($X^i(u)$ has already been made TOP since it is in $\mathcal{B}2\mathcal{T}$).

   (b) $j \neq i$, i.e. $X^i(u')$ remains unchanged.

   In this case $X^i(u') = old(X^i(u'))$. It may be either TOP or BOT.

   i. $X^i(u')$ is TOP.
   In this case too, $X^i(u')$ cannot change the value of $X^i(u)$. ($X^i(u)$ has already been made TOP since it is in $\mathcal{B}2\mathcal{T}$).

   ii. $X^i(u')$ is BOT.
   $X^i(u')$ may change $X^i(u)$ to BOT. We need to incorporate the influence of $X^i(u')$ and hence u must be treated as a boundary point though u' is in the region.

2. u' is not in the affected region.

   In this case u is anyway a boundary point.

Note that situation in step (1b) cannot arise in the bitwise approach. While processing p, if $X^i(u')$ is not made TOP, there is no way u' can be in the affected region (and vice-versa). Hence there is no need to include u in the set of boundary points separately in bitwise approach.

11.2 A Wordwise Algorithm

11.2.1 A Functional Model for Wordwise Analysis

From the discussion in the previous section, it is clear that the local change computation remains same.

The sets used by the functional model for computing the global change (i.e. $\mathcal{GCB}(p)$, $\mathcal{GCT}(p)$, $\mathcal{NCT}(p)$) are defined in terms of a single property.[2] For a wordwise implementation of the algorithm, we need to redefine these sets. This extension is really straight-forward. We use the notation $\mathcal{T2B}(w)$, $\mathcal{B2T}(w)$, $\mathcal{GCB}(w)$, $\mathcal{GCT}(w)$, $\mathcal{NCT}(w)$ to denote the sets for a word w. Thus, $\mathcal{GCB}(w)$ contains the properties in word w which have changed from TOP to BOT at various program points while $\mathcal{GCT}(w)$ contains the properties in w which may change to TOP.

As in the previous case, the global change is defined by $\mathcal{GC} = <\mathcal{B2T}\star, \mathcal{T2B}\star>$ where $\mathcal{B2T}\star$ and $\mathcal{T2B}\star$ are defined as :

$$\begin{aligned}
\mathcal{GCB}(w) &= \mathcal{T2B}(w) \cup \{p' \mid \mathrm{CH}(p,p') \neq \emptyset,\ p \in \mathcal{T2B}(w), old(p') = \mathrm{TOP}\} && (11.1)\\
\mathcal{T2B}\star &= \bigcup_{w} \mathcal{GCB}(w) && (11.2)\\
\mathcal{GCT}(w) &= \mathcal{B2T}(w) \cup \{p' \mid \mathrm{CH}(p,p') \neq \emptyset,\ p \in \mathcal{B2T}(w), old(p') = \mathrm{BOT}\} && (11.3)\\
\mathcal{NCT}(w) &= \{p' \mid p' \in \mathcal{GCT}(w) \text{ and } \exists p'' \in TR_0 \text{ such that } \mathrm{CH}(p'',p') \neq \emptyset\} && (11.4)\\
\mathcal{B2T}\star &= \bigcup_{w} (\mathcal{GCT}(w) - \mathcal{NCT}(w)) && (11.5)
\end{aligned}$$

11.2.2 A Wordwise Incremental Data Flow Analysis Algorithm

Given the changes in one bit vector function $h^m(u', u)$, following steps are carried out by the wordwise algorithm for each word w :

1. Identify the bit functions that have changed in w.
2. Compute the new TR_0.
3. Compute the local change $\mathcal{LC} = <\mathcal{B2T}, \mathcal{T2B}>$.
4. Compute the global change $\mathcal{GC} = <\mathcal{B2T}\star, \mathcal{T2B}\star>$.
   (a) Compute $\mathcal{GCB}(w)$ by propagating the effect of the properties in $\mathcal{T2B}(w)$.

[2] The sets used for the local change (i.e. $\mathcal{T2B}$ and $\mathcal{B2T}$), however, are not defined for each property separately.

procedure $incremental\_dfa(h^m, u', u)$
$\{$ $\quad$ /$\star$ $X(u) \leftarrow h^m(X(u'))$ and $h^m(X) = C_1 + C_2 \cdot X$ $\star$/
$\quad\quad$ $H^m := (old\,(C_1) \cdot \neg C_1 + \neg old\,(C_1) \cdot C_1)$
$\quad\quad$ $compute\_TR_0(h^m, H^m, u)$
$\quad\quad$ $find\_local\_change(h^m, H^m, u', u)$
$\quad\quad$ **for** each word w
$\quad\quad$ $\{$ $\quad$ $propagate\_bot\_influence(w, u, \mathcal{T2B}(w))$
$\quad\quad\quad\quad$ $\mathcal{AR}_w := \mathcal{BR}_w := \emptyset$
$\quad\quad\quad\quad$ $find\_global\_change\_to\_top(w, u, \mathcal{B2T}(w))$
$\quad\quad\quad\quad$ $propagate\_boundary\_effect(w, u)$
$\quad\quad$ $\}$
$\}$

Figure 11.3: A generic algorithm for incremental data flow analysis

procedure $propagate\_bot\_influence(w, u, bot\_influence)$
$\{$ $\quad$ **for** each $u' \in \mathcal{N}(u)$ $\quad\quad$ /$\star$ Let $h^w(u', u)$ be $C_1^w + C_2^w \cdot X$ $\star$/
$\quad\quad$ $\{$ $\quad$ $change := bot\_influence \cdot propagate\_functions(h^w(u', u))$
$\quad\quad\quad\quad$ $change := retain\_TOP\_bits(change, X^w(u'))$
$\quad\quad\quad\quad$ **if** ($change$ is not $\overline{\mathrm{F}}$) **then**
$\quad\quad\quad\quad$ $\{$ $\quad$ $X^w(u') := make\_bits\_BOT(X^w(u'), change)$
$\quad\quad\quad\quad\quad\quad$ $propagate\_bot\_influence(w, u', change)$
$\quad\quad\quad\quad$ $\}$
$\quad\quad$ $\}$
$\}$

Figure 11.4: Computing $\mathcal{GCB}(w)$

(b) Compute $\mathcal{GCT}(w)$ by constructing the affected region $\mathcal{AR}_w$. The program points which are revisited and are neighbours of more than one program point (i.e. $\mid \mathcal{N}^{-1}(u') \mid < 1$), are added to $\mathcal{BR}_w$ which denotes the boundary of the region.

(c) Identify those program points in $\mathcal{AR}_w$ which are neighbours of some program point which is not in $\mathcal{AR}_w$. Include such points in $\mathcal{BR}_w$. Add the header u to $\mathcal{BR}_w$.

(d) Compute the values of the properties in word w for each program point in $\mathcal{BR}_w$. Propagate the effect of BOT properties (if any) to the program points in $\mathcal{AR}_w$.

```
procedure find_global_change_to_top(w, u, top_influence)
{   if (u is in AR_w) then
    {   if (| N^-1(u) | > 1) then
            BR_w := BR_w + {u}
    }
    else AR_w := AR_w + {u}
    for each u' ∈ N(u)            /* Let h^w(u',u) be C_1^w + C_2^w · X */
    {   change := top_influence · propagate_functions(h^w(u',u))
        change := retain_BOT_bits(change, X^w(u'))
        if (change is not F̄) then
        {   X^w(u') := make_bits_TOP(X^w(u'), change)
            find_global_change_to_top(w, u', change)
        }
    }
}
```

Figure 11.5: Computing $\mathcal{AR}_w$

Note that we do not spell out the details of the TR_0 computation and the local change computation in terms of words and retain these computations exactly as in the bitwise algorithm.[3]

11.3 Four Variants of the Wordwise Algorithm

The local change computation, being a fixed sequence of boolean operations, offers little scope for any variations. Global change computation, on the other hand, can be performed in many different ways. In this section, we suggest two heuristics which lead to four variants of the wordwise algorithm.

11.3.1 Revisits to a Program Point

We refer to the program flow graphs in figures 11.1 and 11.2 and make some useful observations.

[3] In practice, even in the bitwise algorithm, these computations will have to be performed on words since computations on entire bit vectors are not possible if the size of the bit vector exceeds the size of the machine word.

```
procedure propagate_boundary_effect(w,u) /⋆ u is the header. ⋆/
{   for each u′ ∈ ℬℛ_w ∪ {u}
    {   temp := ⊤
        for each u″ ∈ 𝒩^{-1}(u′)
            temp := temp ⊓ [h^w(u″,u′)](X^w(u″))
        old_X := X^w(u′)
        X^w(u′) := temp
        change_to_bot := find_BOT_change (old_X, X^w(u′))
        if (change_to_bot is not F̄) then
            propagate_bot_influence(w,u′,change_to_bot)
    }
}
procedure find_BOT_change(x,y)
{   if (⊓ is Π) then
        return x · ¬y
    else if (⊓ is Σ) then
        return ¬x · y
}
```

Figure 11.6: Computing $\mathcal{NCT}(w)$

Observation 11.5 : *If a property is not included in $\mathcal{GCT}(w)$ on the first visit to a program point and is a neighbour of a BOT property, then its final value can never be TOP regardless of its inclusion in $\mathcal{GCT}(w)$ on any subsequent visit.* □

Example 11.1 : In Figure 11.1, IN_4^1 is not included in $\mathcal{GCT}(w)$ on the first visit to the program point $in(4)$. IN_4^1 is a neighbour of OUT_2^1 which is BOT. Thus, though IN_4^1 is included in $\mathcal{GCT}(w)$ on a subsequent visit along path 1,3,4, its final value can never be TOP due to the influence of OUT_2^1. □

This observation can be used to decide which properties must necessarily be BOT. The following observation identifies the properties which may be TOP.

Observation 11.6 : *If a property is not included in $\mathcal{GCT}(w)$ but its value is TOP, and if the old values of the corresponding properties on the following portion of the information flow path are BOT, the final values of these corresponding properties may be TOP and hence they could be included in $\mathcal{GCT}(w)$ unless it is known that they are influenced by lower functions.* □

Example 11.2 : In Figure 11.2, OUT_5^1 is not included in $\mathcal{GCT}(w)$ but its value is TOP. The old values of IN_4^1 and OUT_4^1 are BOT. Since these properties are not influenced by a *lower* function, the final values of these properties are be TOP and hence they could be included in $\mathcal{GCT}(w)$ during the first traversal. □

If such properties are included in $\mathcal{GCT}(w)$ on the first visit, all subsequent visits to that program point can be safely avoided since :

1. if a property is not included in $\mathcal{GCT}(w)$ on the first visit then it is not the neighbour of a TOP property and hence would be BOT. Thus, it need not be included in $\mathcal{GCT}(w)$ on any subsequent visit.

2. Assume that a property p'' associated with a program point u'' is included in $\mathcal{GCT}(w)$ on the first visit to u'' following observation 11.6. Assume further that u'' is revisited and p'' is found to be a neighbour of a BOT property. Since a revisited program point is included in $\mathcal{BR}_w$, u'' would be treated as a boundary point. When the properties at u'' are recomputed, p'' would turn out to be BOT. Thus, procedure *propagate_boundary_effect* will remove such properties by including them in $\mathcal{NCT}(w)$.

So we can avoid revisits to a node if we make the modification summarised by the following observation :

Observation 11.7 : *While computing the change set* (*i.e. the set of properties which may change*) *on line number 8 and 9 of procedure find_global_change_to_top* (*Figure 11.5*)*, instead of retaining only those properties for which the flow function is propagate* (*line number 8*) *and which are TOP* (*line number 9*)*, retain all properties* (*corresponding to the properties in* $\mathcal{B2T}(w)$) *regardless of their values and/or flow functions except the properties for which the flow function is lower.* □

Intuitively, if a property is influenced by a *lower* function, it can never be TOP. Barring such properties, all properties corresponding to the properties in $\mathcal{B2T}$(w) can be made TOP and hence, no additional property will have to be made TOP on a subsequent visit to a program point.

11.3.2 Size of $\mathcal{GCT}(w)$

The modification suggested above may have adverse effect on the size of $\mathcal{GCT}(w)$. The *change* set computed in the original algorithm reduces faster since the properties influenced by both *raise* and *lower* functions are removed from it. With this modification, effectively, the properties which are influenced by *raise* functions (and which may have been removed from the *change* set), are included in the *change* set again.

On the other hand, if we reach a node along a path which has BOT influence on a property then it is guaranteed to be excluded from $\mathcal{GCT}(w)$. Since the algorithm does not have enough information to decide a-priori as to which path should be selected, the size of $\mathcal{GCT}(w)$ could be small in some case and could be large in some other case. This dependence on the path chosen could be reduced if instead of traversing one path fully, the paths are traversed in an interleaved fashion, i.e. instead of propagating the TOP effect as far as possible along one path, it could be first propagated to all neighbouring program points and only then to their neighbours.

Thus, our heuristic for reducing the size of $\mathcal{GCT}(w)$ on an average is summarised by the following observation :

Observation 11.8 : *Instead of visiting the nodes in the depth first order in procedure find_global_change_to_top, the calls could be made in the breadth first order.* □

11.3.3 Incorporating the Heuristics

Since the two heuristics (i.e. avoiding revisits and using breadth-first traversal over the graph) are orthogonal, we get four possible combinations :

1. *Revisits to program points with depth first traversal over the graph.*

   This is the original algorithm as suggested in section 11.2.

2. *No revisits with depth first traversal over the graph.*

   In this case, lines 8 and 9 of procedure *find_global_change_to_top* are replaced by the following line :

   $$\boxed{change := \mathcal{B2T}(w) \cdot \neg\, lower\_functions(h^w(u', u))}$$

   Similarly, line 10 is replaced by :

   $$\boxed{\textbf{if } ((change \text{ is not } \overline{\mathrm{F}}) \textbf{ and } (u' \notin \mathcal{AR}_w)) \textbf{ then}}$$

3. *Revisits to program points with breadth first traversal over the graph.*

In this case instead of making recursive call directly on line 12 of procedure *find_global_change_to_top*, program point u' is added to a list and the next program point is selected from the list.[4] Essentially, we inherit all the advantages of the breadth first traversal.

4. *No revisits with breadth first traversal over the graph.*

   This variant is basically a combination of the changes suggested in steps 2 and 3.

11.4 Complexity of Incremental Data Flow Analysis

It is already an accepted fact that the worst case performance of incremental algorithms can be much worse than the exhaustive algorithms [53]. In our context, this can be very simply concluded by the following argument: Let there be some change in a word w so that $\mathcal{B}2\mathcal{T}(w) \neq \emptyset$ but $\mathcal{B}2\mathcal{T}\star = \emptyset$. It is clear in such a case that $\mathcal{GCT}(w) = \mathcal{NCT}(w)$[5] and if the entire graph is included in $\mathcal{AR}_w$, the entire graph will have to revisited to compute $\mathcal{NCT}(w)$. Thus, the work done by the incremental algorithm is at least twice as much as the work done by the exhaustive algorithm for the same change.

In this section we show that for one change in a word, the order of the work done by the incremental algorithm in the worst case is same as the order of the work done by the exhaustive algorithm. In practice though, the amount of work required in the worst case may be much more than the amount of work required for exhaustive algorithm.

We estimate the complexity of the first variant of the wordwise algorithm. The order of the work remains same for all the four variants and even for the bitwise algorithm except that in the case of bitwise algorithm, a multiplying factor which is equal to the number of bits in a word, may be involved. The estimate is subject to the assumption that the number of edges of a node is be bounded by a constant. This is justified since, the total number of edges in practical flow graphs is linear in the number of nodes.

