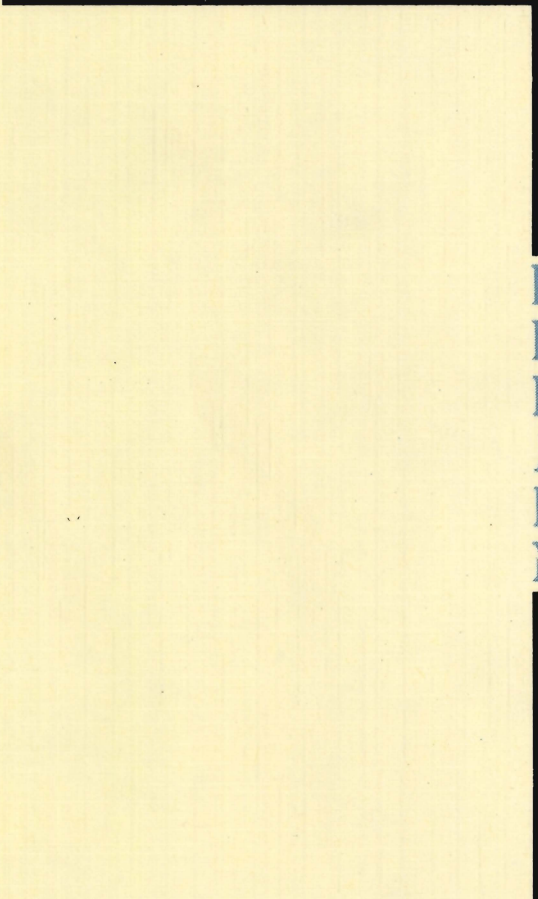


THE MISCONSTRUED SEMICOLON



**RECONCILING
IMPERATIVE
LANGUAGES
AND
DATAFLOW
MACHINES**

ARTHUR VEEN

The Misconstrued Semicolon

The Misconstrued Semicolon

Reconciling Imperative Languages
and
Dataflow Machines

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE
TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE
HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR
MAGNIFICUS, PROF.DR. F.N. HOOGHE, VOOR EEN
COMMISSIE AANGEWEEZEN DOOR HET COLLEGE VAN
DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP
VRIJDAG 13 SEPTEMBER 1985 TE 16.00 UUR

DOOR

Arthur Hugo Veen

GEBOREN TE UTRECHT

Dit proefschrift is goedgekeurd
door de promotoren

Prof.dr. M. Rem

en

Prof.dr. J. Gurd

© 1985 by Arthur H. Veen, The Netherlands.

Cover design by Ruth Hogenboom

Printed at the Centrum voor Wiskunde en Informatica, Amsterdam.

Table of Contents

Samenvatting	v
Inleiding voor de Leek	vii
Acknowledgements	xi
Curriculum Vitae	xii
1 Introduction	1
1.1 The Origin of the Project	2
1.2 The Dataflow Compiler Project	3
1.3 The Demand Graph	4
1.4 Synopsis of the Thesis	5
2 Dataflow Machines	8
2.1 Parallel Computers	8
2.2 Dataflow Machine Language	11
Dataflow Programs	11
Dataflow Graphs	12
Conditional Constructs	13
Iterative Constructs and Reentrancy	15
Procedure Invocation	18
2.3 The Architecture of Dataflow Machines	19
A Processing Element	19
Dataflow Multiprocessors	22
Communication	23
Data Structures	24

2.4 A Survey of Dataflow Machines	24
Direct Communication Machines	26
Static Packet Communication Machines	27
Machines with Code Copying Facilities	28
Machines with Both Tag and Code Copying Facilities	29
Tagged Machines	29
2.5 The Manchester Data Flow Machine	31
2.5.1. Overview	31
2.5.2. The Match Operation	33
2.5.3. Instruction Set	36
2.5.4. State of the Project	37
2.6 Feasibility of Dataflow Machines	37
2.6.1. Processing	38
2.6.2. Storage	39
2.6.3. Conclusions	40
3 Dataflow Programming	44
3.1 Declarative Languages	45
3.1.1. SISAL	46
3.1.2. Functional Languages	48
3.2 Imperative Languages	50
3.3 Imperative versus Declarative Languages	52
4 Program Flow Analysis	55
Graph Terminology	56
4.1 Applications	56
Example of an Application	57
Abstract Applications	58
4.2 Existing Methods	59
4.2.1. Interprocedural Analysis	60
4.2.2. Intraprocedural Analysis	61
5 The Demand Graph Method	66
5.1 Evolution of the Demand Graph Method	66
5.2 Language-Independent Aspects	68
5.2.1. Syntactic Analysis	68
5.2.2. Demand Graph Construction	70
5.2.3. Demand Propagation	80
5.2.4. Extraction	82

6 Demand Graph Construction	83
6.1 The SUMMER Programming Language	83
6.2 Overall Structure	87
The Type Tree	88
Construction of the Syntax Trees	88
Attach Procedures	89
6.3 Naive Demand Graph Construction	89
Assignments, Variables, and Constants	90
Input and Output	91
6.4 Conditional Control Flow	92
BRANCH, MERGE and LINK Nodes	93
Conditional Cocoons	94
Case Expressions	94
Failure Mechanism	95
AND and OR Nodes	96
Conditional Expressions in Address or Value Context	98
Iteration	98
6.5 Multiprocedural Graphs	100
Global Variables	100
Return Expressions	102
6.6 Arrays	103
ARRAY and ARRAY-ACCESS Nodes	104
Accesses from within a Conditional	106
Accesses from within a Loop	107
6.7 Conditional Aliasing	108
The LACAP Algorithm	109
Functional Description	110
Example	112
Implementation	113
Alias Graphs that are not Trees	115
Crossing Cocoon Boundaries	116
Case Expressions, Loops, and Procedures	117
7 Demand Propagation	118
7.1 Forward Propagation through an Acyclic Graph	119
7.2 Propagation in a Cyclic Graph	123
7.3 Backward Flowing Information	127
7.4 Bi-Directional Information Flow	128

8 Generating Dataflow Code	130
8.1 The Target Language	131
8.2 General Mechanisms	136
8.3 Simple Operations	137
Type Handling	137
Strings	138
Literals	139
Input and Output	140
8.4 Control Flow	140
Conditional Constructs	141
Optimizations Recognized by BRANCH Nodes	142
Procedure Interfacing	142
Iteration	143
8.5 Arrays	144
Macros	145
Completion Detection	147
Loops	149
Conditional Aliasing	150
8.6 Loop Optimizations	152
8.6.1. Parallel Distribution of Loop Constants	152
8.6.2. Complete Array Update	155
8.6.3. Reduction Cycles	155
9 Evaluation	157
9.1 Quality of the Generated Dataflow Code	157
9.2 Complexity	161
9.3 Extensions	162
9.3.1. Omissions	162
9.3.2. Further Optimizations	164
9.4 Conclusions	164
Program Analysis	164
Dataflow Programming	165
A Functional Perspective on Imperative Programs	166
I From Program to Parse Tree	168
II Algorithm for Demand Graph Construction	171

Samenvatting

Dit proefschrift heeft drie onderwerpen: dataflow machines, analyse van imperatieve programma's en het gebruik van analyse om het imperatief programmeren van dataflow machines mogelijk te maken.

Dataflow machines zijn asynchrone parallele computers, waarin de processen die parallel worden uitgevoerd zeer klein zijn; ongeveer zo groot als een conventionele machine-instructie. *Scheduling is data-driven*: een proces wordt pas uitgevoerd als alle benodigde invoer beschikbaar is. Hoofdstuk 2 vergelijkt dataflow machines met andere parallele computers en behandelt de architectuur van dataflow machines en de belangrijkste ontwerpbeslissingen. Aan de hand van een algemeen model voor dataflow machines wordt een uitgebreid overzicht gegeven van de meeste ontwerpen waarover is gepubliceerd. Een van de weinige operationele prototypen, de Manchester Dataflow Machine, wordt gedetailleerd beschreven. Op basis van ervaringen met deze machine wordt een eerste evaluatie gegeven van de geschiktheid van het dataflow concept als basis voor een efficiënte *general purpose* computer.