The wordwise algorithm consists of the following steps (Figure 11.3).

1. TR_0 computation.

2. Local change computation.

3. Propagating the influence of properties in $\mathcal{T}2\mathcal{B}$.

[4]Thus the selection of a node being made on line 7 will have to be changed appropriately.

[5]Note that $\mathcal{NCT}(w)$ is a subset of $\mathcal{GCT}(w)$, hence the equality in the case of empty set difference.

4. Computing $\mathcal{AR}_w$ (and $\mathcal{BR}_w$).
5. Computing the values of properties in $\mathcal{BR}_w$ and propagating the effect of BOT properties.

We consider the work done in each step separately. The order of the algorithm will be the maximum work done in any step.

1. *TR_0 computation.*

   For any word this work is bounded by the number of functions influencing the properties at u (alternatively, by $\mid \mathcal{N}^{-1}(u) \mid$) which in turn is bounded by the sum of the number of in-edges and out-edges of a node. Since the number of edges of a node is bounded by a constant, the amount of work is constant.

2. *Local change computation.*

   This is just a series of boolean operations and the amount of work involved is constant.

3. *Propagating the influence of properties in $\mathcal{T2B}$.*

   For a given word, the order of this work is same as the *propagation* phase of the exhaustive algorithm for a word which is linear in the number of nodes of the graph.

4. *Computing $\mathcal{AR}_w$.*

   This is also linear in the number of nodes since at most all program points may be included in $\mathcal{AR}_w$. The possibility of several visits to a program point may imply more work but this work is bounded by the number of bits in a word since every revisit implies that some additional property must be included in $\mathcal{GCT}(w)$ and hence a program point would not be revisited after all its properties are included in $\mathcal{GCT}(w)$. Since the number of bits in a word is constant, the order of work is linear.

5. *Computing the values of properties in $\mathcal{BR}_w$ and propagating the effect of BOT properties.*

   The work involved is similar to the work done in step (3) above except that the values of the properties at the boundary points are also computed. Since the number of functions influencing the properties at a program point is bounded by a constant, the amount of work required for recomputing the properties is bounded by the number of program points in $\mathcal{BR}_w$. In the worst case, all program points may be boundary points. Thus, the total work done in this step is linear in the size of the graph.

We conclude from the above discussion, that the order of the work involved in performing incremental data flow analysis is same as the order of the work involved in exhaustive analysis, though practically, the work may be much more in the worst case. Even though the discussion is restricted to one change only, it has already been argued in section 9.4.3 that multiple changes too can be handled in the same run of the algorithm. The possibility of multiple function changes affects only the local change computation and since the complexity is dominated by global change computation, the order of work remains same. Similar remarks hold for structural changes.

11.5 Implementation Notes

The wordwise algorithm was implemented in the same environment in which the bitwise algorithm was implemented (see section 10.6). The motivation behind this implementation was to demonstrate the correctness of the algorithm practically. This was more important since this is the first algorithm for incremental data flow analysis of bidirectional data flows.

The experimental results are contained in appendix E. We record the following points concerning the implementation :

1. We have taken measurements only in terms of bit vector operations. The current implementation of the algorithm is bound to be much slower in terms of execution time than the exhaustive round robin method since it involves many overheads (in terms of recursion, function calls, list management etc.). This situation is much the same as it was in the case of initial implementations of the generic (exhaustive) algorithm for MRA (see chapter 5). Even there the algorithm was faster in terms of bit vector operations but was slower in terms of execution time. A careful implementation may well lead to gains in terms of execution time as it did in the case of exhaustive analysis.

   The improvements can be made on four levels :

   (a) Elimination of function calls by in-line expansion.

   (b) Implementing the lists using arrays with the indices serving as pointers.

   (c) The third level optimisations concerns the heuristics for list organisation. More specifically, selection of the next node for further processing is a rather crucial decision.

   (d) The bit vector operations performed can be reduced by minimising the boolean expressions appropriately.

2. It is readily seen from Table E.1 that the proposed algorithm for incremental data flow analysis requires much less work than the round robin exhaustive algorithm so much so that we talk about improvement by a factor rather than in terms of percentage.

   The speed up factor for available expressions analysis is substantially smaller than the factor for MRA. This vindicates the fact that the information flow paths in bidirectional data flow problems are much more complex than the information flow paths for unidirectional problems.

3. The fact that the algorithm computes the MFP solution for MRA, confirms the claim that multiple function changes can be handled in wordwise algorithm too, without any special requirements.

4. The performance of the four variants seems to be comparable.[6]

   (a) The variants using breadth first traversal seem to be marginally better than the ones involving depth first traversal for unidirectional data flow problems. It seems to be the other way round for bidirectional data flow problems.

   (b) The speed up factor seems to go down if revisits to a node are avoided. This may be because of the fact that fewer bits are removed from the *change* set (see section 11.3.2) and hence a larger $\mathcal{GCT}(w)$ set is computed.

[6]Note that the notions of "depth first" and "breadth first" traversals are used to traverse the information flow paths rather than graph theoretic paths. In the case of MRA, "breadth first" implies visiting all neighbours of a program point before visiting a "neighbour of a neighbour".

Chapter 12

Correctness of Incremental Data Flow Analysis

Just so, you might say to them : "The proof that the little prince existed is that he was charming, that he laughed and that he was looking for a sheep. If anybody wants a sheep, that is a proof that he exists." ... They would shrug their shoulders, and treat you like a child. But if you said to them : "The planet he came from is Asteroid B-612," then they would be convinced, and leave you in peace from their questions.

The process of incremental data flow analysis is a complicated process. In order to simplify the task of showing correctness, we make the following assumptions :

1. We assume that there are no structural changes.
2. We assume that only one bit vector function h^m has changed.
3. We assume that only one bit function $\mathcal{B}_h^m$ in h^m has changed.

These assumptions are simplifying assumptions rather than constraining in that the correctness proofs remain valid (in spirit, if not in letter) even if these assumptions do not hold. This is very easily justified for the first and the third assumption; we do that in the following paragraphs. Justification for the second assumption is far from obvious and we relegate the task to section 12.5.

It has already been shown in section 9.4.4 that structural changes can be modelled in terms of functions changes. Thus, once the correctness of the model with function changes is established, the correctness of the model for structural changes follows automatically.

A bit vector function consists of many independent bit functions (section 3.1.3). Thus, even if many bit functions change in h^m, they do not influence each other due to bit-independence. Hence we can assume, without any loss of generality, that only one bit function of h^m has changed. The correctness of the properties corresponding to unchanged bit functions follows automatically since their chains and their membership in TR_0 remains unchanged.

12.1 Defining Correctness

Recall that $old(S) = < old(\mathcal{VT}), old(\mathcal{VB}) >$ and $S = < \mathcal{VT}, \mathcal{VB} >$ are the MFP solutions for the instances $old(\mathbf{I})$ and $\mathbf{I}$ respectively, if :

$$\begin{aligned} old(S) \; &: \; \forall p,\; old(p) = \text{BOT } \textit{iff} \\ &\quad\; p \in old(TR_0) \text{ or } \exists\, p' \in old(TR_0) \text{ such that } old(\text{CH}(p', p)) \neq \emptyset \quad (12.1) \\ S \; &: \; \forall p,\; p = \text{BOT } \textit{iff } p \in TR_0 \text{ or } \exists\, p' \in TR_0 \text{ such that } \text{CH}(p', p) \neq \emptyset \quad (12.2) \end{aligned}$$

Showing correctness of the proposed model implies showing that if $old(S)$ is the MFP solution for $old(\mathbf{I})$, then S as defined by equation 9.14 is the MFP solution of $\mathbf{I}$ (i.e. S satisfies condition 8.2). This is achieved in three steps :

(i) Section 12.2 enumerates various cases that arise due to a change in a bit function. We use the partition $\pi_\beta(\Delta)$ (section 9.2.2.1) for this purpose.

(ii) Section 12.3 shows the correctness of TR_0 computation. We use the partition $\pi_\alpha(\Delta)$ (section 9.2.2.1) for this purpose.

(iii) Finally, section 12.4 shows that condition 12.2 is satisfied by every property in the MFP solution.

12.2 Influence of a Function Change

12.2.1 Local Influence

Consider a function $p \leftarrow \mathcal{B}_h^m(p')$. A change in $\mathcal{B}_h^m$ may influence the property p in three different ways.

1. $p \in \mathcal{T}2\mathcal{B}$: The new value of p is BOT.

2. $p \in \mathcal{B}2\mathcal{T}$: The new value of p *may be* TOP.

3. $p \notin (\mathcal{T}2\mathcal{B} \cup \mathcal{B}2\mathcal{T})$: The value of p does not change.

From observation 9.6 the following three factors, each having multiple possibilities, determine the exact influence :

(i) The function change $\partial\mathcal{B}_h^m$.

(ii) (a) For $\partial\mathcal{B}_h^m \in \sigma_\beta^1(\Delta)$: Old value of p.
(b) For $\partial\mathcal{B}_h^m \in \sigma_\beta^2(\Delta)$: Old influence of $\mathcal{B}_h^m(p')$.

(iii) (a) For $\partial\mathcal{B}_h^m \in \sigma_\beta^1(\Delta)$: Old value of p'.
(b) For $\partial\mathcal{B}_h^m \in \sigma_\beta^2(\Delta)$: Existence of a *lower* function in $\Omega(p)$.

The *lower* function has been abbreviated by l in the both the figures.

First factor has six possible values ($|\Delta| = 6$) while second and third factors have two values each. Of the 24 possible combinations,

- some may be ruled out (for instance, if there exists a lower function in $\Omega(p) - \mathcal{B}_h^m$, then the old value of p cannot be TOP),
- some may be irrelevant (*viz.* old value of p' is relevant only if $\mathcal{B}_h^m$ is *propagate*),
- some may have the same influence (*viz.* $\partial\mathcal{B}_h^m \equiv r\text{-}\!\!\triangleright l$ and $\partial\mathcal{B}_h^m \equiv p\text{-}\!\!\triangleright l$ have the same influence).

We enumerate the distinct cases in Figures 12.1 and 12.2. Together, the two figures cover the partition $\pi_\beta(\Delta)$. Each factor mentioned above, introduces branching in the tree, exhausting all possibilities for that factor.[1] This guarantees that we cover all the cases that may arise. In particular, the figures answer following two questions for each case :

(i) Does p belong to TR_0? (Is p in $\mathcal{TR}^-$, $\mathcal{TR}^+$?)

(ii) Is the value of p likely to change? (Is p in $\mathcal{T}2\mathcal{B}$, $\mathcal{B}2\mathcal{T}$?)

The *lower* function is abbreviated by l in the figure for convenience.

Figure 12.1 : $\partial\mathcal{B}_h^m \in \sigma_\beta^1(\Delta)$

The first branching occurs with the two partitions of $\sigma_\beta^1(\Delta)$ which are $\sigma_\alpha^1(\Delta)$ and $\{r\text{-}\!\!\triangleright p\}$ (observation 9.1). Left branch covers the cases when $p \in \mathcal{TR}^+$ while the right branch covers

[1]Different combinations having same influence are combined into one.

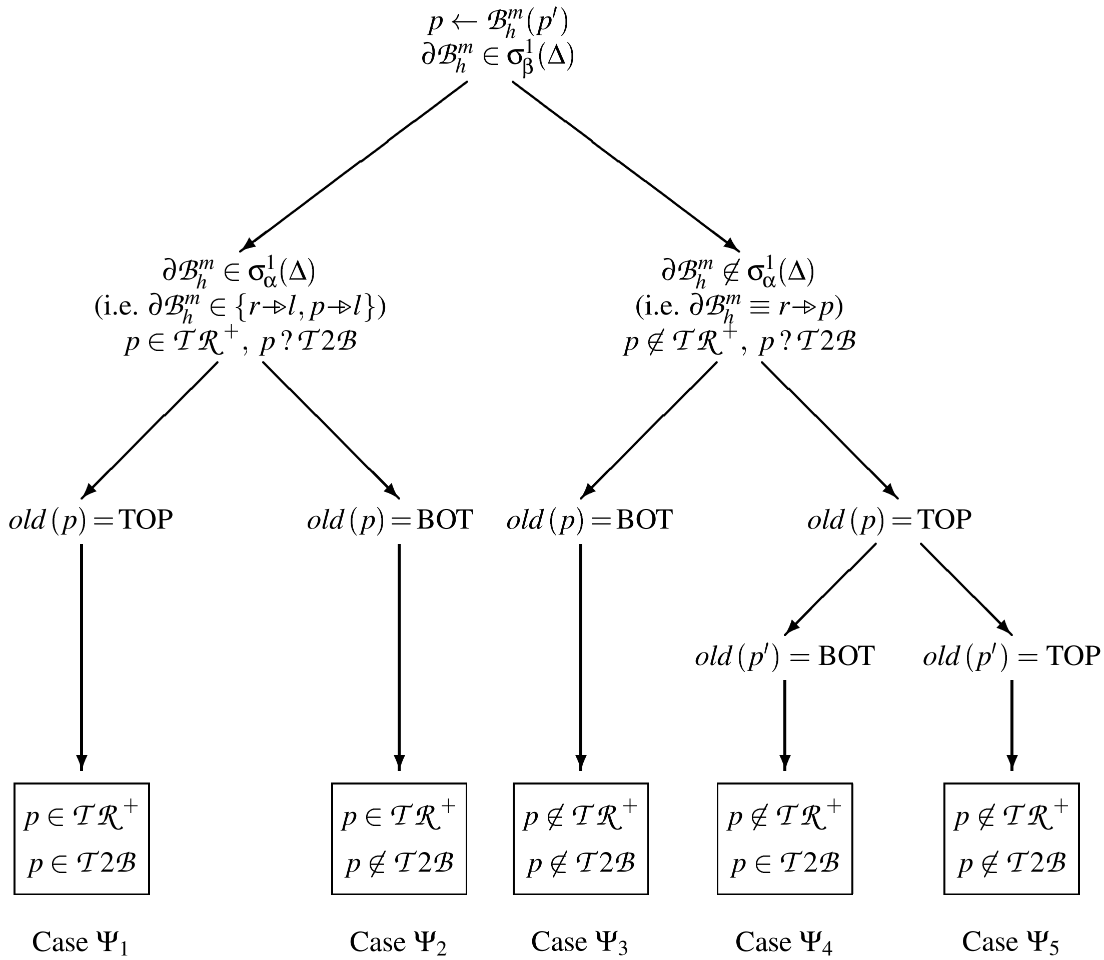


Figure 12.1: Various cases for $\partial\mathcal{B}_h^m \in \sigma_\beta^1(\Delta)$

the cases when $p \notin \mathcal{TR}^+$. However, it is not known at this level in the tree whether p is in $\mathcal{T2B}$; this is indicated by "$p\,?\,\mathcal{T2B}$".

The second branching occurs with the second factor, i.e. the old value of p. Since $old\,(p)$ is BOT in cases Ψ_2 and Ψ_3, $p \notin \mathcal{T2B}$. Since $old\,(p)$ is TOP in case Ψ_1, $p \in \mathcal{T2B}$. However, though $old\,(p)$ is TOP in cases Ψ_4 and Ψ_5, $old\,(\mathcal{B}_h^m)$ is *raise*, and $\mathcal{B}_h^m$ is *propagate*. Hence the membership of p in $\mathcal{T2B}$ depends on $old\,(p')$. This is the third factor which introduces the third branching which is restricted to cases Ψ_4 and Ψ_5 only.

Figure 12.2 : $\partial\mathcal{B}_h^m \in \sigma_\beta^2(\Delta)$

The first branching occurs with the two partitions of $\sigma_\beta^2(\Delta)$ which are $\sigma_\alpha^2(\Delta)$ and $\{p \mathrel{-\!\!\triangleright} r\}$ (observation 9.1). Right branch covers the cases when $p \notin \mathcal{TR}^-$. Note that the membership of p in $\mathcal{TR}^-$ is not known along the left branch since there may exist a *lower* function in $\Omega(p)$. The membership of p in $\mathcal{B2T}$ is not known at this level along either of the branches.

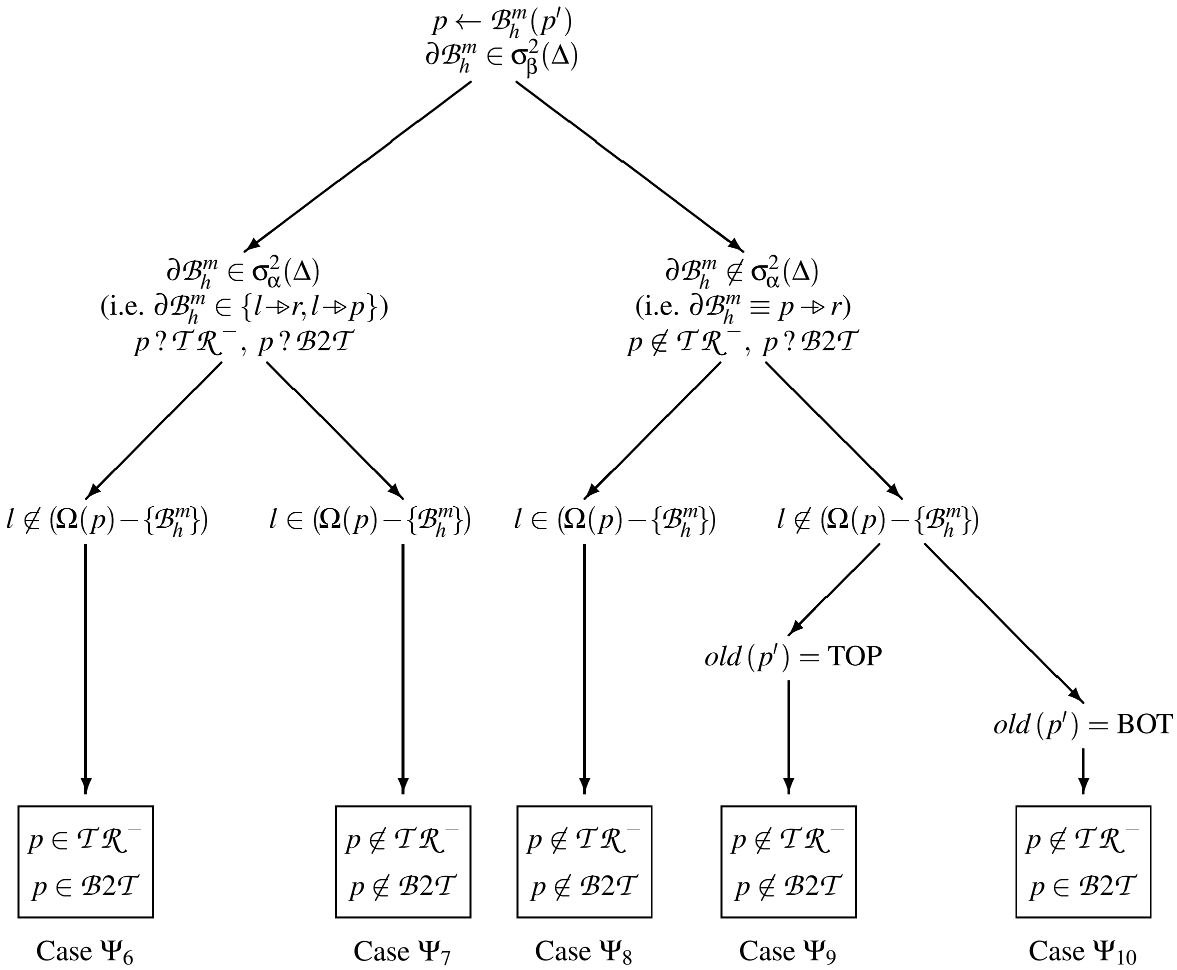


Figure 12.2: Various cases for $\partial\mathcal{B}_h^m \in \sigma_\beta^2(\Delta)$

The second factor, i.e. the old influence of $\mathcal{B}_h^m(p')$ is subsumed for the partition $\sigma_\alpha^2(\Delta)$ since $old\,(\mathcal{B}_h^m)$ is *lower*. Hence the second branching is introduced by the third factor, i.e. existence of a *lower* function influencing p. Since there exists such a function in cases Ψ_7 and Ψ_8, p cannot be deleted from TR_0 and hence is not included in $\mathcal{TR}^-$. Also, its value remains BOT due to the *lower* function and it cannot be included in $\mathcal{B2T}$. However, no *lower* function exists in $\Omega(p)$ in case Ψ_6, which implies that p must be deleted from TR_0 and its value may genuinely be TOP. Hence p is in both $\mathcal{TR}^-$ and $\mathcal{B2T}$.

Note that there is no *lower* function in $\Omega(p)$ in cases Ψ_9 and Ψ_{10} but this is not sufficient to decide the membership of p in $\mathcal{B2T}$; since $old\,(\mathcal{B}_h^m)$ is *propagate*, the old influence on p may well have been TOP in which case $p \notin \mathcal{B2T}$. This factor cannot be ignored in these cases. This introduces the third branching which helps to distinguish between the cases Ψ_9 and Ψ_{10}.

In all we have 10 distinct cases. We define a set $\Psi\star$ to contain these cases, and a partition π_λ on $\Psi\star$ such that each subset corresponds to one of the three influences mentioned at the beginning of the section :

$$\begin{aligned} \Psi\star &= \{\Psi_1, \Psi_2, \Psi_3, \Psi_4, \Psi_5, \Psi_6, \Psi_7, \Psi_8, \Psi_9, \Psi_{10}\} \\ \pi_\lambda(\Psi\star) &= \{\sigma_\lambda^1(\Psi\star), \sigma_\lambda^2(\Psi\star), \sigma_\lambda^3(\Psi\star)\} \\ &= \{\{\Psi_1, \Psi_4\}, \{\Psi_6, \Psi_{10}\}, \{\Psi_2, \Psi_3, \Psi_5, \Psi_7, \Psi_8, \Psi_9\}\} \end{aligned}$$

This partition captures the following influences :

1. $\sigma_\lambda^1(\Psi\star)$ contains the cases in which p may change to BOT since $p \in \mathcal{T}2\mathcal{B}$ and $p \notin \mathcal{B}2\mathcal{T}$.

2. $\sigma_\lambda^2(\Psi\star)$ contains the cases in which p may change to TOP since $p \notin \mathcal{T}2\mathcal{B}$ and $p \in \mathcal{B}2\mathcal{T}$.

3. $\sigma_\lambda^3(\Psi\star)$ contains the cases in which do not affect p since $p \notin (\mathcal{T}2\mathcal{B} \cup \mathcal{B}2\mathcal{T})$.

12.2.2 Global Influence

The global influence is captured in terms of the possible change in the value of a property $p'' \in \propto(p)$ for $p \leftarrow \mathcal{B}_h^m(p')$. The three possible influences on p'' are :

1. $p'' \in \mathcal{GCB}(p)$: The value of p'' is BOT.

2. $p'' \in \mathcal{GCT}(p) - \mathcal{NCT}(p)$: The value of p'' is TOP.

3. $p'' \notin \mathcal{GCB}(p) \cup \mathcal{GCT}(p)$: The value of p'' does not change.

The exact influence is determined by the chains that reach the property p''.

12.3 TR_0 Computation

Lemma 12.1 : TR_0 *update as defined by equation* (9.6) *satisfies equation* (4.3).

Proof : The lemma trivially holds for the properties for which no flow function changes as also for the properties in *Boundaryinfo* since there values are constant. We use partition $\pi_\alpha(\Delta)$ for other properties.

Figure 12.3 summarises all possible cases for the property $p \leftarrow \mathcal{B}_h^m(p')$. A "?" in the figure indicates that the set-membership of an element is not known.

Case 1 : $p \in \mathcal{TR}^+ \Rightarrow p \in TR_0$.

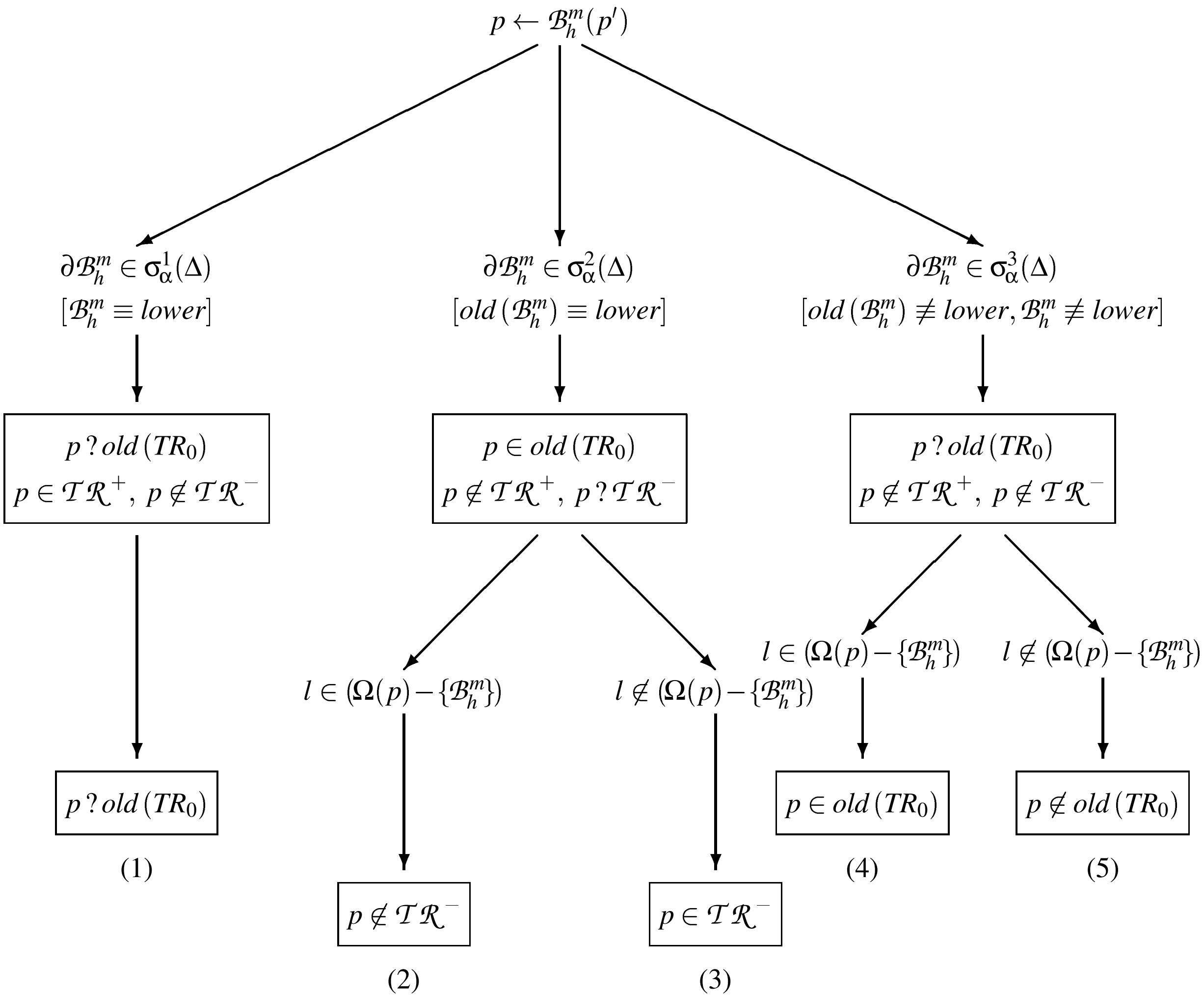


Figure 12.3: Various cases for lemma 12.1.

$\mathcal{B}_h^m \equiv lower \Rightarrow lower \in \Omega(p)$.

Case 2 : $p \in old(TR_0)$ and $p \notin \mathcal{TR}^- \Rightarrow p \in TR_0$.
$lower \in (\Omega(p) - \{\mathcal{B}_h^m\}) \Rightarrow lower \in \Omega(p)$.

Case 3 : $p \in old(TR_0)$ and $p \in \mathcal{TR}^- \Rightarrow p \notin TR_0$.
$lower \notin (\Omega(p) - \{\mathcal{B}_h^m\})$ and $\mathcal{B}_h^m \not\equiv lower \Rightarrow lower \notin \Omega(p)$.

Case 4 : $p \in old(TR_0)$ and $p \notin \mathcal{TR}^- \Rightarrow p \in TR_0$.
$lower \in (\Omega(p) - \{\mathcal{B}_h^m\}) \Rightarrow lower \in \Omega(p)$.

Case 5 : $p \notin old(TR_0)$ and $p \notin \mathcal{TR}^+ \Rightarrow p \notin TR_0$.
$lower \notin (\Omega(p) - \{\mathcal{B}_h^m\})$ and $\mathcal{B}_h^m \not\equiv lower \Rightarrow lower \notin \Omega(p)$.

Hence it follows that $p \in TR_0$ *iff* there exists a function $\mathcal{B}_h$ such that $p \leftarrow \mathcal{B}_h(p')$ and $\mathcal{B}_h \equiv lower$.
□

12.4 Correctness of S

12.4.1 An Outline of the Proof

Recall our convention that $p \leftarrow \mathcal{B}_h^m(p')$ is the property which is likely to be locally affected while p'' such that $p'' \in \propto(p)$, is a property elsewhere in the graph which is likely to be affected due to a change in p. We use the chains in $\text{CH}(p, p'')$ to discuss the possible influence on p''. Since chains are acyclic, p'' cannot be p, i.e. this discussion does not cover p. Hence we have to treat p and p'' distinctly and discuss the case of p separately.

Recall that showing correctness of S implies showing that each property in S satisfies condition 12.2 which is reproduced below :

$$\forall p,\ p = \text{BOT } \textit{iff } p \in TR_0 \text{ or } \exists\, p' \in TR_0 \text{ such that } \text{CH}(p', p) \neq \emptyset \tag{12.2}$$

It is easy to see that condition 12.2 implies the following two conditions :

$$\forall p,\ p = \text{BOT} \ \Rightarrow\ p \in TR_0 \text{ or } \exists\, p' \in TR_0 \text{ such that } \text{CH}(p', p) \neq \emptyset \tag{12.2.1}$$

$$\forall p,\ p = \text{TOP} \ \Rightarrow\ p \notin TR_0 \text{ and } \nexists\, p' \in TR_0 \text{ such that } \text{CH}(p', p) \neq \emptyset \tag{12.2.2}$$

The discussion in section 12.2 provides a neat outline for proving that the solution S as defined by the equation 9.14 is indeed the MFP solution. In the following, we consider only p and the properties corresponding to p.

1. *Correctness of unchanged properties.*

   A property $p'' \in \propto(p)$ may be influenced only if $\text{CH}(p, p'') \neq \emptyset$. Lemma 12.2 shows that a property p'' such that $\text{CH}(p, p'') = \emptyset$ satisfies condition 12.2 in the new MFP solution.