Hoofdstuk 3 vergelijkt verschillende methoden voor het programmeren van dataflow machines. De gebruikelijke benadering is het gebruik van een, al dan niet speciaal ontworpen, applicatieve programmeertaal, omdat een dergelijke taal vrij eenvoudig naar dataflow machinecode is te vertalen. Een voorbeeld, de dataflow taal SISAL, wordt beschreven en de beperkingen van een dergelijke benadering worden behandeld. Dataflow machines zouden aantrekkelijker worden als zij ook efficiënt zouden kunnen worden geprogrammeerd in de voor andere computers gebruikelijke talen, de zogenaamde imperatieve programmeertalen. Een overzicht wordt gegeven van de problemen die bij een dergelijke aanpak verwacht kunnen worden.

In de rest van dit proefschrift wordt een compiler beschreven, die een imperatief programma naar dataflow machinecode vertaalt. Zo'n compiler moet een vrij uitgebreide data-afhankelijkheidsanalyse uitvoeren. Omdat een dergelijke analyse ook nuttig is voor velerlei andere toepassingen, werd een algemene methode voor analyse van imperatieve talen ontwikkeld, de z.g. *demand graph* methode. De volgende vier hoofdstukken behandelen analyse van imperatieve programma's onafhankelijk van dataflow machines. Hoofdstuk 4 geeft een overzicht van bestaande analysemethoden en introduceert terminologie die in de volgende drie hoofdstukken wordt gebruikt. Hoofdstuk 5 bevat een globale beschrijving van de *demand graph* methode. Een compiler gebaseerd op deze methode vertaalt een programma eerst naar een graaf waarin alle data-afhankelijkheden expliciet zijn weergegeven. Hoofdstuk 6 geeft een gedetailleerde beschrijving van het construeren van een *demand graph* zoals geïmplementeerd voor de programmeertaal SUMMER. Dit is een imperatieve taal, waarin voorwaartse sprongen en *aliasing* een prominente rol spelen. Aan deze twee taalelementen, die de analyse aanzienlijk compliceren, wordt in dit hoofdstuk uitgebreid aandacht besteed.

Als de constructie van de *demand graph* voor een programma is voltooid, wordt de analyse voortgezet met het verspreiden van informatie in de graaf. De aard van de informatie en de wijze van verspreiden is afhankelijk van de toepassing van de analyse. In hoofdstuk 7 wordt dit deel van de analyse besproken aan de hand van enkele voorbeelden.

Hoofdstuk 8 behandelt de toepassing van de *demand graph* methode voor het vertalen van SUMMER naar machinecode voor de Manchester Dataflow Machine. De vertaling van *demand graph* naar dataflow machinecode is over het algemeen vrij eenvoudig. Een efficiënte implementatie van arrays vereist echter nog enige voortgezette analyse. Om het parallellisme van het vertaalde programma te verhogen, worden nog enkele optimalisaties voor *loops* uitgevoerd.

In hoofdstuk 9 wordt de kwaliteit van de code, gegenereerd voor enkele mini-programma's, vergeleken met de code geproduceerd door een compiler voor de dataflow taal SISAL. Het blijkt dat, althans voor deze eenvoudige programma's, de kwaliteit weinig verschilt, zowel wat betreft efficiëntie als parallellisme. Dit laatste resultaat is te danken aan de verwijdering van het sequentiële karakter van een imperatief programma (gesymboliseerd door de puntkomma operator) tijdens de vertaling naar de *demand graph*. In dit hoofdstuk wordt ook de complexiteit van de *demand*-graaf methode besproken. De evaluatie leidt tot de conclusie dat een imperatieve taal geschikt is voor het efficiënt programmeren van een dataflow machine.

Inleiding voor de Leek

Van de lezer van dit proefschrift wordt verwacht dat hij bekend is met computers en met de problemen die bij hun ontwerp en gebruik een rol spelen. In deze inleiding wordt getracht de belangrijkste punten uit het proefschrift voor een breder publiek duidelijk te maken. Eerst wordt de behoefte aan parallelle computers behandeld en vervolgens komen dataflow machines, programma-analyse en imperatieve programmeertalen aan de orde.

Hoewel veel van de huidige computers miljoenen bewerkingen per seconde kunnen uitvoeren, bestaat er voor vele toepassingen behoefte aan nog veel snellere computers. Weersvoorspelling is één van die toepassingen. Om een voorspelling van het weer van morgen te maken op grond van de huidige weersgesteldheid, berekent een snelle computer de veranderingen in de atmosfeer die het komende etmaal zullen plaatsvinden. Omdat de hoeveelheid gegevens die nodig zijn om de atmosfeer te beschrijven te groot is, moeten bij deze berekening vele locale effecten worden verwaarloosd. De voorspelling is daarom op een grove benadering gebaseerd: een berekening met iets meer detail zou weken duren, en dan zou er van voorspellen geen sprake meer zijn. Voor betere weersvoorspelling zijn dus veel snellere computers nodig. Er zijn nog vele andere toepassingen, waarvoor de snelheid van de huidige computers ontoereikend is. Bovendien, hoe snel computers ook zullen worden, er zal altijd behoefte blijven bestaan aan nog snellere.

De snelste computers van nu zijn duizenden malen sneller dan die van twintig jaar geleden. Deze versnelling is grotendeels te danken aan verbeteringen in de elektronische onderdelen, waaruit computers zijn opgebouwd. De komende jaren zullen nog meer van dit soort verbeteringen te zien geven, maar verwacht wordt dat het tempo van deze versnelling sterk zal afnemen. Mogelijkheden om computers sneller te maken moeten daarom elders worden gezocht, met name in de interne organisatie van computers.

Vrijwel alle bestaande computers zijn *sequentieel*: de biljoenen bewerkingen, waaruit een ingewikkelde berekening bestaat, worden alle in één lange reeks uitgevoerd door één centraal onderdeel: de *processor*. Met de huidige fabricagetechnieken voor *chips* kan een processor voor zeer weinig geld worden geproduceerd, speciaal als deze niet erg snel hoeft te zijn. Een snelle computer zou kunnen worden geconstrueerd, als deze goedkope processors zó aan elkaar kunnen worden gekoppeld dat ze nuttig kunnen samenwerken aan een gemeenschappelijke berekening. Zo'n computer, waarin vele bewerkingen tegelijkertijd worden uitgevoerd, wordt een *parallele computer* genoemd. Dit idee is al bijna zo oud als de computer zelf en er zijn de laatste twintig jaar verscheidene parallele computers ontworpen. Geen van deze computers is echter in staat gebleken om een hoge snelheid te behalen voor een grote verscheidenheid aan toepassingen.

De problemen die bij het ontwerp van een efficiënte parallele computer aan de orde komen worden uitgelegd aan de hand van een culinaire analogie. Het volgende lijstje geeft de overeenkomst aan.

keuken	parallele computer
bereiding	berekening
kok	processor
ingrediënt	invoergegeven
lopende band	pijplijn
recept	programma
receptenstijl	programmeertaal

We bekijken de organisatie in de keuken van een groot restaurant. Een sequentiële computer is als een keuken met slechts één kok. Als er veel gasten zijn, zal één kok het eten niet op tijd af krijgen: een aantal mensen, die we hier voor het gemak *koks* blijven noemen, zal moeten samenwerken. De vraag is nu, hoe de keuken georganiseerd moet worden zodat een groot aantal koks efficiënt kan samenwerken, zonder dat veel tijd verloren gaat met coördinatie of met wachten op elkaar. We zullen drie vormen van organisatie bekijken en het daarmee overeenkomend type parallele computer.