2. *Correctness of local change.*

   (a) Cases in $\sigma^1_\lambda(\Psi\star)$:
   In these cases,

   $$p \in \mathcal{T2B} \Rightarrow p \in \mathcal{GCB}(p) \Rightarrow p = \text{BOT}$$

   Since p cannot be TOP, condition 12.2.2 holds vacuously. Lemma 12.3 shows that the property p satisfies condition 12.2.1 for the cases in $\sigma^1_\lambda(\Psi\star)$.

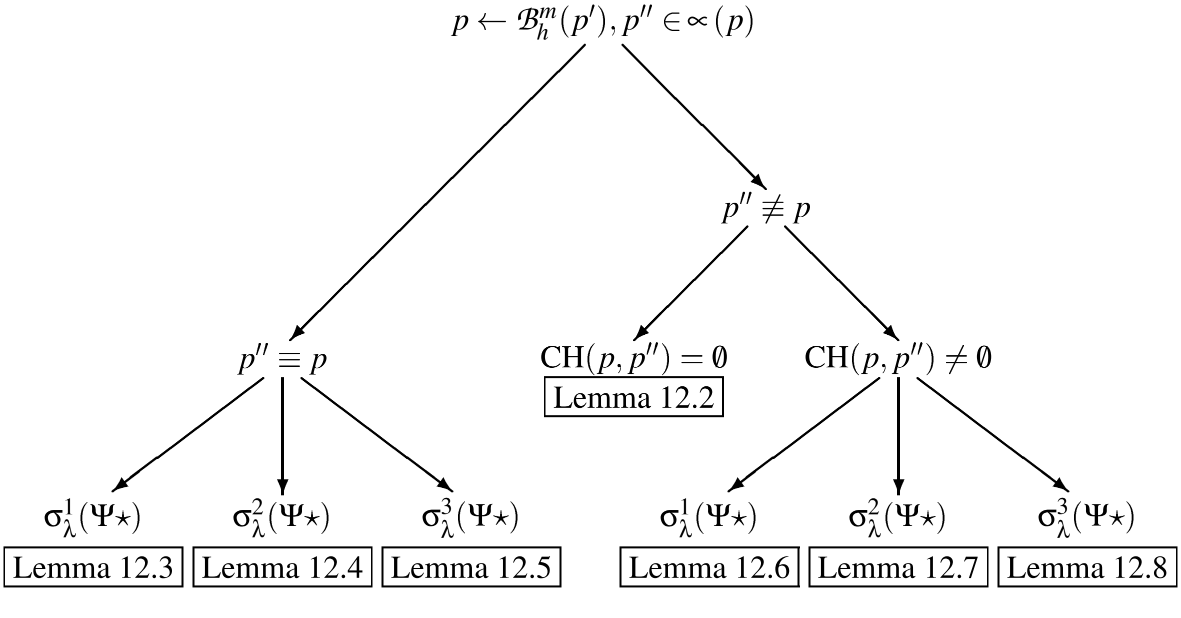


Figure 12.4: An outline of the proof

(b) Cases in $\sigma_\lambda^2(\Psi\star)$:

In these cases, p is in $\mathcal{B}2\mathcal{T}$ and its value could be either TOP or BOT. Lemma 12.4 shows that the property p satisfies conditions 12.2.1 and 12.2.2 for the cases in $\sigma_\lambda^2(\Psi\star)$.

(c) Cases in $\sigma_\lambda^3(\Psi\star)$:

In these cases $p \notin \mathcal{B}2\mathcal{T} \cup \mathcal{B}2\mathcal{T}$ i.e. the value of p does not change. It retains its old value which could be either TOP or BOT. Lemma 12.5 shows that the property p satisfies conditions 12.2.1 and 12.2.2 for the cases in $\sigma_\lambda^3(\Psi\star)$.

3. *Correctness of global change.*

In this category, we consider the properties $p'' \in \propto(p)$ such that $\text{CH}(p, p'') \neq \emptyset$.[2]

(a) Cases in $\sigma_\lambda^1(\Psi\star)$:

In these cases, p may or may not be in $\mathcal{GCB}(p)$. However, in either case, the value of p is BOT. Since p'' cannot be TOP, condition 12.2.2 holds vacuously. Lemma 12.6 shows that a property $p'' \in \propto(p)$, $p'' \not\equiv p$ such that $\text{CH}(p, p'') \neq \emptyset$, satisfies condition 12.2.1 for the cases in $\sigma_\lambda^1(\Psi\star)$.

[2]Note that some these properties too may remain unchanged but all of them can potentially change unlike the properties in step 1 above.

(b) Cases in $\sigma_\lambda^2(\Psi\star)$:

In these cases, $p'' \in \mathcal{GCT}(p)$. Its value could be either TOP or BOT. Lemma 12.7 shows that a property $p'' \in \propto(p)$, $p'' \not\equiv p$ such that $\text{CH}(p, p'') \neq \emptyset$, satisfies conditions 12.2.1 and 12.2.2 for the cases in $\sigma_\lambda^2(\Psi\star)$.

(c) Cases in $\sigma_\lambda^3(\Psi\star)$:

$\mathcal{GCB}(p) = \mathcal{GCT}(p) = \emptyset$ since $p'' \notin \mathcal{GCB}(p) \cup \mathcal{GCT}(p)$. Thus the value of p'' does not change; it could be either TOP or BOT. Lemma 12.8 shows that a property $p'' \in \propto(p)$, $p'' \not\equiv p$ such that $\text{CH}(p, p'') \neq \emptyset$, satisfies conditions 12.2.1 and 12.2.2 for the cases in $\sigma_\lambda^3(\Psi\star)$.

Figure 12.4 presents an outline of the proof.

12.4.2 Correctness of Unchanged Properties

Lemma 12.2 : *Let* $p \leftarrow \mathcal{B}_h^m(p')$ *and* $p'' \in \propto(p)$, $p'' \not\equiv p$ *such that* $\text{CH}(p, p'') = \emptyset$. *Then,*

$p'' = \text{BOT}$ *iff* $p'' \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\text{CH}(p''', p'') \neq \emptyset$.

Proof : We first show the following :

1. $p'' = \text{BOT}$ *iff* $old(p'') = \text{BOT}$, and

2. $p'' \in TR_0$ *iff* $old(p'') \in old(TR_0)$, and

3. $\exists\, p''' \in TR_0$ such that $\text{CH}(p''', p'') \neq \emptyset$ *iff*

$$\exists\, p''' \in old(TR_0) \text{ such that } old(\text{CH}(p''', p'')) \neq \emptyset.$$

Thus, our proof consists of the following steps :

1. $\text{CH}(p, p'') = \emptyset$
$$\begin{aligned} &\Rightarrow\ p'' \notin \mathcal{GCB}(p) \text{ and } p'' \notin \mathcal{GCT}(p) \\ &\Rightarrow\ p'' = old(p'') \\ &\Rightarrow\ p'' = \text{BOT } \textit{iff } old(p'') = old(\text{BOT}) \end{aligned}$$

2. $p'' \not\equiv p$
$$\begin{aligned} &\Rightarrow\ p'' \notin \mathcal{TR}^+ \cup \mathcal{TR}^- \\ &\Rightarrow\ p'' \in TR_0 \textit{ iff } p'' \in old(TR_0) \end{aligned}$$

3. $\triangleright$ $\text{CH}(p, p'') = \emptyset$

$\Rightarrow$ no chain in $\bigcup\limits_{p_i}$ CH(p_i, p'') includes p

$\Rightarrow$ no chain in $\bigcup\limits_{p_i}$ CH(p_i, p'') changes ... from section 9.2.2.2

$\Rightarrow$ $\forall p_i, \mathrm{CH}(p_i, p'') = old\,(\mathrm{CH}(p_i, p''))$ (12.2.A)

$\Rightarrow$ $old\,(\mathrm{CH}(p, p'')) = \emptyset$ (12.2.B)

$\triangleright$ $\exists\, p''' \in old\,(TR_0)$ such that $old\,(\mathrm{CH}(p''', p'')) \neq \emptyset$

$\Rightarrow$ p''' cannot be p ... since $old\,(\mathrm{CH}(p, p'')) = \emptyset$

$\Rightarrow$ $p''' \in old\,(TR_0)$ *iff* $p''' \in TR_0$[3] (12.2.C)

$\triangleright$ $\exists\, p''' \in old\,(TR_0)$ such that $old\,(\mathrm{CH}(p''', p'')) \neq \emptyset$

iff $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$ ... from (12.2.A) and (12.2.C)

Since condition 12.1 holds for $old\,(\mathbf{I})$, it follows that

$p'' = \mathrm{BOT}$ *iff* $p'' \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\mathrm{CH}(p''', p'') \neq \emptyset$. $\square$

12.4.3 Correctness of Local Change

Lemma 12.3 : *Let* $p \leftarrow \mathcal{B}_h^m(p')$ *for the cases in* $\sigma_\lambda^1(\Psi\star)$. *Then,*

$p = \mathrm{BOT} \Rightarrow p \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\mathrm{CH}(p''', p) \neq \emptyset$.

Proof : As noted earlier, $p \in \mathcal{T2B} \Rightarrow p \in \mathcal{GCB}(p) \Rightarrow p = \mathrm{BOT}$. We consider the two cases in $\sigma_\lambda^1(\Psi\star)$ separately.

1. Ψ_1 : $p \in \mathcal{TR}^+ \Rightarrow p \in TR_0$. (12.3.A)

2. Ψ_4 : Since $old\,(p') = \mathrm{BOT}$, either

   (a) $p' \in old\,(TR_0)$, or

   (b) $\exists\, p''' \in old\,(TR_0)$ such that $old\,(\mathrm{CH}(p''', p')) \neq \emptyset$.

   We consider the two cases separately. We know that $\mathrm{CH}(p', p) \neq \emptyset$ since $p' \in \mathcal{D}(p)$.

   $\triangleright$ $p' \in old\,(TR_0)$

   $\Rightarrow$ $p' \in TR_0$ ... since $p' \notin \mathcal{TR}^-$

   $\Rightarrow$ $\exists\, p' \in TR_0$ such that $\mathrm{CH}(p', p) \neq \emptyset$ (12.3.B)

   $\triangleright$ $\exists\, p''' \in old\,(TR_0)$ such that $old\,(\mathrm{CH}(p''', p')) \neq \emptyset$

[3] p is the only property that can be removed from or added to TR_0.

$\Rightarrow$ $p''' \in old(TR_0)$

$\Rightarrow$ p''' cannot be p ... since $old(p)$ is TOP.

$\Rightarrow$ $p''' \in TR_0$ (12.3.C)

$\triangleright$ $\exists\, p''' \in old(TR_0)$ such that $old(\mathrm{CH}(p''', p')) \neq \emptyset$ (12.3.D)

$\Rightarrow$ p''' cannot be p ... since $old(p)$ is TOP.

$\Rightarrow$ $old(\mathrm{CH}(p''', p'))$ cannot include p ... since $old(p)$ is TOP.

$\Rightarrow$ $old(\mathrm{CH}(p''', p')) = \mathrm{CH}(p''', p')$

$\Rightarrow$ Chains in $\mathrm{CH}(p''', p)$ are created ... since $\mathrm{CH}(p', p) \neq \emptyset$.

$\Rightarrow$ $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$ 12.3.E

... from (12.3.C) and (12.3.D)

It follows from statements (12.3.A), (12.3.B), and (12.3.E) that

$p = \mathrm{BOT} \Rightarrow p \in TR_0$ or $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$. □

Lemma 12.4 : *Let $p \leftarrow \mathcal{B}_h^m(p')$ for the cases in $\sigma_\lambda^2(\Psi\star)$. Then,*

$p = \mathrm{BOT}$ *iff* $p \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\mathrm{CH}(p''', p) \neq \emptyset$.

Proof : From equation 9.9, $p \in \mathcal{GCT}(p)$. The two possible cases are

1. $p \in \mathcal{NCT}(p) \Rightarrow p = \mathrm{BOT}$ ... from section (12.2.2)

   $p \in \mathcal{NCT}(p) \Rightarrow \exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$ ... from equation 9.12

2. $p \in (\mathcal{GCT}(p) - \mathcal{NCT}(p)) \Rightarrow p = \mathrm{TOP}$ ... from section (12.2.2)

   $\triangleright$ $p \in \mathcal{B2T}$

   $\Rightarrow$ $lower \notin \Omega(p)$ ... from equation 9.7

   $\Rightarrow$ $p \notin TR_0$ ... from Lemma 12.1

   $\triangleright$ $p \in (\mathcal{GCT}(p) - \mathcal{NCT}(p))$

   $\Rightarrow$ $p \notin \mathcal{NCT}(p)$ and $p \in \mathcal{GCT}(p)$

   $\Rightarrow$ $\nexists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$ ... from equation 9.12

   Hence it follows that

   $p = \mathrm{TOP} \Rightarrow p \notin TR_0$ and $\nexists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$.

□

Lemma 12.5 : *Let $p \leftarrow \mathcal{B}_h^m(p')$ for the cases in $\sigma_\lambda^3(\Psi\star)$. Then,*

$p = \text{BOT}$ *iff $p \in TR_0$ or $\exists\, p''' \in TR_0$ such that* $\text{CH}(p''', p) \neq \emptyset$.

Proof : Since $p \notin \mathcal{TR}^+$ and $p \notin \mathcal{TR}^-$, $TR_0 = old\,(TR_0)$ (12.5.A)

There are two possibilities : $p = \text{BOT}$ and $p = \text{TOP}$.

1. $p = old\,(p) = \text{TOP}$.

   Out of the six cases in $\sigma_\lambda^3(\Psi\star)$, only two cases qualify in this class : Ψ_5 and Ψ_9. In all other cases, i.e. in Ψ_2, Ψ_3, Ψ_7 and Ψ_8, $old\,(p)$ is BOT.[4]

   We need to show that $p \notin TR_0$ and $\nexists\, p''' \in TR_0$ such that $\text{CH}(p''', p) \neq \emptyset$. The first part is very easily to shown :

   $old\,(p) = \text{TOP}$
   $\Rightarrow\; p \notin old\,(TR_0)$
   $\Rightarrow\; p \notin TR_0$ ... from (12.5.A)

   In the following, we show that $\nexists\, p''' \in TR_0$ such that $\text{CH}(p''', p) \neq \emptyset$.

   - Ψ_5 : $\triangleright$ $\partial\mathcal{B}_h^m \equiv r \twoheadrightarrow p$
     $\Rightarrow$ chains involving $< p', p, \mathcal{B}_h^m >$ are created in $\text{CH}(p', p)$ (12.5.B)
     $\triangleright$ Acyclicity of chains
     $\Rightarrow$ $\forall p'''$, $\forall ch \in \text{CH}(p''', p')$, ch does not involve $< p', p, \mathcal{B}_h^m >$
     $\Rightarrow$ $\forall p'''$, no chains are added to $\text{CH}(p''', p')$
     (nor are chains deleted since $\partial\mathcal{B}_h^m \equiv r \twoheadrightarrow p$)
     $\Rightarrow$ $\forall p'''$, $\text{CH}(p''', p') = old\,(\text{CH}(p''', p'))$ (12.5.C)
     $\triangleright$ $old\,(p') = \text{TOP}$
     $\Rightarrow$ $p' \notin old\,(TR_0)$ and $\nexists\, p''' \in old\,(TR_0)$ s.t. $old\,(\text{CH}(p''', p')) \neq \emptyset$
     $\Rightarrow$ $p' \notin TR_0$ and $\nexists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p') \neq \emptyset$
     ... from (12.5.A) and (12.5.C)
     $\Rightarrow$ $\nexists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$ ... from (12.5.B)
   - Ψ_9 : $\triangleright$ $old\,(p) = \text{TOP}$
     $\Rightarrow$ $\nexists\, p''' \in old\,(TR_0)$ s.t. $old\,(\text{CH}(p''', p)) \neq \emptyset$
     $\Rightarrow$ $\nexists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$
     (since $\partial\mathcal{B}_h^m \equiv p \twoheadrightarrow r$, chains may only be deleted from $\text{CH}(p''', p)$)

[4] $old\,(p)$ is BOT in Ψ_7 and Ψ_8 since $lower \in \Omega(p)$.