- Eén uiterste is een *lopende band*, die gerechten in verschillende stadia van bereiding van de ene kok naar de andere voert en waarbij iedere kok steeds dezelfde handeling herhaalt. Als alles soepel verloopt, is er vrijwel geen coördinatie nodig tijdens het koken. Een lopende band is een voorbeeld van een *synchrone* organisatie: alle koks werken in een vaste cadans. Om de lopende band efficiënt te laten werken, moeten de bewerkingen alle van gelijke duur zijn, anders staat een kok met een korte bewerking steeds te wachten. Een uitgebreide analyse is nodig om het kookproces in zulke stappen van gelijke lengte te verdelen. Zo'n analyse is alleen zinvol als de bereiding iedere dag hetzelfde is, zoals in een restaurant met een beperkt menu dat nooit verandert.

De meeste zeer snelle computers van dit moment zijn gebouwd rond een z.g. *pijplijn*. De functie van zo'n pijplijn komt overeen met die van een lopende band. Dit type computers is zeer geschikt voor berekeningen die

een grote regelmaat vertonen. Een uitgebreide analyse is echter nodig om programma's zó te schrijven, dat de pijplijn een groot deel van de tijd wordt benut. Deze analyse kan soms gedeeltelijk worden uitgevoerd door de *compiler*; dit is het programma dat nodig is om programma's die in een hoog-niveau programmeertaal zijn geschreven te vertalen naar eenvoudige laag-niveau bewerkingen.

- Een soepeler organisatie is vereist als de bereiding niet zo'n grote regelmaat vertoont, zoals in een restaurant met een groot en veranderlijk menu. Eén van de mogelijkheden is om iedere kok aan een apart gerecht te laten werken. Pas als zijn gerecht helemaal klaar is vraagt een kok aan een *coördinator* om een volgend gerecht. In dit geval werken de koks *asynchroon*: één kok kan achter elkaar een aantal simpele gerechten bereiden in de tijd die een andere kok nodig heeft voor de bereiding van één ingewikkeld gerecht. Een nadeel van deze organisatie is dat een kok soms tijdens de bereiding van zijn gerecht moet wachten, b.v. tot het water kookt. In deze tijd had hij kunnen assisteren bij de bereiding van een ander gerecht. Nog meer tijd gaat verloren als er op een bepaald moment meer koks beschikbaar zijn dan gerechten om te bereiden.

In een z.g. *grofkorrelige asynchrone parallelle computer* is de berekening op een soortgelijke manier verdeeld in een aantal grote deeltaken. Zo'n machine vertoont soortgelijke problemen als de zojuist beschreven keukenorganisatie. Tijdens de uitvoering van een deeltaak moet een processor soms wachten tot een invoergegeven beschikbaar komt. Ook het *parallelisme* van de berekening, dat is het aantal deeltaken dat op een bepaald moment tegelijkertijd uitgevoerd kan worden, kan soms onvoldoende zijn om alle processors te benutten. De programmeur moet de berekening op een zinnige manier weten te verdelen in een voldoende aantal deeltaken. Dit is vaak verre van eenvoudig. Bij deze analyse is veel minder hulp van de compiler te verwachten dan het geval is bij pijplijn-computers.

- De bereiding kan ook worden verdeeld in een groot aantal zeer simpele bewerkingen, die ieder in een korte tijd kunnen worden voltooid (b.v. "draai het vuur laag" als het water kookt). Er zijn twee voordelen vergeleken met de vorige organisatie. De eenvoudige bewerkingen zijn zo gekozen dat een kok nooit hoeft te wachten tijdens zo'n bewerking. Er zijn ook gemiddeld meer bewerkingen die tegelijkertijd uitgevoerd kunnen worden. Een groot nadeel van deze organisatie is dat veel tijd verloren gaat met coördinatie: de coördinator heeft vaak meer tijd nodig om een kok te vertellen wat hij moet doen dan de kok vervolgens nodig heeft om de bewerking uit te voeren.

In een z.g. *fijnkorrelige parallelle computer* is dit probleem ondervangen door een speciaal onderdeel in te bouwen dat zeer snel kan coördineren. Het programma moet daarvoor echter in een speciale vorm zijn geschreven. Het best ontwikkelde type fijnkorrelige parallelle computer is de *dataflow machine*. In zo'n computer bestaan de programma's uit een verzameling simpele bewerkingen en een beschrijving van hoe deze bewerkingen van elkaar afhangen. Dit noemen we een *dataflow programma*.

Dataflow programma's zijn van een laag niveau: de bewerkingen zijn zeer simpel en het programma is daarom lang. Programmeurs specificeren hun programma's in een hoog niveau programmeertaal. De vertaling tussen deze twee niveaus wordt verzorgd door een *compiler*. Om deze vertaling eenvoudig te houden, worden dataflow machines meestal geprogrammeerd in een *applicatieve* programmeertaal. In zo'n taal wordt een berekening gespecificeerd als een reeks definities in een willekeurige volgorde. In de meer gebruikelijke *imperatieve* programmeertalen wordt een berekening gespecificeerd als een reeks bewerkingen, waarbij de volgorde wel van belang is.

De keuken kan dienen om het verschil tussen deze twee typen talen uit te leggen. Een recept komt overeen met een programma. In de meeste recepten wordt expliciet de volgorde aangegeven waarin bewerkingen moeten worden uitgevoerd. Een recept voor "toast met ei" zou er als volgt uit kunnen zien:

Kook een ei 8 minuten.
Rooster een boterham.
Snij het ei in plakjes en doe het op de toast.

Dit noemen we een imperatief recept. Een hiermee overeenkomend applicatief recept ziet er als volgt uit:

Toast met ei is toast met een in plakjes gesneden hard-gekookt ei.
Toast is een geroosterde boterham.
Een hard-gekookt ei is een rauw ei dat 8 minuten heeft gekookt.

Applicatieve programmeertalen zijn vrij nieuw en het is nog niet duidelijk hoe geschikt ze zijn voor realistische grote programma's. Een groot probleem is ook dat vrijwel alle bestaande programma's in imperatieve programmeertalen zijn geschreven. Dataflow machines zouden veel aantrekkelijker worden als een compiler beschikbaar zou zijn die een imperatief programma kan vertalen naar een dataflow programma. Zo'n compiler is het belangrijkste onderwerp van dit proefschrift.

Om de analyse die zo'n compiler moet uitvoeren, te verduidelijken, keren we nog eenmaal terug naar de keuken. Een kok die volgens een imperatief recept kookt hoeft zich niet strikt aan de volgorde in dat recept te houden. Een deel van de volgorde is overbodig en zelfs, in geval een snelle bereiding nodig is, ongewenst. In bovenstaand voorbeeld kan het brood geroosterd worden terwijl het ei staat te koken. Het in plakjes snijden moet echter wachten tot het ei gekookt is. De analyse van het imperatief recept om te bepalen welke volgorde essentieel is en welke bewerkingen gelijktijdig kunnen is vaak zo simpel dat een kok er zich niet van bewust is.

De compiler die een imperatief programma naar een dataflow programma vertaalt moet een soortgelijke analyse uitvoeren. Zo'n compiler brengt de ordening in een imperatief programma terug tot het essentiële. Deze ordening wordt in veel imperatieve programmeertalen aangegeven door een *puntkomma*; de compiler leidt daarom in zekere zin tot een andere kijk op de puntkomma.

Acknowledgements

In addition to those officially associated with this thesis, many other people contributed to it. All their help I gratefully acknowledge.

The cradle of the project was the dataflow club, an informal and inspiring discussion group at the former Mathematical Centre. Its members have made valuable contributions over the past five years. From *Jan Heering* I learned to appreciate the spirit of scientific investigation. *Paul Klint* conceived and delivered SUMMER and has kept its implementation in working order. With *Wim Böhm* I shared the fascination with parallel computing. They all read early versions of this thesis and made many helpful comments.

The Centre for Mathematics and Computer Science gave the financial support and has been a pleasant place to work. Especially the excellent computing facilities provided by the *Informatica Laboratorium* have been of great help. *Frank van Dijk* and *Fred Veldkamp* implemented most of the algorithm for demand graph construction. The *Dataflow Research Group* in Manchester provided software, stimulating discussion and support. *Paul Vudnyi* gave advice on complexity of graph algorithms. *Gerard Kindervater*, *Steven Pemberton*, *Shirley Edwards*, and *Bert Mentink* made helpful remarks about the text. *Ruth Hogenboom* designed the cover. *Eloy Everwijn* gave valuable suggestions.

The help of *Marleen Sint* has been both essential and diverse. She is partly responsible for SUMMER and its implementation. She helped to clarify the main concepts in this thesis. She managed to read incomprehensible versions of this thesis and improved them considerably. She gave encouragement in the periods I needed it most. And finally she put up with me during the months of obsession with issues like italic font, past tense, and semicolons.

Curriculum Vitae

- Naam Arthur Veen.
- Geboren 27 april 1949, te Utrecht.
- 1967 Eindexamen HBS-B, Sint Bonifaciuslyceum, Utrecht.
- 1968 - 1970 Kandidaatassistent bij Prof.Dr. A. Lindenmayer, Utrecht.
- 1970 Kandidaatsexamen Wis- en Natuurkunde, Rijksuniversiteit Utrecht.
- 1971 - 1973 Research assistant bij Prof.Dr. R. Erickson, Philadelphia.
- 1973 Master of Computer and Information Science,
University of Pennsylvania, Philadelphia.
- 1974 & 1976 Research associate bij Prof.Dr. L. Peachey, Philadelphia.
- 1975 & 1976 Systeemanalist bij het Laboratorium voor Grondmechanica, Delft.
- 1977 - 1978 Projectleider bij het BAZIS, Academisch Ziekenhuis Leiden.
- 1978 - 1984 Wetenschappelijk medewerker bij het Centrum voor Wiskunde
en Informatica, Amsterdam.
- 1985 Research associate bij Prof.Dr. J. Gurd, Manchester.

Chapter 1

Introduction

Efficient cooperation is not easy. In the course of time organizational structures have evolved that allow groups of people to cooperate successfully. For computer processors, cooperation would also be desirable, but the organizational structures that are available are still primitive. These organizational structures have been studied in the areas of parallel computer architecture and distributed computing. The central problem is efficient coordination: processors have to be kept busy with relevant tasks, using each others results when appropriate, but the overhead associated with this coordination should not overshadow the real computation.

For certain well defined problem areas good solutions have been found. If the structure of the computational task is highly regular, the task can be easily divided, and the amount of work involved in each subtask accurately predicted. Scheduling, i.e. deciding when and where a subtask is to be executed, can then be done when the problem is analyzed rather than during execution. Many parallel computers that exploit such knowledge of the problem domain have been designed and some of them have been quite successful.

Most desirable is, of course, a general purpose parallel computer that performs well on a wide variety of computational tasks, but this is very hard to achieve. Most computational tasks show great and unpredictable variation in the distribution of their computing demands. Adjusting to this variation efficiently requires a flexible machine that constantly reallocates its resources. Such flexibility is offered by machines that maintain a common pool of executable subtasks. The problem is to limit the overhead that is involved in maintaining this common pool, while keeping the pool full enough to keep most processors busy.

The approach used in *fine grain* parallel computers is to maximize the number of concurrently executable tasks by dividing the program into many small subtasks, often the size of a conventional machine instruction. Since the average subtask is so small its scheduling should be highly efficient. Part of the scheduling overhead is due to the need for suspension of executing subtasks, when they need data from other subtasks. *Dataflow machines* are fine grain parallel computers in which coordination overhead is

reduced by obviating such suspensions: a subtask is not executable until all its input data are available. Scheduling overhead is further reduced by a combination of special hardware and a program format in which each subtask contains pointers to all subtasks that are dependent on its results. In this program format, called a *dataflow graph*, there are no control flow instructions and the data flow is made explicit.

Over the past fifteen years numerous dataflow machines have been proposed and most proposals have been accompanied by a special programming language that allows for simple translation from programs into dataflow graphs. These languages are known as *dataflow languages*. Dataflow graphs, however, can be generated for all kinds of programs including those written in more conventional, so called *imperative*, languages. This thesis results from a project in which this type of translation was studied. Before discussing the aims of this project we take a short look at its origins.

1.1. The Origin of the Project

We became familiar with literature on dataflow machines and early single assignment languages towards the end of 1979. Having had some experience with language design, we knew how hard it is to design a practical general purpose programming language and we were not impressed by the languages the dataflow field had produced so far. Neither were we convinced by the argument with which the development of dataflow languages was usually motivated: the complexity, or even impossibility, of translating any of the existing languages into dataflow graphs with sufficient parallelism. Even though converting control flow programs into dataflow graphs may not be straightforward, a large part of the data-dependency information could be uncovered relatively easy, as demonstrated by numerous optimizing compilers that use data-dependency analysis to help bridge the gap between language and machine. It was not clear to us *a priori* that the gap between existing languages and dataflow machines could not be bridged similarly. Several reasons make the issue too important to abandon without a serious effort.

The development of high-level programming languages has been intertwined with that of computer architecture. The connection has been far too intimate. The quality of a language should be judged by how well it supports good programming practice, whereas a good implementation (i.e. the combination of compiler and machine) should execute programs efficiently. These should be separate concerns, but the design of most languages has been guided by the implementations that were deemed feasible. FORTRAN is a prime example of this uneasy compromise between conflicting demands: although the language was intended to hide the peculiarities of a particular machine, at the time of its conception the concern with computing efficiency was so pervasive and the experience with translation so minimal that the class of machines for which it was designed is clearly visible. FORTRAN rapidly gained such a wide popularity that the language in turn guided, and probably hampered, the evolution of new architectures: a new machine was not attractive if it could not execute the existing software more efficiently than the old one. In fact, a similar influence works the other way around: in many eyes a new language is not attractive if its implementation on existing machines is much less efficient than implementations of existing languages. Architecture and language design are thus kept in a mutual strangle-hold. The development of dataflow languages in conjunction with dataflow machines is an attempt to break this strangle-hold by assuming that continuity in software development can be safely ignored. Several examples in the past indicate that this is a precarious assumption. A more fruitful approach may be to allow a wider gap between architecture and language and to develop program analysis methods to provide efficient translation.

To explore the difficulties involved in the translation of imperative languages into dataflow graphs, a pilot compiler was implemented that accepted a subset of the locally used language SUMMER and produced code for the dataflow machine being designed in Manchester. No description of the instruction set of the target machine was available at the time so a simple instruction set and a simulator for a somewhat idealized machine were devised. The central part of the translation was a data-dependency analysis that connected each instruction with all instructions that were dependent on its result. The analysis was supported by objects, called *cocoons*, that mimicked the role of the memory during conventional execution. Separate cocoons were created for each control flow path and the expressions translated within separate cocoons were connected by interface nodes, which in turn mimicked the control flow operators during dataflow execution.

The design and implementation of the pilot compiler were encouraging. In less than two months a compiler was produced that accepted programs with multiple assignment, global variables, conditionals, iteration, procedure calls, and interactive I/O. The auspicious implementation was partly due to the target machine, which was a conveniently idealized model of a real machine: its basic types and arithmetic operations coincided with those of the input language. Another factor was that no attention was paid to efficiency, although an effort was made to generate code with sufficient parallelism. The main reason for the success was however the choice of the input language: the subset avoided the complications caused by escapes, pointers, aliasing, and user defined types. Case statements, recursion, and arrays were also excluded from the subset, but the implementation of these features was expected to be straightforward.

1.2. The Dataflow Compiler Project

Encouraged by the results of the pilot compiler a research project was initiated to test the validity of the following hypothesis:

- A well structured imperative language is a suitable source language for a dataflow machine.

With "well structured" was meant a language without unrestricted jumps. The term "suitable" was made more precise by two supporting hypotheses:

- A translator from an imperative language into dataflow machine code is similar in complexity to a conventional optimizing compiler.
- Such a translator produces code similar in quality to that generated from a dataflow language.

One way to demonstrate the validity of these hypotheses would have been to implement the straightforward extensions to the compiler and to show somehow that the resulting input language was a generally useful programming language. In addition, it had to be shown that the simulated target machine was a realistic model for a dataflow machine. The latter point seemed easy enough, but proving the former point did not seem attractive: discussions on the usefulness of programming languages are hopelessly dominated by issues of taste.