Thus it follows that $p = \text{TOP} \Rightarrow p \notin TR_0$ and $\nexists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$.

2. $p = old\,(p) = \text{BOT}$.

   The remaining four cases in $\sigma^3_\lambda(\Psi\star)$ qualify in this class. Additionally, though $old\,(p')$ is TOP in Ψ_9, $old\,(p)$, and hence, p could be BOT due to the influence of some other property. Thus, in all we have five cases in this class. For each case, we have to show that either $p \in TR_0$ or $\exists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$.

   - Ψ_2 : $p \in \mathcal{TR}^+ \Rightarrow p \in TR_0$.
   - Ψ_3 : Since $old\,(p) = \text{BOT}$, either

     (a) $p \in old\,(TR_0)$, or

     (b) $\exists\, p''' \in old\,(TR_0)$ such that $old\,(\text{CH}(p''', p)) \neq \emptyset$.

     We consider the two cases separately.

     $\triangleright$ $p \in old\,(TR_0)$
     $\quad\Rightarrow\ p \in TR_0$ ... from (12.5.A)
     $\triangleright$ $\exists\, p''' \in old\,(TR_0)$ s.t. $old\,(\text{CH}(p''', p)) \neq \emptyset$
     $\quad\Rightarrow\ \exists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$ ... from (12.5.A)
     $\quad$ (since $\partial\mathcal{B}^m_h \equiv r \dashrightarrow\!\triangleright p$, chains may only be added to $\text{CH}(p''', p)$)

   - Ψ_7, Ψ_8 : $lower \in (\Omega(p) - \{\mathcal{B}^m_h\})$ ... from Figure 12.2
     $\quad\Rightarrow\ p \in TR_0$
   - Ψ_9 : $\triangleright$ $old\,(p') = \text{TOP}$ and $old\,(p) = \text{BOT}$ ... from Figure 12.2
     $\quad\Rightarrow\ \forall p''', \forall ch \in old\,(\text{CH}(p''', p))$, ch cannot involve $< p', p, old\,(\mathcal{B}^m_h) >$
     $\quad\Rightarrow$ no chain is deleted from $\text{CH}(p''', p)$
     $\quad$ (nor is any chain added to $\text{CH}(p''', p)$ since $\partial\mathcal{B}^m_h \equiv p \dashrightarrow\!\triangleright r$)
     $\quad\Rightarrow\ \forall p''',\ \text{CH}(p''', p) = old\,(\text{CH}(p''', p))$ (12.5.D)
     $\triangleright$ $old\,(p) = \text{BOT}$ and $lower \notin (\Omega(p) - \{\mathcal{B}^m_h\})$ ... from Figure 12.2
     $\quad\Rightarrow\ \exists\, p''' \in old\,(TR_0)$ s.t. $old\,(\text{CH}(p''', p)) \neq \emptyset$ (12.5.E)
     $\quad\Rightarrow\ \exists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$ ... from (12.5.A) and (12.5.D)

   Thus it follows that $p = \text{BOT} \Rightarrow p \in TR_0$ or $\exists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p) \neq \emptyset$.

$\square$

12.4.4 Correctness of Global Change

In this section, we are considering $p'' \in \propto(p)$, $p'' \not\equiv p$ such that $\mathrm{CH}(p, p'') \neq \emptyset$.

Lemma 12.6 : *Let* $p \leftarrow \mathcal{B}_h^m(p')$ *for the cases in* $\sigma_\lambda^1(\Psi\star)$. *Let* $p'' \in \propto(p)$, $p'' \not\equiv p$ *such that* $\mathrm{CH}(p, p'') \neq \emptyset$. *Then,*

$p'' = \mathrm{BOT} \Rightarrow p'' \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\mathrm{CH}(p''', p'') \neq \emptyset$.

Proof : In these cases, p'' may or may not be in $\mathcal{GCB}(p)$; the latter situation arises if $old(p'')$ is BOT. In either case, $p'' = \mathrm{BOT}$ and condition 12.2.2 holds vacuously. Further, though the cause of p'' being BOT may differ, the following argument holds for both the cases.

$p \in \mathcal{T}2\mathcal{B}$

$\Rightarrow$ either $p \in TR_0$ or $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$ …from Lemma 12.3

$\Rightarrow$ $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$ … since $\mathrm{CH}(p, p'') \neq \emptyset$

□

Lemma 12.7 : *Let* $p \leftarrow \mathcal{B}_h^m(p')$ *for the cases in* $\sigma_\lambda^2(\Psi\star)$. *Let* $p'' \in \propto(p)$, $p'' \not\equiv p$ *such that* $\mathrm{CH}(p, p'') \neq \emptyset$. *Then,*

$p'' = \mathrm{BOT}$ *iff* $p'' \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\mathrm{CH}(p''', p'') \neq \emptyset$.

Proof : We first show that $p'' \in \mathcal{GCT}(p)$ for the cases in $\sigma_\lambda^2(\Psi\star)$.

▷ $p \in \mathcal{B}2\mathcal{T} \Rightarrow old(p) = \mathrm{BOT}$ … from equation 9.7

▷ $\forall ch \in \mathrm{CH}(p, p'')$, ch does not include p

$\Rightarrow$ $\mathrm{CH}(p, p'') = old(\mathrm{CH}(p, p''))$

$\Rightarrow$ $old(p'') = \mathrm{BOT}$ … since $old(p)$ is BOT

$\Rightarrow$ $p'' \in \mathcal{GCT}(p)$ … since $p \in \mathcal{B}2\mathcal{T}$ and $\mathrm{CH}(p, p'') \neq \emptyset$

The only two cases which are possible, are

1. $p'' \in \mathcal{NCT}(p) \Rightarrow p'' = \mathrm{BOT}$. … from section (12.2.2)

   $p'' \in \mathcal{NCT}(p) \Rightarrow \exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$ … from equation 9.12

2. $p'' \in (\mathcal{GCT}(p) - \mathcal{NCT}(p)) \Rightarrow p'' = \mathrm{TOP}$ … from section (12.2.2)

   $\mathrm{CH}(p, p'') \neq \emptyset$ … from definitions (8.2) and (8.3)

   $\Rightarrow$ $lower \notin \Omega(p'')$ … from Lemma 12.1

$\Rightarrow\ p'' \notin TR_0$

$p'' \notin \mathcal{NCT}(p) \Rightarrow \nexists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$ ... from equation 9.12

Thus it follows that

$p'' = \mathrm{TOP} \Rightarrow p'' \notin TR_0$ and $\nexists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$.

□

Lemma 12.8 : *Let* $p \leftarrow \mathcal{B}_h^m(p')$ *for the cases in* $\sigma_\lambda^3(\Psi\star)$. *Let* $p'' \in \propto(p)$, $p'' \not\equiv p$ *such that* $\mathrm{CH}(p, p'') \neq \emptyset$. *Then,*

$p'' = \mathrm{BOT}$ *iff* $p'' \in TR_0$ *or* $\exists\, p''' \in TR_0$ *such that* $\mathrm{CH}(p''', p'') \neq \emptyset$.

Proof : Consider the chains from some property p''' to the property p''. Such chains are contained in $\mathrm{CH}(p''', p'')$. These chains can be divided into two categories :

1. Chains which involve $< p', p, \mathcal{B}_h^m >$: Let the set containing such chains be denoted by $\mathrm{CH}_p(p''', p'')$.
2. Chains which do not involve $< p', p, \mathcal{B}_h^m >$: Let the set containing such chains be denoted by $\mathrm{CH}_{np}(p''', p'')$.

Thus, it follows that,

$$\forall p''',\ \mathrm{CH}(p''', p'') = \mathrm{CH}_{np}(p''', p'') \cup \mathrm{CH}_p(p''', p'')$$

Hence this lemma can be proved by proving the following.

$$\begin{aligned} p'' = \mathrm{BOT} \quad &\textit{iff} \quad p'' \in TR_0 \text{ or} \\ &\exists\, p''' \in TR_0 \text{ s.t. either } \mathrm{CH}_{np}(p''', p'') \neq \emptyset \text{ or } \mathrm{CH}_p(p''', p'') \neq \emptyset^{5} \end{aligned}$$

We make some useful observations for proving the lemma.

▷ $p \notin \mathcal{TR}^+$ and $p \notin \mathcal{TR}^- \Rightarrow TR_0 = old(TR_0)$ (12.8.A)

▷ Cases Ψ_5 and Ψ_9

$\Rightarrow$ neither $old(\mathcal{B}_h^m)$ nor $\mathcal{B}_h^m$ is *lower*

$\Rightarrow$ only the chains which involve $< p', p, \mathcal{B}_h^m >$ may change[6]

[5] We do not exclude the possibility that both of them could be non-empty.

[6] If *lower* function is involved in $\partial\mathcal{B}_h^m$, the chains which include p, but do not involve $< p', p, \mathcal{B}_h^m >$ may also change. See section 9.2.2.2 for details.

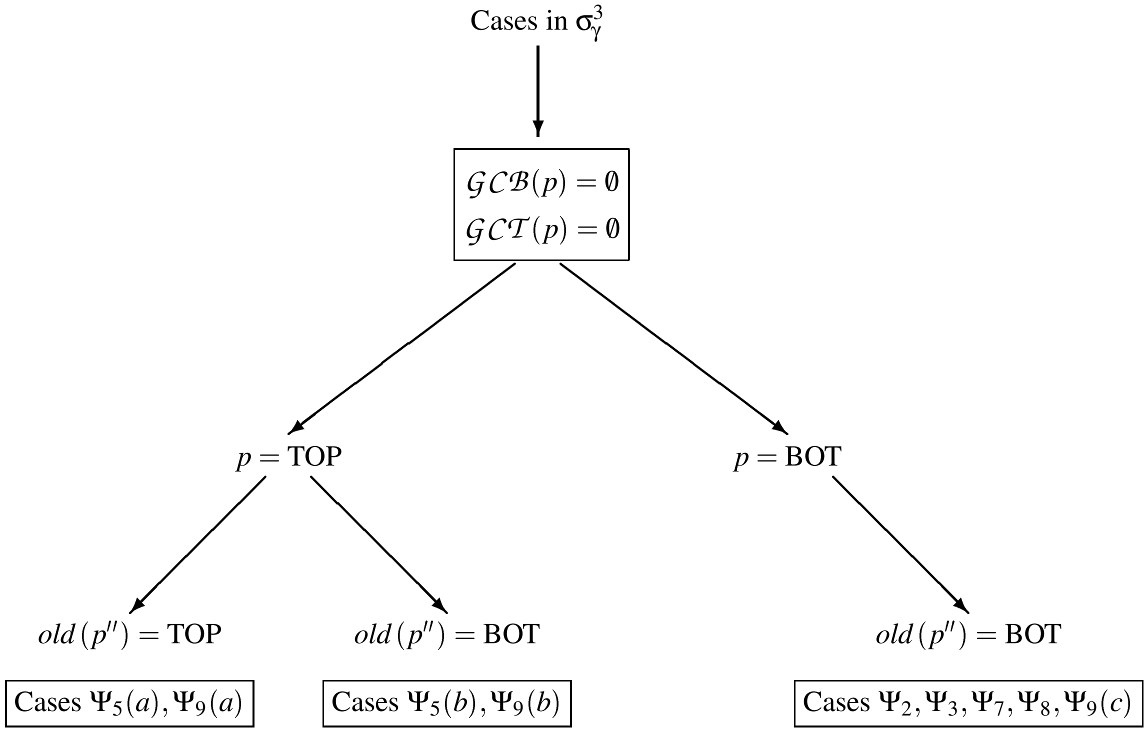


Figure 12.5: Various cases for a property p'' such that $\text{CH}(p, p'') \neq \emptyset$ for cases in σ_γ^1.

$$\Rightarrow\ \forall p''',\ \text{CH}_{np}(p''', p'') = old\,(\text{CH}_{np}(p''', p'')) \tag{12.8.B}$$

▷ Chains in $\text{CH}(p, p'')$ do not include p

$$\Rightarrow\ \forall p'',\ \text{CH}(p, p'') = old\,(\text{CH}(p, p'')) \tag{12.8.C}$$

Since $\mathcal{GCT}(p) = \mathcal{GCB}(p) = \emptyset$, no property changes. p and p'' retain their old values and the following possibilities exist :

1. As noted in lemma 12.5, p is necessarily BOT in cases Ψ_2, Ψ_3, Ψ_7 and Ψ_8. Additionally, p may be BOT in case Ψ_9 also. In the cases when p is BOT, p'' can only be BOT since $\text{CH}(p, p'') \neq \emptyset$.

2. p is necessarily TOP in case Ψ_5; it may be TOP in Ψ_9 too. p'' may or may not be TOP.

3. In the other cases, p'' could be TOP or BOT.

We consider the following three categories separately.

1. $p = \text{TOP}$ and $p'' = \text{TOP}$: Cases $\Psi_5(a)$ and $\Psi_9(a)$

2. $p = \text{TOP}$ and $p'' = \text{BOT}$: Cases $\Psi_5(b)$ and $\Psi_9(b)$

3. $p = \text{BOT}$ and $p'' = \text{BOT}$: Cases $\Psi_2, \Psi_3, \Psi_7, \Psi_8$ and $\Psi_9(c)$

Figure 12.5 contains the details.[7]

1. $\Psi_5(a), \Psi_9(a)$: $p = old(p) = \text{TOP}$ and $p'' = old(p'') = \text{TOP}$.

   We need to show that

   - $p'' \notin TR_0$, and (12.8.D)
   - $\nexists\, p''' \in TR_0$ such that $\text{CH}_{np}(p''', p'') \neq \emptyset$, and (12.8.E)
   - $\nexists\, p''' \in TR_0$ such that $\text{CH}_{p}(p''', p'') \neq \emptyset$ (12.8.F)

   Showing (12.8.D) and (12.8.E) is straightforward :

   ▷ $old(p'') = \text{TOP}$
   $\Rightarrow\ p'' \notin old(TR_0)$
   $\Rightarrow\ p'' \notin TR_0$ … from (12.8.A)

   ▷ $old(p'') = \text{TOP}$
   $\Rightarrow\ \nexists\, p''' \in old(TR_0)$ s.t. $old(\text{CH}_{np}(p''', p'')) \neq \emptyset$
   $\Rightarrow\ \nexists\, p''' \in TR_0$ s.t. $\text{CH}_{np}(p''', p'') \neq \emptyset$ … from (12.8.A) and (12.8.B)

   (12.8.F) is shown as follows. We consider the cases Ψ_5 and Ψ_9 separately.

   - $\Psi_5(a)$: Since $old(\mathcal{B}_h^m)$ is *raise*, $old(\text{CH}_p(p''', p'')) = \emptyset$, i.e. no chain from p''' to p'' involving $< p', p, old(\mathcal{B}_h^m) >$ existed in $old(\mathbf{I})$. Since $\partial\mathcal{B}_h^m \equiv r \twoheadrightarrow p$, chains are created in $\text{CH}_p(p', p'')$.