Instead it was decided to follow a more complicated but potentially more convincing route by implementing a compiler for an existing language and an existing machine. Corners not cut did not have to be shown to be unimportant. The choice of a target machine was easy: the Manchester Dataflow Machine had reached its final stages of construction and its instruction set had stabilized. The choice of the input language was harder. SUMMER is purely a research language, but it contains most of the features that make translating imperative languages into dataflow graphs problematic. Since the

compiler is meant to demonstrate the feasibility of such a translation, rather than to be used as a production compiler, we decided after ample deliberation to stick with SUMMER as input as well as implementation language. An attractive consequence of this choice was that if a full implementation was produced, it could run on the dataflow machine itself. We did not fully realize at the time that some of the more obscure features of SUMMER make it into one of the hardest languages to translate into dataflow graphs.

Around the same time F. van Dijk and A. Veldkamp, students at the University of Amsterdam, started a short-term project to improve the conventional implementation of SUMMER by implementing a static type analyzer. Since the dataflow code generator would also need some form of static type analysis and since the data-dependency analysis needed in both projects was quite similar, it was decided to join forces into a new project. Its goal was to produce a general analyzer to be used for the two original projects and useful for other applications of flow analysis as well. This decision had far-reaching consequences; the emphasis of the research shifted from just dataflow code generation to program flow analysis in general.

1.3. The Demand Graph

A general data-dependency analyzer should express its results in a format that is convenient for a variety of applications. We decided to combine the data-dependency information with the syntax tree of the analyzed program into a new program representation, which we called the *demand graph*. It is structurally similar to a dataflow graph with all its arcs reversed. The demand graph is constructed with the aid of cocoons similar to the ones used in the pilot compiler. It does not contain any explicit control flow constructs: these have all been interpreted during the data-dependency analysis and their effects have been expressed in interface nodes created by the cocoon mechanism. Interface nodes encode the *static ambiguity* of data-dependency: they appear wherever data-dependency is influenced by conditional control flow.

An interesting effect is that often two different programs are translated into exactly the same demand graph. In this way the demand graph construction algorithm defines an equivalence relation on programs. The differences removed by the equivalence relation are due to an over-specification of execution order inherent in an imperative program. The statements in a program text are completely ordered, whereas the nodes in the demand graph constitute a partial order. In the interpretation of control flow constructs this superfluous ordering is removed. A poignant illustration is offered by the *semicolon* considered as sequence operator. During demand graph construction a semicolon separating two statements is interpreted as ordering the two statements only if dictated by data-dependencies. The semicolon thus changes from a sequence operator into a mere separator; the same role it has in many applicative languages.

The demand graph is a convenient program representation to carry out various flow analysis applications. The application specific analysis consists of depositing initial information in demand graph nodes and propagating the information through the graph, combining information when appropriate. The analysis has to be concerned only with data flow, since all control flow operators have already been interpreted. When the information collected in each node has stabilized, the results of the analysis can be extracted from selected nodes.

Implementing a demand graph constructor for the complete SUMMER language turned out to be too ambitious for the available man-power. The main reasons for this are:

- Designing and implementing a fully general analysis method was more work than the two original applications together.
- In some sense SUMMER is imperative to the extreme: both escapes and aliasing are pervasive in most programs. Dealing with these two issues efficiently required a considerable effort.

The main omissions are user-defined types, cyclic data structures, and interprocedural aliasing. The implemented subset, however, amounts to a fully usable language. The dataflow code generator developed for this subset allows some interesting comparisons with dataflow languages to be made; these will be discussed in the concluding chapter.

1.4. Synopsis of the Thesis

The chapters of this thesis do not have to be read in strict order. The chapter on dataflow code generation presupposes familiarity both with dataflow machines and how they are programmed (chapters 2 and 3) as well as with the analysis method (chapters 4 until 7). These two parts can be read in any order or concurrently.

Chapter 2 contains a comprehensive survey of dataflow machines. It presents a general model of a dataflow machine and discusses the crucial design choices. Numerous designs for dataflow machines, either constructed or merely proposed, are described as special cases of the general model. The use of a unifying terminology greatly facilitates comparisons between the different designs. The chapter contains a detailed description of the target machine for the code generator and is concluded by a discussion on the feasibility of dataflow machines as general purpose computers. This discussion is based on figures derived from experience with the Manchester Dataflow Machine, but has ramifications for other fine grain parallel computers including reduction machines.

Chapter 3 elaborates on the differences between applicative languages (of which dataflow languages are examples) and imperative languages, especially in relation to dataflow machines. It describes the notion of the average interface size of statements and presents this as the major factor determining the suitability of a program for fine grain parallel execution. It sheds new light on the continuing discussion about the relative merits of applicative and imperative languages.

Chapter 4 discusses the area of flow analysis and compares some existing methods, but is not intended as a survey. It introduces terminology used in the description of the analysis method.

The general analysis method is described in chapter 5; it subsequently treats the four phases of the analysis: syntactic analysis, demand graph construction, demand propagation, and extraction. Since the analysis method has a wider applicability than the input language for which it was implemented, the discussion in this chapter is kept independent of SUMMER.

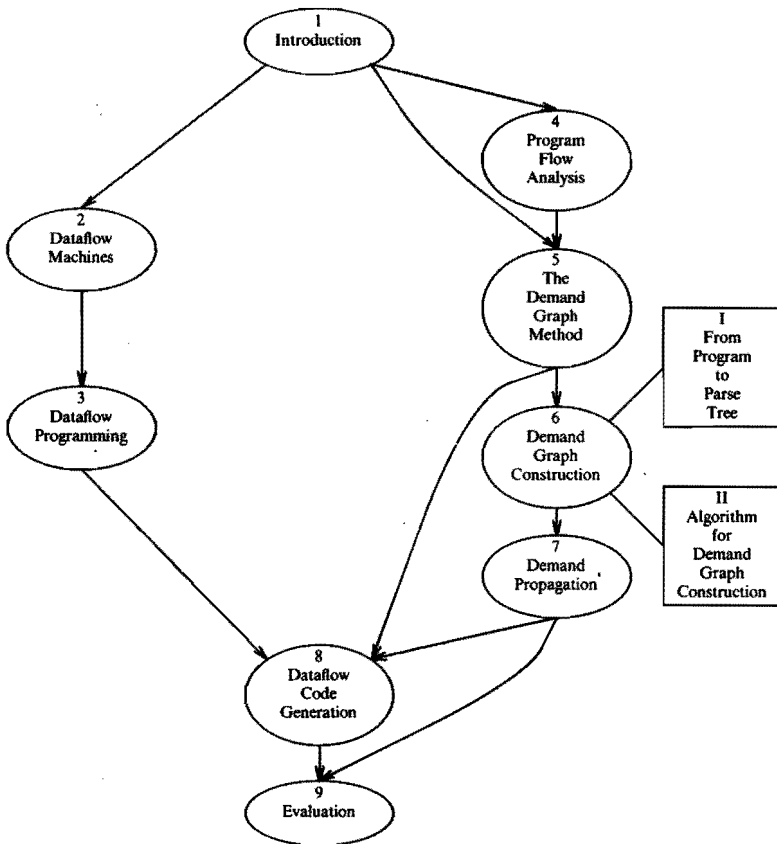


Figure 1.1. Dependency graph of the thesis.

Each ellipse stands for a chapter and each box for an appendix. The two chapters on dataflow can be read independently of the four chapters on flow analysis. Those readers only interested in the analysis method can skip chapters 2, 3, and 8. Readers that are mostly interested in dataflow code generation could skip chapters 4, 6, and 7.

Chapter 6 is the most technical one of the thesis; it contains a detailed description of the crucial part of the analysis method: the construction of the demand graph. It starts with a short description of SUMMER and then presents algorithms for the treatment of the language features for which analysis has been implemented. Much attention is given to the integrated treatment of escapes and the efficient handling of aliases. Aliasing can be dealt with quite easily, but could result in a large and therefore inefficient demand graph. Limiting the graph to a reasonable size is a complicated but interesting problem. The last section of this chapter describes the algorithm developed for this.

Chapter 7 gives examples of the application specific propagation of demands. The main application that is described is the one that performs static type checking. A simpler version of this application is included as part of the code generator.

Chapter 8 describes the generation of code for the Manchester Dataflow Machine. For most language features the translation from demand graph to dataflow graph is straightforward. Type analysis is needed to cater to the strong typing of the target

machine. Interesting issues are the implementation of *in situ* update for arrays and optimizations for loops that result in highly parallel code.

In chapter 9 the compiler is evaluated. The code the new compiler generates for several mini-programs is compared with that generated by an existing compiler for a dataflow language. This comparison shows that, at least for these small programs, there is not a significant difference in quality, neither in terms of efficiency nor parallelism. A discussion on the complexity of the new compiler estimates that it is comparable to that of a conventional optimizing compiler. Both results lend strong support to the hypothesis that an imperative language is a suitable source language for a dataflow machine.

Chapter 2

Dataflow Machines

Early advocates of data-driven parallel computers had grand visions of plentiful computing power provided by machines that were based on simple architectural principles and that were easy to program, maintain, and extend. Experimental dataflow machines have now been around for almost a decade, but still there is no consensus whether data-driven execution, besides being intuitively appealing, is also a viable means to make these visions become reality.

To facilitate the continuing debate, this chapter provides an introduction to dataflow machines and their underlying principles. No familiarity with parallel computers or graph terminology is assumed. The first section places dataflow machines in the context of other parallel computers. The next two sections introduce dataflow graphs, describe the execution of a program on a dataflow machine, and discuss different types of machine organizations. Section 2.4 presents a comparative survey of a wide variety of machine proposals and is followed by detailed study of one, operational, prototype. The concluding section discusses the feasibility of the dataflow concept on the basis of this prototype.

2.1. Parallel Computers

The term parallel computers could be somewhat misleading, since it suggests a monopoly on the exploitation of parallelism. However, Babbage's design for his analytical engine called for arithmetic to be performed on fifty digits in parallel [Hock81], the ENIAC also added the ten digits of its numbers in parallel [Gold72], and nearly all computers built since used parallelism in one form or another to speed up operation. As pointed out by Hockney [Hock81], the speed of computers has increased by roughly five orders of magnitude in the period between 1950 and 1975; three orders of magnitude are attributable to an increase in speed of the basic components while the rest of the speed-up is due chiefly to the introduction of parallelism.

Most of the parallel features were pioneered in "supercomputers", i.e. machines that were designed to be the most powerful that were available at the time. In the early fifties the overlapping of I/O operations with computation and even some primitive

form of vector processing were introduced; the ACE computer, which became operational in 1951, was the first. About a decade later parallel features like pipelining, instruction look ahead, cache memory, and memory interleaving were pioneered in the design of the ATLAS and the STRETCH computer. Almost all computers perform their arithmetic in parallel except the ones that were built just after the introduction of the fast but expensive electronic valve. Although most of these forms of parallelism are commonplace today even in computers with moderate performance, the term parallel computer is reserved for a machine in which parallel features are prominently visible at the machine language level.

The integration of more and more components onto a single chip makes parallel computers more attractive, and the availability of VLSI technology has spurred a renewed interest in this field. In principle, cheap processing power in VLSI form makes it possible to build a very fast parallel supercomputer, which would hitherto have been unaffordable. But VLSI makes parallelism attractive even for medium performance machines. The reasons for this are mostly economic. A higher level of integration leads to more computing power per dollar, since it rapidly decreases the manufacturing cost per gate but not the design cost of each unique part. This ever increasing ratio between design and manufacturing costs has a profound influence on systems architecture. It is most cost effective to design parts which are replicated many times (amortizing the design costs). Memory, in which one design is replicated billions of times, is the driving force behind the integration efforts. Popular microprocessors, which are both cheap and universal, follow in their wave. Machines with a much less wide appeal, such as high or medium performance machines, can only take full advantage of VLSI if design costs can be amortized internally: such machines should contain a few different parts that are simple and that are replicated many times. Because the parts have to be simple, concurrency is the only hope to achieve high performance.

The efficiency of a parallel computer is influenced by several conflicting factors. A major problem is *contention* for a shared resource, usually shared memory or some other communication channel. If during a significant part of a computation, a major part of the processing power is not engaged in useful computation we speak of *under-utilization*. If under-utilization is due to contention for a particular resource, then this resource will be called a *bottleneck*. The severity of bottlenecks can often be reduced by careful coordination, allocation, and scheduling, but if this is done at run-time it increases the *overhead* due to parallelism, i.e. processing that would be unnecessary without parallelism. Next to speed the most important quality of a parallel computer is its *effective utilization*, i.e. utilization corrected for overhead. The best one can hope for is that the effective utilization of a parallel computer approaches that of a well-designed sequential computer. Another desirable quality is *extensibility*, i.e. the property that the performance of the machine can always be improved by adding more processing elements. We speak of *linear speed-up* (and excellent extensibility) if the utilization does not drop when the machine is extended.

Some parallel computers are asynchronous at the level of the machine language: as long as two concurrent computations are independent, no assumptions can be made about their relative timing. These we will call *asynchronous* machines; the term refers to the architecture and does not imply that the organization of the machine is also asynchronous. In the programming of *synchronous* parallel computers the timing of concurrent computations plays a prominent role. They require skillful programming to bring utilization to an acceptable level since scheduling and allocation, i.e. deciding when and where a computation will be executed, has to be done by the programmer.

For certain kinds of applications this is quite feasible. For instance in low level signal processing massive amounts of data have to be processed in exactly the same way: the algorithms exhibit a high degree of regular parallelism. Various parallel computers have been successfully employed for these kind of applications.

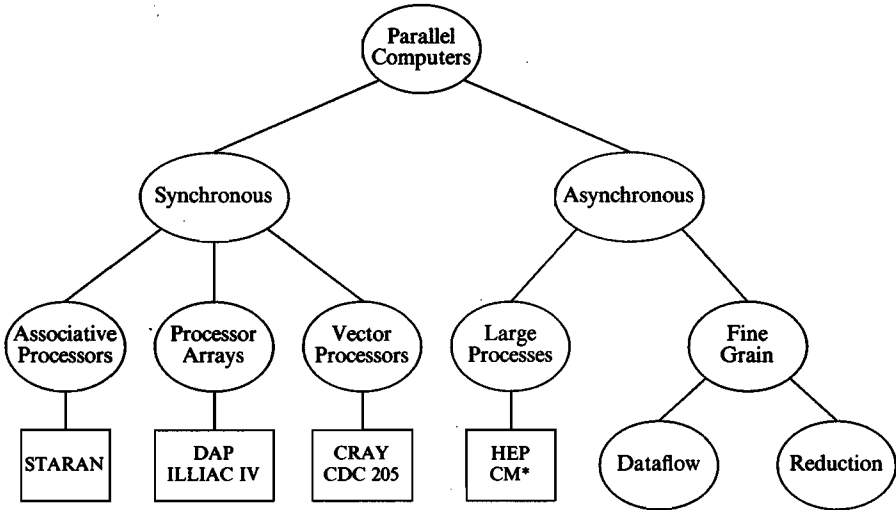


Figure 2.1. Some of the design options for parallel computers.

The distinction between synchronous and asynchronous corresponds to the classic distinction between SIMD (Single Instruction Multiple Data stream) and MIMD (Multiple Instruction Multiple Data stream), but is somewhat more informative. If the parallel operations are synchronized at the machine language level, scheduling and allocation needs to be done by the programmer. In asynchronous machines the processes that run in parallel need to be synchronized whenever they communicate with each other.