     ▷ Acyclicity of chains
     $\Rightarrow\ \forall p''', \forall ch \in \text{CH}(p''', p')$, ch does not involve $< p', p, \mathcal{B}_h^m >$
     $\Rightarrow\ \forall p'''$, no chains are added to $\text{CH}(p''', p')$
     (nor are chains deleted since $\partial\mathcal{B}_h^m \equiv r \twoheadrightarrow p$)
     $\Rightarrow\ \forall p''', \text{CH}(p''', p') = old(\text{CH}(p''', p'))$ (12.8.G)

     ▷ $old(p') = \text{TOP}$
     $\Rightarrow\ \nexists\, p''' \in old(TR_0)$ s.t. $old(\text{CH}(p''', p')) \neq \emptyset$
     $\Rightarrow\ \nexists\, p''' \in TR_0$ s.t. $\text{CH}(p''', p') \neq \emptyset$
     … from (12.8.A) and (12.8.G)

[7] Note that the possibility that $p = \text{BOT}$ and $p'' = \text{TOP}$ cannot arise since $\text{CH}(p, p'') \neq \emptyset$.

$\Rightarrow \;\; \nexists\, p''' \in TR_0$ s.t. $\mathrm{CH}_p(p''', p'') \neq \emptyset$[8]

- $\Psi_9(a)$: Since $old\,(\mathcal{B}_h^m)$ is *propagate*, $old\,(\mathrm{CH}_p(p''', p''))$ may or may not have been empty, i.e. such chains may have existed in $old\,(\mathbf{I})$. They become non-existent in $\mathbf{I}$ since $\partial\mathcal{B}_h^m \equiv p \not\triangleright r$.

  $\triangleright\; \mathcal{B}_h^m \equiv raise$,

  $$\Rightarrow \;\; \mathrm{CH}_p(p''', p'') = \emptyset \qquad (12.8.\mathrm{H})$$

  $\triangleright\; old\,(p'') = \mathrm{TOP}$

  $\Rightarrow \;\; p'' \notin old\,(TR_0)$ and $\nexists p''' \in old\,(TR_0)$ s.t. $old\,(\mathrm{CH}(p''', p'')) \neq \emptyset$

  $\Rightarrow \;\; \nexists p''' \in old\,(TR_0)$ s.t. $old\,(\mathrm{CH}(p''', p'')) \neq \emptyset$

  $\Rightarrow \;\; \nexists p''' \in TR_0$ s.t. $\mathrm{CH}(p''', p'') \neq \emptyset$

  (from (12.8.A) and the fact that no chains are added)

  $\Rightarrow \;\; \nexists\, p''' \in TR_0$ s.t. $\mathrm{CH}_p(p''', p'') \neq \emptyset$

  … from (12.8.A) and (12.8.H)

Thus it follows that

$p'' = \mathrm{TOP} \Rightarrow p'' \notin TR_0$ and $\nexists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$.

2. $\Psi_5(b), \Psi_9(b)$: $p = old\,(p) = \mathrm{TOP}$ and $p'' = old\,(p'') = \mathrm{BOT}$.

   For these cases, we need to show that

   $$p'' \in TR_0 \text{ or}$$
   $$\exists\, p''' \in TR_0 \text{ such that either } \mathrm{CH}_{np}(p''', p'') \neq \emptyset \text{ or } \mathrm{CH}_p(p''', p'') \neq \emptyset$$

   $\triangleright\; old\,(p) = \mathrm{TOP}$

   $$\Rightarrow \;\; \forall p''' \in old\,(TR_0), old\,(\mathrm{CH}_p(p''', p'')) = \emptyset \qquad (12.8.\mathrm{I})$$

   $\triangleright\; old\,(p'') = \mathrm{BOT}$

   $\Rightarrow \;\; p'' \in old\,(TR_0)$ or $\exists\, p''' \in old\,(TR_0)$ s.t. $old\,(\mathrm{CH}(p''', p'')) \neq \emptyset$

   $\Rightarrow \;\; p'' \in old\,(TR_0)$ or $\exists\, p''' \in old\,(TR_0)$ s.t. $old\,(\mathrm{CH}_{np}(p''', p'')) \neq \emptyset$

   (since $old\,(\mathrm{CH}(p''', p'')) = old\,(\mathrm{CH}_p(p''', p'')) \cup old\,(\mathrm{CH}_{np}(p''', p''))$, and $old\,(\mathrm{CH}_p(p''', p'')) = \emptyset$ from (12.8.I))

   $\Rightarrow \;\; p'' \in TR_0$ or $\exists\, p''' \in TR_0$ s.t. $\mathrm{CH}_{np}(p''', p'') \neq \emptyset$

   … from (12.8.A) and (12.8.B)

[8] This is, admittedly, a difficult step to figure out. It can be explained intuitively as follows : Since there is no p''' in TR_0 such that a chain from p''' to p' exists, there can be no p''' in TR_0 such that a chain from p''' to p'' involving $< p', p, \mathcal{B}_h^m >$ exists.

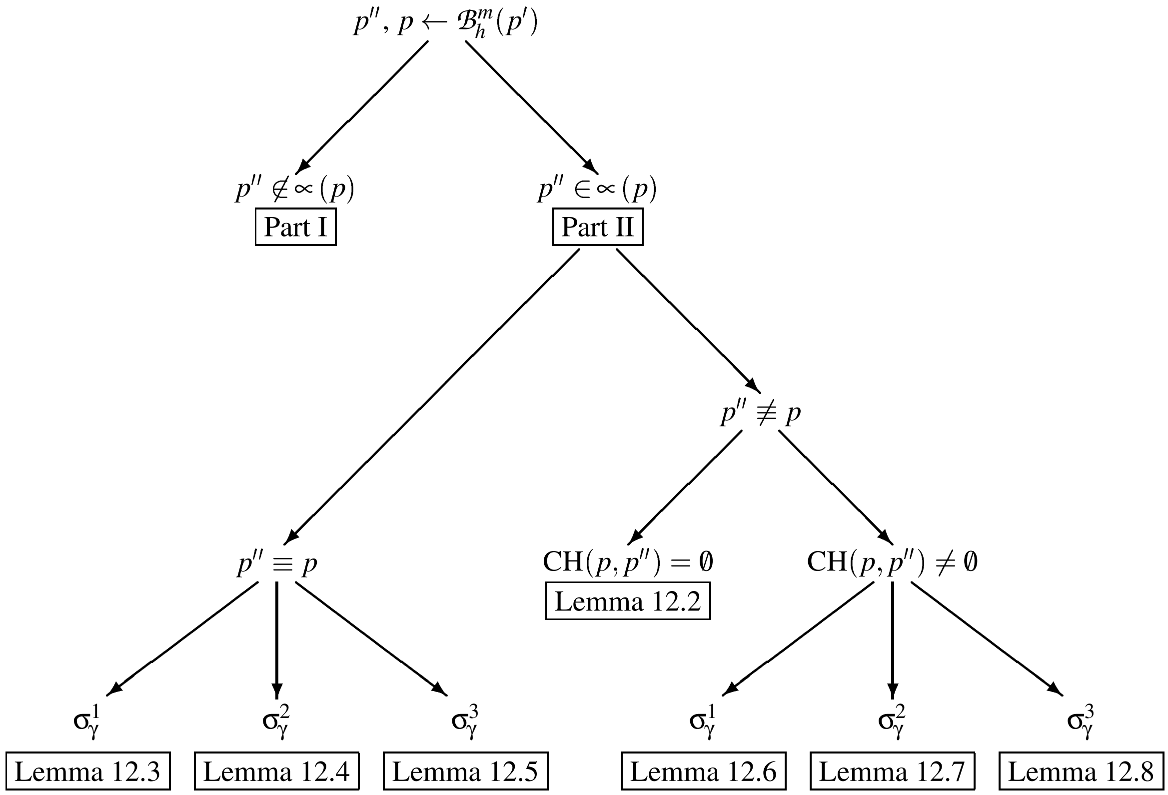


Figure 12.6: Various cases for theorem 12.1

Thus it follows that

$p'' = \mathrm{BOT} \Rightarrow p'' \in TR_0$ or $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$.

3. $\Psi_2, \Psi_3, \Psi_7, \Psi_8, \Psi_9(c) : p = old\,(p) = \mathrm{BOT}$.

$\triangleright$ $old\,(p)$ is BOT and $\mathrm{CH}(p, p'') \neq \emptyset$

$\Rightarrow$ $old\,(p)$ is BOT and $old\,(\mathrm{CH}(p, p'')) \neq \emptyset$ ... from (12.8.C)

$\Rightarrow$ $old\,(p'') = \mathrm{BOT}$

Thus, $p'' = old\,(p'') = \mathrm{BOT}$ and we need to show that,

$$p'' \in TR_0 \text{ or } \exists\, p''' \in TR_0 \text{ such that } \mathrm{CH}(p''', p'') \neq \emptyset$$

$\triangleright$ $p = \mathrm{BOT}$

$\Rightarrow \; p \in TR_0$ or $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p) \neq \emptyset$ ... from lemma 12.5

$\Rightarrow \; \exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$ ... since $\mathrm{CH}(p, p'') \neq \emptyset$

Thus it follows that

$p'' = \mathrm{BOT} \Rightarrow p'' \in TR_0$ or $\exists\, p''' \in TR_0$ such that $\mathrm{CH}(p''', p'') \neq \emptyset$.

□

Theorem 12.1 : *S as defined by equation 9.14 is the MFP solution for the instance* **I**.
Proof : For a property p such that $p \leftarrow \mathcal{B}^m_h(p')$, a property p'' may belong to either of the following categories :

- $p'' \notin \propto(p)$.

  Since such properties and their chains are unaffected by a change in $\mathcal{B}^m_h$, the theorem trivially follows.

- $p'' \in \propto(p)$.

  Various sub-cases, and the lemmas which prove them are presented in Figure 12.6.

It is easy to see from Figure 12.6, that all possible cases for a property p'' are covered. Hence the theorem follows. □

12.5 Multiple Function Changes

Devising formal proofs of correctness in the case of multiple function changes seems a rather formidable task. In this section, we provide intuitive arguments to justify why correctness should hold even if the assumption of a single function change is violated. Empirically, it has been found that the algorithm based on the model is capable of handling multiple function changes. This has been ascertained by implementing incremental data flow analysis algorithm for MRA (section 10.6).

Though there could be conflicting influences on the value of a property, it is automatically taken care of by the definitions in the model. We make some useful observations in this regard.

Observation 12.1 : $\left(\bigcup_{h^m \in \mathcal{H}^m} \mathcal{B}2\mathcal{T}_I(h^m)\right) \bigcap \left(\bigcup_{h^m \in \mathcal{H}^m} \mathcal{T}2\mathcal{B}_I(h^m)\right) = \emptyset$. □

This observation follows from the fact that a property belongs to either $\mathcal{B}2\mathcal{T}_I(h^m_1)$ set or $\mathcal{T}2\mathcal{B}_I(h^m_2)$ set depending upon its old value. Thus, in no case can it belong to both of them despite the possibility of two functions influencing a property in conflicting ways, directly. Let there be two functions $h_1^m(u', u)$ and $h_2^m(u'', u)$ influencing the properties at u. If $\mathcal{B}_{h1}^m$ and $\mathcal{B}_{h2}^m$ change in such a way that $\mathcal{B}_{h1}^m$ has TOP influence on a property at u while the $\mathcal{B}_{h2}^m$ has BOT influence, then the value of the property would be BOT. This is handled by the model as follows :

1. If the old value of a property p is BOT then
   - if $\mathcal{B}_{h2}^m$ is *lower*, p cannot be in $\mathcal{B}2\mathcal{T}_I(h_1^m)$ since $lower \in \Omega(p)$. Its value remains BOT.
   - if $\mathcal{B}_{h2}^m$ is *propagate*, p cannot be in $\mathcal{T}2\mathcal{B}_I(h_2^m)$ but would be in $\mathcal{B}2\mathcal{T}_I(h_1^m)$ and hence would be included in $\mathcal{GCT}(p)$. However, since the influence of $\mathcal{B}_{h2}^m$ is BOT, p will also be included in $\mathcal{NCT}(p)$ and its final value will be BOT.
2. If the old value of p is TOP, p would be in $\mathcal{T}2\mathcal{B}_I(h_2^m)$ and its final value would be BOT.

Observation 12.2 : *Let $p' \in \propto(p)$ such that $p \in \mathcal{T}2\mathcal{B}$ and $p' \in \mathcal{B}2\mathcal{T}$. Let $p'' \in \propto(p)$. If $CH(p, p'') \neq \emptyset$ and $CH(p', p'') \neq \emptyset$, then $p'' \in \mathcal{GCB}(p)$ if and only if $p'' \notin \mathcal{GCT}(p')$.* □

We consider two possible situations separately :

1. $old(p'')$ is TOP.

   In such a case, $p'' \in \mathcal{GCB}(p)$ and since its old value is TOP, it cannot be in $\mathcal{GCT}(p')$. Thus, p'' is BOT due to the BOT influence of p, regardless of the influence of p'.

2. $old(p'')$ is BOT.

   In such a case, $p'' \in \mathcal{GCT}(p')$ and since its old value is BOT, it cannot be in $\mathcal{GCB}(p)$. However since $CH(p, p'') \neq \emptyset$, p'' must be in $\mathcal{NCT}(p')$ and it would again be BOT.

Observation 12.3 : *Let $p' \in \propto(p)$ such that both p and p' are in $\mathcal{B}2\mathcal{T}$. Consider a p'' such that $p'' \in \mathcal{GCT}(p)$ and $p'' \in \mathcal{GCT}(p')$. Then, $p'' \in \mathcal{NCT}(p)$ if and only if $p'' \in \mathcal{NCT}(p')$.* □

This is obviously so since if $\exists\, p''' \in TR_0$ such that $CH(p''', p'') \neq \emptyset$ then p'' must be included in $\mathcal{NCT}$ set for every property for which it is included in $\mathcal{GCT}$ set.

In conclusion, the possibility of several functions changing together does not constrain any proposition of the model and there is no need to perform separate analysis for different changes; all changes can be handled simultaneously.

Part III

Concluding Remarks

Chapter 13

Loose Ends and Final Thoughts

> *But seeds are invisible. They sleep deep in the heart of the earth's darkness, until someone among them is seized with the desire to awaken. Then this seed will stretch itself and begin — timidly at first — to push a charming little sprig, inoffensively upward toward the sun. If it is only a sprout of radish or the sprig of a rose-bush, one would let it grow wherever it might wish.*

The generalised theory has evolved from some rather simple but fundamental observations. In particular, the basic insights revolve around the need to distinguish between :

- the edge flow and the node flow,
- the graph theoretic path and the information flow path,
- the characterisation of incremental data flow analysis and the algorithm to perform incremental data flow analysis.
- the incremental change in solution and the overall solution

The realisation that these distinctions need to be made, arises out of a series of not-so-satisfactory efforts at characterising bidirectional flows. Since these distinctions raise much more fundamental questions than the ones raised by the earlier (pioneering) efforts, they need much deeper probings which deliver much more than the earlier efforts — both, in terms of the number, as well as the significance of the results. The fact that this seems to have been achieved in an elegant manner may have something to do with the profoundness of the fundamental observations.

13.1 Contributions of this Work

The major contributions of this work are :

1. Formal characterisation of the notions in exhaustive and incremental data flow analysis (viz. information flow paths, influence of incremental changes in flow functions, dependence of data flow properties on other data flow properties etc.).

2. Formal models for exhaustive and incremental data flow analysis to :

   - define the exhaustive and incremental solutions of data flow problems.
   - show the correctness of exhaustive and incremental solutions.

3. Generic worklist based iterative algorithms for performing exhaustive and incremental data flow analysis.

4. Significant findings in complexity of exhaustive data flow analysis :

   - *Worklist based iterative data flow analysis* - We show that the complexity of unidirectional and bidirectional data flow analysis is same.
   - *Round robin iterative data flow analysis* - We define the notion of *width* which provides the first (strict) bound on the number of iterations for bidirectional data flow analysis. For the unidirectional problems, the width provides a more accurate bound than the traditional measure of *depth*.

5. Motivation and explanation of efficient solution techniques for exhaustive data flow analysis.

These results unfold deep insights into the process of data flow analysis and provide a firm theoretical foundation for understanding (and predicting) the behaviour of various data flow problems making it easier to devise and experiment with more unified optimising transformations.

13.2 Applicability

All formulations in this work have been conceived for bidirectional flows which subsume unidirectional flows as a special case. As a consequence, all results of this research are uniformly applicable to unidirectional and bidirectional data flows.

13.2.1 Applicability of the Results in Exhaustive Analysis

Though the exposition of the theory in this thesis is restricted to bit vector problems only, it is applicable to all bounded monotone data flow frameworks which possess the property of the separability of solution.

Let the effective height of $\mathcal{L}$ be H. Thus, a property can assume at most $H+1$ values during data flow analysis. This has the following consequences :

- MBVP now implies that a node variable changes from X_1 to X_2 where $X_1 \sqsupset X_2$.
- The notion of information flow now becomes :

  information flows from a program point u to a program point v when a change in a property at u causes the corresponding property at v to change.
- A property may change H times rather than only once; hence a program point may appear H times in an *ifp*.

These changes do not constrain any propositions in the theory except that a property may have to be processed H times rather than only once. The generic algorithm is also applicable to such problems by

- replacing the single-bit representations of a data flow property by a suitable data structure, and by
- replacing the words "is BOT" and "becomes BOT" by "is not TOP" and "changes", respectively.

If H is constant (i.e. independent of the number of nodes in the flow graph), the complexity of the algorithm remains same.

13.2.2 Applicability of the Results in Incremental Analysis

The results in incremental analysis are restricted to bit vector data flow problems only. The six possibilities of changes in flow functions introduced in chapter 9 form the basis of the functional model, and hence the entire discussion of incremental data flow analysis. The idea of extending the functional model to more general bounded problems[1] seems forbidding since one may have to consider many more possibilities and the whole treatment of incremental analysis would become unwieldy (not that it is very simple now!).

[1] Recall that bit vector problems are 2–bounded.

Appendix A

Constructing *ifp* Patterns

Seen from a slight distance, that would make a splendid spectacle. The movements of this army would be regulated like those of the ballet in the opera ... And never would they make a mistake in the order of their entry upon the stage. It would be magnificent.

We define the following predicates :

- $\mathcal{E}(h)$ indicates the existence of function h (i.e. h is non-$\top$).
- $\mathcal{T}(h)$ indicates that $h(\top) = \top$.
- $\mathcal{C}(in)$ indicates that CONST_IN could be $\perp$.
- $\mathcal{C}(out)$ indicates that CONST_OUT could be $\perp$.

These predicates can be determined directly from the data flow equations. A regular expression defining the *ifp*'s of a data flow framework is constructed in two steps :

1. Constructing DFA (Deterministic Finite Automaton) from flow functions.
2. Constructing a regular expression from DFA.

Standard techniques exist for constructing regular expressions from DFA [2, 33]. Here, we describe step (1).

Constructing DFA from flow functions

Define DFA $\mathcal{D} = < Q, \Gamma, \Delta, Q_{\mathcal{S}}, Q_{\mathcal{F}} >$ where,

- $Q \subseteq \{\mathcal{S}_s, \mathcal{S}_f, \mathcal{S}_b\}$ is the set of states where,

Δ	T_e^f		T_e^b	
	Condition	State	Condition	State
$\mathcal{S}_s$	$(\neg\mathcal{T}(f^f)+\mathcal{C}(out))\cdot\mathcal{E}(g^f)$	$\mathcal{S}_f$	$(\neg\mathcal{T}(f^b)+\mathcal{C}(in))\cdot\mathcal{E}(g^b)$	$\mathcal{S}_b$
$\mathcal{S}_f$	$\mathcal{E}(f^f)$	$\mathcal{S}_f$	$\mathcal{E}(g^f)\cdot\mathcal{E}(g^b)$	$\mathcal{S}_b$
$\mathcal{S}_b$	$\mathcal{E}(g^f)\cdot\mathcal{E}(g^b)$	$\mathcal{S}_f$	$\mathcal{E}(f^b)$	$\mathcal{S}_b$

Table A.1: Conditional transitional table for constructing DFA

  - $\mathcal{S}_s$ is the start state.
  - $\mathcal{S}_f$ is the state reached after a forward edge traversal (T_e^f) is made.
  - $\mathcal{S}_b$ is the state reached after a backward edge traversal (T_e^b) is made.

- $\Gamma \subseteq \{T_e^f, T_e^b\}$ is the alphabet.
- Δ is the state transition table which is derived from the conditional transition table (Table A.1) where a transition is possible only if the associated condition holds.
- $Q_{\mathcal{S}} = \mathcal{S}_s$.
- $Q_{\mathcal{F}} \subseteq \{\mathcal{S}_f, \mathcal{S}_b\}$ is the set of final states.

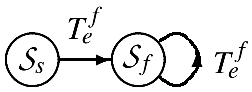
Note that the regular expression describing *ifp*'s for non-singular data flow problems, can be defined more precisely by examining the influence of two different confluence operators as discussed in example 6.1.

Example A.1 : For the problem of reaching definitions, both $\mathcal{C}(in)$ and $\mathcal{C}(out)$ are **F**. The predicates defined for flow functions have following values.

	f^f	f^b	g^f	g^b
$\mathcal{E}$	**T**	**T**	**F**	**F**
$\mathcal{T}$	**F**	**T**	**F**	**F**

The resulting transition table and its transition diagram is :

Δ	T_e^f
$\mathcal{S}_s$	$\mathcal{S}_f$
$\mathcal{S}_f$	$\mathcal{S}_f$



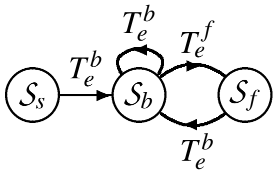
$\Gamma = \{T_e^f\}$, $\mathcal{Q} = \{\mathcal{S}_s, \mathcal{S}_f\}$ and $\mathcal{Q}_{\mathcal{F}} = \{\mathcal{S}_f\}$. The regular expression describing the *ifp*'s is $(T_e^f)^+$. $\square$

Example A.2 : For MRA, $\mathcal{C}(in)$ is **T** while $\mathcal{C}(out)$ is **F**. The predicates defined for flow functions have following values.

	f^f	f^b	g^f	g^b
$\mathcal{E}$	**F**	**T**	**T**	**T**
$\mathcal{T}$	**T**	**F**	**F**	**T**

The resulting transition table and its transition diagram is :

Δ	T_e^f	T_e^b
$\mathcal{S}_s$		$\mathcal{S}_b$
$\mathcal{S}_f$		$\mathcal{S}_b$
$\mathcal{S}_b$	$\mathcal{S}_f$	$\mathcal{S}_b$



$\Gamma = \{T_e^f, T_e^b\}$, $\mathcal{Q} = \{\mathcal{S}_s, \mathcal{S}_f, \mathcal{S}_b\}$ and $\mathcal{Q}_{\mathcal{F}} = \{\mathcal{S}_f, \mathcal{S}_b\}$. Note that the transition from $\mathcal{S}_s$ to $\mathcal{S}_f$ is not possible indicating that the *ifp*'s of MRA cannot begin with a forward edge traversal.

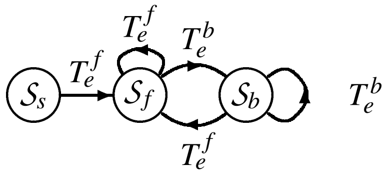
The regular expression in this case is $((T_e^b)^+(T_e^f \mid \varepsilon))^+$. $\square$

Example A.3 : For CHSA, $\mathcal{C}(in)$ is **F** while $\mathcal{C}(out)$ is **T**. The predicates defined for flow functions have following values.

	f^f	f^b	g^f	g^b
$\mathcal{E}$	**T**	**T**	**T**	**T**
$\mathcal{T}$	**T**	**T**	**T**	**T**

The resulting transition table and its transition diagram is :

Δ	T_e^f	T_e^b
$\mathcal{S}_s$	$\mathcal{S}_f$	
$\mathcal{S}_f$	$\mathcal{S}_f$	$\mathcal{S}_b$
$\mathcal{S}_b$	$\mathcal{S}_f$	$\mathcal{S}_b$



$\Gamma = \{T_e^f, T_e^b\}$, $\mathcal{Q} = \{\mathcal{S}_s, \mathcal{S}_f, \mathcal{S}_b\}$ and $\mathcal{Q}_{\mathcal{F}} = \{\mathcal{S}_f, \mathcal{S}_b\}$. $\mathcal{T}(f^b)$ is **T** and $\mathcal{C}(in)$ is **F**. Thus, there is no transition from $\mathcal{S}_s$ to $\mathcal{S}_b$ and the *ifp*'s cannot begin with T_e^b (table 3.2); they begin with T_e^f since $\mathcal{C}(out)$ is **T**.

The regular expression in this case is $(T_e^f)(T_e^f \mid T_e^b)^*$. $\square$

Appendix B

Performance of MRA Solution Procedure

On making his discovery, the astronomer had presented it to the International Astronomical Congress, in a great demonstration. But he was in Turkish costume, and nobody would believe what he said.
Grown-ups are like that . . .
Fortunately, however, for the reputation of Asteroid B-612, a Turkish dictator made a law that his subjects, under pain of death, should change to European costume. . . . And this time everybody accepted his report.

The algorithm was implemented and integrated in the optimiser of an optimising compiler developed at IIT Bombay. Extensive experiments were carried out with a set of scientific programs written in Fortran. All measurements were taken on a SPARC machine.[1]

Four heuristics were implemented and their performance was compared with the round robin method in terms of bit vector operations as well as time.

- RR refers to the round robin method.
- FIFO refers to *First in first out* strategy.
- MBOT refers to propagating the effect of node with *Maximum number of BOT* properties.
- PORD refers to maintaining the list sorted according *Postorder*.
- IPLM refers to an *Improved postorder list management* scheme.

[1] Initial implementation was carried out by Kavita Bala [6].

Bit vector operations

Table B.1 contains the results of measurements in terms of bit vector operations. The column headings have the following meanings :

n : Number of nodes.

$|X$: Number of expressions.

no_w : Number of words in a bit vector.

BO : Number of bit vector operations.

SF : Speed up factor over the round robin method.

Test Program				RR	FIFO		MBOT		PORD		IPLM	
Name	n	$\|X\|$	no_w	BO	BO	SF	BO	SF	BO	SF	BO	SF
kbr1	36	168	6	7056	3916	1.80	2139	3.30	2429	2.90	1755	4.02
kbr2	36	63	2	2400	2201	1.09	1083	2.22	998	2.40	727	3.30
kbr3	48	111	4	4176	5174	0.81	1942	2.15	1814	2.30	1218	3.43
kbr4	60	143	5	6480	9200	0.70	3025	2.14	2817	2.30	1877	3.45
kbr5	45	241	8	4560	3011	1.51	2684	1.70	2692	1.69	1912	2.38
kbr6	30	160	6	5544	3079	1.80	1708	3.25	1700	3.26	1282	4.32
kbr7	140	360	12	51840	24047	2.16	15563	3.33	16986	3.05	13005	3.99
kbr8	26	215	7	4095	6494	0.63	1931	2.12	1871	2.19	1263	3.24
kbr9	34	140	5	3600	1974	1.82	1407	2.56	1513	2.38	1142	3.15
kbr10	73	344	11	23232	15395	1.51	7343	3.16	7254	3.20	5133	4.53
kbr11	72	477	15	33120	17500	1.89	10244	3.23	10560	3.14	7852	4.22
kbr12	78	351	11	28776	36411	0.79	9925	2.90	9542	3.02	6365	4.52
kbr13	130	750	24	87552	30473	2.87	25809	3.39	25835	3.39	19023	4.60
kbr14	62	365	12	20880	8678	2.41	6243	3.34	6270	3.33	4509	4.63
kbr15	80	225	8	19200	10269	1.87	5959	3.22	6000	3.20	4262	4.50
kbr16	37	205	7	7476	5598	1.34	2408	3.10	2381	3.14	1644	4.55
kbr17	80	225	8	19200	10072	1.91	5979	3.21	5988	3.21	4277	4.49
Average				19363	11381	1.58	6199	2.84	6273	2.83	4543	3.96

Table B.1: Performance of MRA solution procedure in terms of bit vector operations.

Time measurements

Table B.2 contains the results of measurements in terms of time. The column headings have the following meanings :

n : Number of nodes.

$|X$: Number of expressions.

no_w : Number of words in a bit vector.

T : Number of bit vector operations.

SF : Speed up factor over the round robin method.

Test Program				RR	FIFO		MBOT		PORD		IPLM	
Name	n	$\|X\|$	no_w	T	T	SF	T	SF	T	SF	T	SF
kbr1	36	168	6	0.30	0.51	0.59	1.09	0.28	0.34	0.88	0.12	2.50
kbr2	36	63	2	0.13	0.31	0.42	0.71	0.18	0.15	0.87	0.07	1.86
kbr3	48	111	4	0.17	0.71	0.24	1.33	0.13	0.32	0.53	0.10	1.70
kbr4	60	143	5	0.25	1.30	0.19	2.26	0.11	0.44	0.57	0.14	1.79
kbr5	45	241	8	0.19	0.50	0.38	1.42	0.13	0.43	0.44	0.17	1.12
kbr6	30	160	6	0.20	0.37	0.54	0.92	0.22	0.23	0.87	0.09	2.22
kbr7	140	360	12	1.97	3.28	0.60	8.85	0.22	2.29	0.86	0.73	2.70
kbr8	26	215	7	0.15	0.68	0.22	1.22	0.12	0.25	0.60	0.09	1.67
kbr9	34	140	5	0.19	0.31	0.61	0.77	0.25	0.22	0.86	0.10	1.90
kbr10	73	344	11	0.78	1.93	0.40	4.37	0.18	0.89	0.88	0.33	2.36
kbr11	72	477	15	1.08	2.13	0.51	5.49	0.20	1.25	0.86	0.41	2.63
kbr12	78	351	11	0.98	4.30	0.23	6.83	0.14	1.26	0.78	0.41	2.39
kbr13	130	750	24	2.98	4.07	0.73	12.96	0.23	3.15	0.95	1.33	2.24
kbr14	62	365	12	0.70	1.20	0.58	3.14	0.22	0.86	0.81	0.32	2.19
kbr15	80	225	8	0.73	1.37	0.53	3.60	0.20	0.81	0.90	0.31	2.35
kbr16	37	205	7	0.31	0.74	0.42	1.49	0.21	0.33	0.94	0.13	2.38
kbr17	80	225	8	0.72	1.31	0.55	3.40	0.21	0.86	0.84	0.28	2.57
Average				0.70	1.47	0.46	3.52	0.19	0.83	0.79	0.30	2.15

Table B.2: Performance of MRA solution procedure in terms of time in seconds.