Synchronous parallel computers show a great variety in the power of individual processors and in the access paths between processors and memory. In associative processors (e.g. STARAN) many primitive processing elements are directly connected to their own data; those processing elements that are active in a given cycle all execute the same instruction. Contention is thus minimized at the cost of low utilization. Achieving a reasonable utilization is also problematic for processor arrays such as ILLIAC IV, DAP, and PEPE. The most popular of today's supercomputers are pipelined vector processors, such as the CRAY-1s and the CDC 205. These machines attain their speed through a combination of fast technology and strong reliance on pipelining geared towards floating point arithmetic on long vectors. The performance of vector processors is highly dependent on the algorithms used and especially on the access patterns to data structures. The reason for this is the large discrepancy between the performance of the machine when it is doing what it is designed to do, i.e. processing vectors of the right size, and when it is doing something else; the speed of scalar and vector operations differ more than an order of magnitude.

In many areas that have great needs for processing power, the behavior of algorithms is irregular and highly dependent on the input data making it necessary to perform scheduling at run time. This calls for asynchronous machines in which computations are free to follow their own instruction stream without interference from other computations. However, computations are seldom completely independent and at the points where interaction occurs they need to be synchronized by some special

mechanism. This synchronization overhead is the price to be paid for the higher utilization allowed by asynchronous operation.

There are different strategies to keep this price to an acceptable level. One is to keep the communication between computations to a minimum by dividing the task into large processes that operate mainly on their own private data, such as in the HEP [Smit78] or the CM* [Swan77]. Although in such machines scheduling is done at run time, the programmer has to be aware of segmentation, i.e. the partitioning of program and data into separate processes. Again the difficulty of this task is highly dependent on the regularity of the algorithm. Extension of the machine is not easy, since it requires the program to be repartitioned differently. Another problem is that processes may have to be suspended, leading to complications such as process swapping and the possibility of deadlock.

A different strategy to minimize synchronization overhead is to make communication simple and cheap, by providing special hardware and coding the program in a special format. Examples are reduction and dataflow machines. Because communication is so cheap, the processes can be made very small; about the size of a single instruction in a conventional computer. This makes segmentation trivial and improves extensibility, since the programs are effectively divided into many processes and special hardware determines which of them can execute concurrently.

In dataflow machines scheduling is based on availability of data; this is called *data-driven* execution. In reduction machines scheduling is based on the need for data; this is known as *demand-driven* execution. Demand-driven machines are currently under extensive study. There are close parallels between dataflow machines and reduction machines, but the relative merits of each type remain unclear. Most of the crucial implementation problems are probably shared by both types of machines. See [Trel82b] for a comparative survey.

2.2. Dataflow Machine Language

Although each dataflow machine has a different machine language, they are all based on the same principles. These shared principles are treated in this section. Because we are concerned with a wide variety of machines, we often have to be somewhat imprecise. More specific information is provided in section 2.5, which deals with one particular machine. We start with a description of dataflow programs and the ways they differ from conventional programs. Dataflow programs are usually presented in the form of a graph; a short summary of the terminology of dataflow graphs is given. The rest of this section shows how these graphs can be used to specify a computation.

DATAFLOW PROGRAMS

In most dataflow machines the programs are stored in an unconventional form called a *dataflow program*. Although a dataflow program does not differ much from a control flow program it nevertheless calls for a completely different machine organization. Figure 2.2 serves to illustrate the difference. A control flow program contains two kinds of references: those pointing to instructions and those pointing to data. The first kind indicates control flow and the second kind organizes data flow. The coordination of data and control flow creates only minor problems in sequential processing (e.g. reference to an uninitialized variable), but becomes a major issue in parallel processing. In particular when the processors work asynchronously, references to shared memory must be carefully coordinated. Dataflow machines use a different coordination scheme called *data-driven execution*: the arrival of a data item serves as the signal that may enable the execution of an instruction, obviating the need for separate control flow arcs.

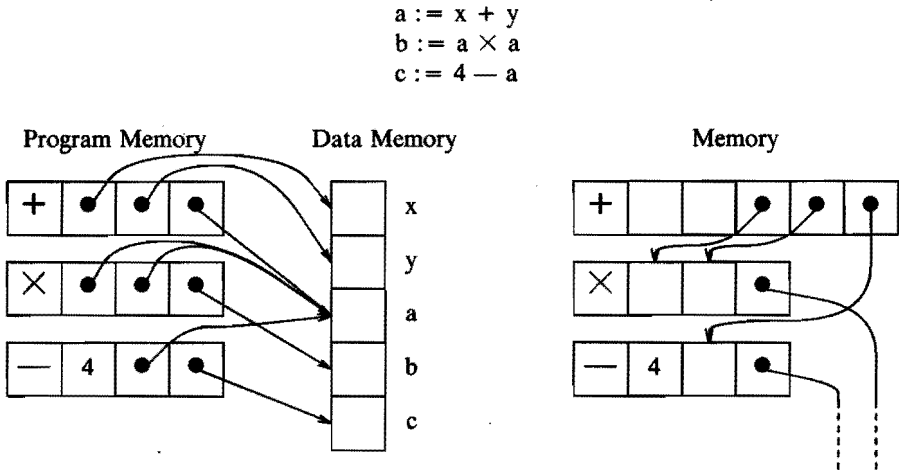


Figure 2.2. A comparison of control flow and dataflow programs.

On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are not shown. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

In dataflow machines each instruction is considered to be a separate process. To facilitate data-driven execution each instruction that produces a value contains pointers to all its consumers. Since an instruction in such a *dataflow program* contains only references to other instructions, it can be viewed as a node in a graph; the dataflow program in figure 2.2 is therefore often represented as in figure 2.3. In this notation, referred to as a *dataflow graph*, each node with its associated constants and its outgoing arcs corresponds to one instruction.

Because the control flow arcs have been eliminated, the problem of synchronizing data and control flow has disappeared. This is the main reason why dataflow programs are well suited for parallel processing. In a dataflow graph without cycles the arcs between the instructions directly reflect the partial ordering imposed by their data dependencies, which would have to be extracted by analysis if a control flow representation were used. Instructions between which there is no path in the dataflow graph can safely be executed concurrently.

DATAFLOW GRAPHS

The prevalent description of dataflow programs as graphs has led to a characteristic and sometimes confusing terminology stemming from Petri net and graph theory. Instructions are known as *nodes*, and instead of data items one talks of *tokens*. A producing node is connected to a consuming node by an *arc*, and the "point" where an arc enters a node is called an *input port*. The execution of an instruction is called the *firing* of a node. This can only occur if the node is *enabled*, which is determined by the *enabling rule*. Usually a *strict* enabling rule is specified, which states that a node is enabled when each input port contains a token. In the examples in this section all nodes are strict unless noted otherwise. When a node fires it removes one token from each input port and places at most one token on each of its output arcs. In so called queued architectures, arcs behave like FIFO queues. In most machines each port acts as a bag: the tokens present at a port can be absorbed in any order.

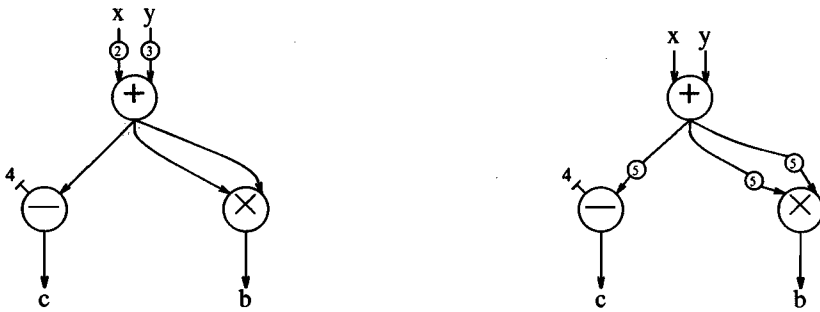


Figure 2.3. The dataflow program of figure 2.2 depicted as a graph.

The small circles indicate tokens. The symbol at the left input of the subtraction node indicates a constant input. In the situation depicted on the left the first node is enabled, since a token is present on each of its input ports. The graph on the right depicts the situation after the firing of that node.