Appendix C

Width as a Complexity Measure

"Because an explorer who told lies would bring disaster on the books of the geographer. So would an explorer who drank too much."
"Why is that?" asked the little prince.
"Because intoxicated men see double. Then the geographer would note down two mountains in a place where there was only one."

The round robin iterative algorithms for MRA, EPA, and MMRA were implemented and integrated in the optimiser of an optimising compiler developed at IIT Bombay. Algorithms[1] for computing the widths of given test programs for MRA, EPA, and MMRA were also implemented. Extensive experiments were carried out with a a set of scientific programs written in Fortran.[2]

Table C.1 contains the results of the measurements. The round robin method visited the nodes in postorder for all the problems (i.e. T_G^b graph traversal). A very co-relation is found between the width and the number of iteration for MRA. For EPA and MMRA, the difference can be attributed to the presence of unbounded segments in both the directions. See section 6.6 for a complete analysis of the results.

Note that though $w+1$ iterations are sufficient for converging on a fixed point, an additional iteration is required to ascertain that the fixed point is reached. Thus, the actual bound on the number iterations for a round robin algorithm is $w+2$. It can easily be seen that on an average, the method os alternating iterations seems to be performing better that the round robin

[1]These algorithms were admittedly exponential, though some heuristics (based on the knowledge of *ifp*'s) were used which reduced the time requirements to almost half.

[2]Viral Acharya and Moses Charikar carried out the implementation and measurements.

program	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert X\rvert$	w'	MRA		EPA				MMRA			
					Backward		Backward		Alternate		Backward		Alternate	
					w	$\#i$	w	$\#i$	w_a	$\#i$	w	$\#i$	w_a	$\#i$
pgm1	36	62	168	1	2	4	13	4	5	5	26	6	20	5
pgm2	36	64	63	1	3	4	12	3	5	4	27	5	22	5
pgm3	48	68	111	1	3	3	23	10	7	5	39	10	31	5
pgm4	60	84	143	1	3	3	27	9	7	5	50	18	39	5
pgm5	45	50	241	1	2	2	37	3	5	3	41	2	9	3
pgm6	30	47	160	1	2	4	13	4	5	5	23	4	17	5
pgm7	34	46	140	1	4	3	22	5	7	5	27	4	15	5
pgm8	37	52	204	1	2	4	18	7	5	5	28	7	19	5
pgm9	61	84	224	1	3	4	29	3	5	4	47	4	29	4
pgm10	40	53	343	2	3	4	24	6	8	5	37	4	25	6
pgm11	46	78	207	1	2	4	14	5	5	5	26	7	20	6
pgm12	33	48	305	1	2	4	14	3	4	3	28	7	23	5
pgm13	34	48	54	1	2	4	24	3	3	3	33	4	11	4
pgm14	37	43	98	1	2	4	35	3	3	3	36	7	3	3
pgm15	62	83	365	1	2	4	30	3	5	4	41	4	21	5

Table C.1: Width as a complexity measure

method for EPA and MMRA. The column headings in the table have the following meanings :

$\lvert N\rvert$: Number of nodes

$\lvert E\rvert$: Number of edges

$\lvert X\rvert$: Number of expressions

$\#i$: Number of iterations

w' : Width for unidirectional problems

w : Width for bidirectional problems

w_a : Width for alternating iterations

Appendix D

Bitwise Algorithm for Incremental Data Flow Analysis : Some Measurements

Grown-ups love figures. When you tell them you have made a new friend, they never ask you any questions about essential matters. They never say to you, "What does his voice sound like ? What games does he love best? Does he collect butterflies?" Instead, they demand : "How old is he? How many brothers has he? How much does he weigh ? How much money does his father make?" Only from these figures do they think they have learned anything about him.

The algorithm was implemented and integrated in the optimiser of an optimising compiler developed at IIT Bombay. Extensive experiments were carried out with a set of scientific programs written in Fortran. All the measurements were taken on a SPARC machine.[1]

Table D.1 contains the results of measurements. It can be easily verified from the data that most of the time a function change affects a small fraction of the program flow graph vindicating the need for incremental analysis. Performance measurements were not carried out since the bitwise algorithm traverses a program flow graph for each bit separately while the exhaustive algorithm processes all the bits in a word simultaneously. On our machine, the word size was 32 bits which rendered any such comparison meaningless.

[1]The implementation was carried out by Sandeep Kumar [41].

The column headings have the following meanings :

n : Number of nodes.

$|X|$: Number of expressions.

no_w : Number of words in a bit vector.

AVA : Available Expressions Analysis.

MRA : Morel-Renvoise Algorithm.

FC : Number of (bit vector) function changes.

TB : Total number of properties in all $\mathcal{T}2\mathcal{B}$ sets.

BT : Total number of properties in all $\mathcal{B}2\mathcal{T}$ sets.

AR : Average size of the affected region as a fraction of the graph.

Test Program				AVA				MRA			
Name	n	$\|X\|$	no_w	FC	TB	BT	AR	FC	TB	BT	AR
kbr1	36	168	6	79	8	503	0.3928	224	16	1093	0.2500
kbr2	36	63	2	20	6	96	0.2798	74	8	344	0.0833
kbr3	48	111	4	19	0	9	0.1574	111	33	89	0.1250
kbr4	60	143	5	21	5	0	0.0000	168	38	33	0.0500
kbr5	45	241	8	142	3	3818	0.0852	461	5	5579	0.0444
kbr6	30	160	6	31	0	397	0.2866	129	29	777	0.5333
kbr7	140	360	12	324	6	1355	0.1791	1810	26	15135	0.0571
kbr8	26	215	7	19	12	22	0.5018	97	22	48	0.1538
kbr9	34	140	5	95	0	1158	0.0294	0	0	0	0.0000
kbr10	73	344	11	157	14	3356	0.1254	539	16	3116	0.1369
kbr11	72	477	15	115	9	2122	0.3058	1167	30	14130	0.1250
kbr12	78	351	11	144	18	265	0.6890	903	55	7126	0.0384
kbr13	130	750	24	737	23	16644	0.0668	3148	43	34191	0.0307
kbr15	80	225	8	115	7	1356	0.3154	856	11	7873	0.1625
kbr16	37	205	7	16	15	17	0.2480	24	18	24	0.0810
kbr17	80	225	8	106	6	1355	0.3134	775	10	8119	0.1875

Table D.1: Bitwise algorithm of incremental data flow analysis : Some Measurements.

Appendix E

Wordwise Algorithm for Incremental Data Flow Analysis : Some Measurements

> *If you were to say to the grown-ups : "I saw a beautiful house made of rosy brick, with geraniums in the windows and doves on the roof. " they would not be able to get any idea of the house at all. You would have to say to them : I saw a house that cost £4,000." Then they would exclaim : "Oh, what a pretty house that is !"*

The algorithm was implemented and integrated in the optimiser of an optimising compiler developed at IIT Bombay. Extensive experiments were carried out with a set of scientific programs written in Fortran. All the measurements were taken on a SPARC machine.[1]

Table E.1 contains the results of the empirical performance of the algorithm in terms of bit vector operations. No measurements were carried out for performance in terms of time since the goal of the implementation was to demonstrate the correctness of the algorithm practically.

No function changed for MRA in the case of program kbr9 (see Table D.1); hence no processing was required. Also, since information flow path do not necessary follow graph theoretic paths in the case of MRA, breadth first traversal is taken to mean essentially a list based traversal where nodes are added to and are selected from a list.

The column headings have the following meanings :

---

[1] Very early implementations of this algorithm were carried out by Kavita Bala [6] and Girija Narlikar [47]. Both the implementations failed to give correct results. Later, the approach of devising formal model before designing the algorithm was chosen and this time, the implementation was relatively straight-forward.

n : Number of nodes.
$|X|$: Number of expressions.
no_w : Number of words in a bit vector.
AVA : Available Expressions Analysis.
MRA : Morel-Renvoise Algorithm.
DR : Depth first traversal with revisits.
DN : Depth first traversal with no revisits.
BR : Breadth first traversal with revisits.
BN : Breadth first traversal with no revisits.

Test Program				Speed up factor for AVA				Speed up factor for MRA			
Name	n	$\|X\|$	no_w	DR	DN	BR	BN	DR	DN	BR	BN
kbr1	36	168	6	6.58	6.48	6.67	6.67	2.85	1.23	1.33	2.79
kbr2	36	63	2	4.84	4.84	4.94	4.94	8.69	8.44	8.44	8.69
kbr3	48	111	4	24.23	24.20	24.37	24.37	4.45	4.48	4.50	4.40
kbr4	60	143	5	46.61	46.61	46.61	46.61	10.56	12.52	12.52	10.56
kbr5	45	241	8	11.82	11.82	12.25	12.25	4.56	4.10	4.17	4.56
kbr6	30	160	6	5.49	5.40	5.60	5.60	3.09	1.87	1.87	3.07
kbr7	140	360	12	31.91	31.54	32.61	32.61	10.20	5.50	5.79	9.60
kbr8	26	215	7	9.28	9.26	9.34	9.34	7.53	8.32	8.28	7.53
kbr9	34	140	5	13.43	13.43	13.43	13.43	—	—	—	—
kbr10	73	344	11	25.34	25.28	25.93	25.93	4.98	2.62	2.68	4.93
kbr11	72	477	15	17.07	16.97	17.51	17.51	3.02	2.52	2.76	2.93
kbr12	78	351	11	39.46	39.08	39.35	39.35	13.86	12.21	12.19	13.85
kbr13	130	750	24	72.30	72.14	74.04	74.04	14.56	11.51	12.20	14.35
kbr15	80	225	8	21.73	21.79	22.21	22.21	3.41	2.79	2.89	3.37
kbr16	37	205	7	18.26	18.25	18.35	18.35	12.23	13.43	13.43	12.21
kbr17	80	225	8	18.03	18.06	18.51	18.51	3.42	2.69	2.89	3.36

Table E.1: Wordwise algorithm of incremental data flow analysis : Some Measurements.

Bibliography

[1] W. R. Adrion and M. A. Branstad J. C. Cherniacsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of ACM*, 19(3):137–147, 1977.

[4] T. J. Marlowe B. G. Ryder and M. C. Paull. Conditions for incremental iteration: examples and counterexamples. *Science of Computer Programming*, 11(1):1–15, 1988.

[5] R. C. Backhouse. Global data flow problems arising in locally least-cost error recovery. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[6] Kavita Bala. Performance evaluation of bidirectional data flow analysis. Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1992. B. Tech. project report.

[7] S. Biswas, G. P. Bhattacharjee, and P. Dhar. A comparison of some algorithms for live variable analysis. *International Journal of Computer Mathematics*, 8:121–134, 1980.

[8] M. Burke. An interval analysis approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, 1990.

[9] M. G. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7):723–728, 1990.

[10] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.

[11] M. Carroll and B. Ryder. Incremental data flow analysis via dominator and attribute udpates. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, 1988.

[12] F. C. Chow. *A portable machine-independent global optimizer — Design and measurements*. PhD thesis, Computer Systems Laboratory, Stanford University, 1983.

[13] K. Cooper. Analyzing aliases of reference formal parameters. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, 1985.

[14] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. SIGPLAN Notices, 19(6):247–258, 1984.

[15] P. Cousot. Semantic foundations of program analysis. (In [46]).

[16] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[17] D. M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[18] D. M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[19] D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimization. *International Journal of Computer Mathematics*, 27(1):1–14, 1989.

[20] D. M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instruction. *Computer Languages*, 15(2):83–94, 1990.

[21] D. M. Dhamdhere. Comments on practical adaptation of the global optimization algorithm by Morel & Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1994.

[22] D. M. Dhamdhere and J. R. Isaac. A composite algorithm for strength reduction and code movement optimization. *International Journal of Computers and Information Sciences*, 9(3):243–273, 1980.

[23] D. M. Dhamdhere and Uday P. Khedker. Complexity of bidirectional data flow analysis. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, 1993.

[24] D. M. Dhamdhere and Harish Patil. An elimination algorithm for bidirectional data flow analysis using edge placement technique. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, 1993.

[25] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[26] Vikram Dhaneshwar. M. Tech. dissertation (stage 1). Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1992.

[27] K. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimizations by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, 1988.

[28] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, 1976.

[29] S. M. Freudenberger. On the use of global optimization algorithms for the detection of semantic programming errors. Technical report NSO-24, NewYork University, 1984.

[30] V. Ghodssi. *Incremental analysis of programs*. PhD thesis, Department of Computer Science, Central Florida University, 1983.

[31] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of ACM*, 23(1):172–202, 1976.

[32] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc., 1977.

[33] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[34] S. M. Joshi and D. M. Dhamdhere. A composite algorithm for strength reduction and code movement : part I. *International Journal of Computer Mathematics*, 11(1):21–44, 1982.

[35] S. M. Joshi and D. M. Dhamdhere. A composite algorithm for strength reduction and code movement : part II. *International Journal of Computer Mathematics*, 11(2):111–126, 1982.

[36] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977.

[37] K. Kennedy. Safety of code movement. *International Journal of Computer Mathematics*, 3:112–130, 1972.

[38] Uday P. Khedker and D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, 1994.

[39] G. Kildall. A unified approach to global program optimization. In *Proceedings of the 1^{st} Annual ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[40] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992. Also Published as *SIGPLAN Notices*, 27(7).

[41] Sandeep Kumar. Implementation of a framework for global data flow analysis. Department of Computer Science, University of Pune, Pune, 1995. M. Tech. project report.

[42] T. J. Marlowe. *Incremental iteration and data flow analysis*. PhD thesis, Department of Computer Science, Rutgers University, 1989.

[43] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of the 17^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, 1990.

[44] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

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

[46] S. S. Muchnick and N. D. Jones. *Program Flow Analysis : Theory and Applications*. Prentice-Hall, Inc., 1981.

[47] Girija Narlikar. Formalization of a new model for incremental dataflow analysis. Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1993. B. Tech. project report.

[48] L. J. Osterweil. Using data flow tools in software engineering. (In [46]).

[49] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, 1989.

[50] B. Rosen. A lubricant for data flow analysis. *SIAM Journal of Computing*, 11(3):493–511, 1982.

[51] B. K. Rosen. Degrees of availability as an introduction the general theory of data flow analysis. (In [46]).

[52] B. K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal of Computing*, 9(1):159–196, 1980.

[53] B. K. Rosen. Linear cost is sometimes quadratic. *Proceedings of the 8^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 117–124, 1981.

[54] B. G. Ryder. Incremental data flow analysis. In *Proceedings of the 10^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 167–176, 1983.

[55] B. G. Ryder and M. Carroll. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 171–179, 1986. Also Published as *SIGPLAN Notices*, 21(1).

[56] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18:277–316, 1986.

[57] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.

[58] A. Sorkin. Some comments on a solution to a problem with Morel and Renvoise's "Global optimizations by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[59] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer Systems and Science*, 9(3):355–365, 1974.

[60] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of ACM*, 28(3):594–614, 1981.

[61] R. E. Tarjan. A unified approach to path problems. *Journal of ACM*, 28(3):577–593, 1981.

[62] V. Vyssotsky and P. Wegner. A graph theoretical fortran source language analyzer. AT & T Bell Laboratories, Murray Hill, N. J., 1963. (Manuscript).

[63] F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of SIGPLAN'84 Symposium. on Compiler Construction*, pages 132–143, 1984. Also Published as *SIGPLAN Notices*, 19(6).