Figure 2.3 serves to illustrate these notions. It shows an acyclic graph comprising three nodes, with a token present in each of the two input ports of the PLUS node (marked with the operator "+"). This node is therefore enabled and it will fire at some unspecified time. Firing involves the removal of the two input tokens, the computation of the result, and the production of three identical tokens on the input ports of the other two nodes. Both of these nodes are then enabled and they may fire in any order or concurrently. Note that, on the average, a node that produces more tokens than it absorbs increases the level of concurrency. All three nodes in this example are *functional*, i.e. the value of their output tokens is fully determined by the node descriptions and the values of their input tokens. A more formal treatment of these notions can be found in [Veen81].

CONDITIONAL CONSTRUCTS

Conditional execution and repetition require nodes that implement controlled branching. The conditional jump of a control flow program is represented in a dataflow graph by BRANCH nodes. The most common form is the one depicted in figure 2.4.



Figure 2.4. BRANCH and MERGE nodes.

A BRANCH node on the left and a non-deterministic MERGE node on the right.

A copy of the token absorbed from the *value* port is placed on the *true* or on the *false* output arc depending on the value of the control token. Variations of this node with more than two alternative output arcs or with more than one *value* port (compound BRANCH) have also been proposed. As we shall see shortly, the complement of the BRANCH node is also needed. Such a MERGE node does not have a strict enabling rule, i.e. not all input ports have to contain a token before the node can fire. In the

deterministic variety the value of a control token determines from which of the two input ports a token is absorbed. A copy of the absorbed token is sent to the output arc. The non-deterministic MERGE node (i.e. a MERGE node without control input) is enabled as soon as one of its input ports contains a token; when it fires it simply copies the token that it receives to its successors. This is equivalent to allowing more than one arc to end at the same port. If such *knots* [Veen81] are allowed, MERGE nodes can be abolished, with the advantage that a strict enabling rule is all that has to be supported.

Figure 2.5 shows an implementation of a conditional construct. If one token enters at each of the three arcs at the top of the graph, the two BRANCH nodes will each send a token to subgraph *f* or to subgraph *g*. Only the activated subgraph will eventually send a token to the MERGE node. If certain assumptions are made about the two subgraphs, it can easily be shown that this graph has the property that when one token is placed on each input arc, exactly one token is produced on the output arc. Furthermore, no port will ever contain more than one token. Such a graph is called *safe*. It ensures deterministic behavior even in the presence of non-deterministic MERGE nodes.

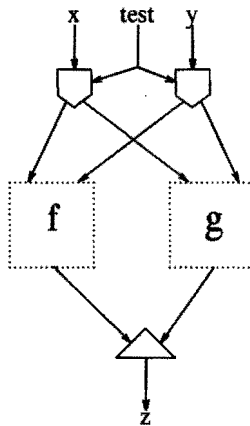


Figure 2.5. Conditional expression.

The graph corresponding to the expression $z := \text{if test then } f(x,y) \text{ else } g(x,y) \text{ fi}$. If *test* succeeds, both BRANCH nodes will send a token to the left, otherwise the tokens will go to the right. Note the use of the non-deterministic MERGE node.

Figure 2.6 shows a number of problems that may arise when BRANCH and (non-deterministic) MERGE nodes are used in an improper manner. All nodes in this figure are strict, except the MERGE nodes, and produce tokens on all output arcs when they fire, except the BRANCH nodes. The first graph is unsafe. If a pair of tokens arrives at the input ports of node A, the node is enabled and will fire, but this will not enable node B, since it receives only one token on one of its input ports. A new token may end up at the same port, if a second pair of tokens enters the graph. The second graph is also unsafe. When a token enters the graph, node A will fire and place a token on each of the input ports of the MERGE node. This node will then send two tokens to its output arc. In the third graph a token will be left behind at an input port of either node C or node D depending on the value of the control token of the BRANCH node. Such a graph is called *unclean*.

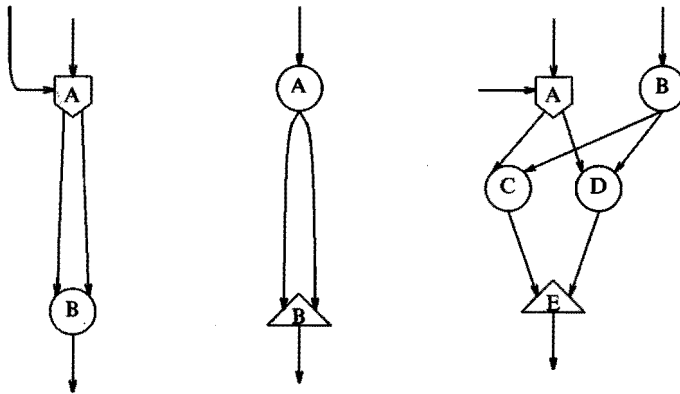


Figure 2.6. Problems resulting from the improper use of BRANCH and MERGE nodes. The first two graphs are unsafe; the third one is unclear.

ITERATIVE CONSTRUCTS AND REENTRANCY

Figure 2.7 illustrates problems that may arise when the graph contains a cycle. The simple graph on the left will *deadlock* unless it is possible to initialize the graph with a token on the feedback arc. Such an initial placement of tokens is known as *priming* the graph. The graph on the right is unsafe since after the firing of the node two tokens will be present on its input port. Although these are not realistic graphs, the same problems may arise in any cyclic graph unless special precautions are taken.

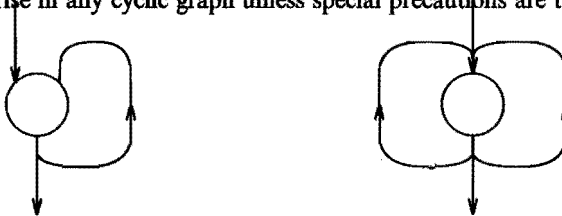


Figure 2.7. Problems with cyclic graphs. The graph on the left will deadlock, the one on the right is unsafe.

A correct way to implement a loop construct is shown in figure 2.8. Note the use of a compound BRANCH node rather than a series of simple BRANCH nodes as in figure 2.5. The strict enabling rule of this node ensures that it does not fire before subgraph *g* is free of tokens. Tokens for the next iteration can therefore be safely sent into the same subgraph. Because the nodes in subgraph *g* can fire repeatedly, it is an example of a *reentrant* graph. The way reentrancy is handled is a key issue in dataflow architecture. A dataflow graph is attractive as a machine language for a parallel machine, since all nodes that are not data dependent can fire concurrently. In case of reentrancy, however, this maximum concurrency can lead to non-deterministic behavior unless special measures are taken.

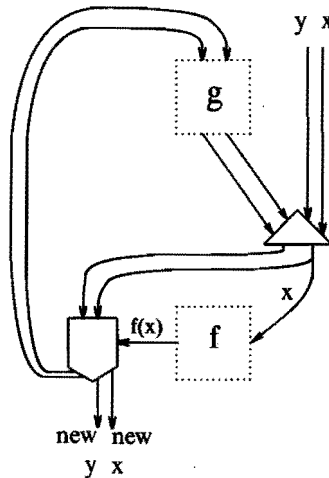


Figure 2.8. A loop construct according to the lock method.

An implementation of the expression **while** $f(x)$ **do** $(x,y) := g(x,y)$ **od**, using the lock method to protect the reentrant subgraphs f and g .

A graph in which reentrancy can lead to non-determinism is illustrated in figure 2.9, where the cycles for x and y lead through separate MERGE and BRANCH nodes. In the first iteration the first PLUS node will calculate the value for x and send copies to subgraph h and to one of the MERGE nodes. Subgraph h may postpone the absorption of its input token. Meanwhile the nodes on the cycle for x may fire again and the PLUS node may send a second token to subgraph h . The use of the compound BRANCH node in figure 2.8 is therefore essential for its safety. This method we will call the *lock* method. It is safe and simple, but not very attractive for parallel machines: the level of concurrency is low, since the BRANCH node acts as a lock that prevents the initiation of a new iteration before the previous one has been concluded.

An alternative approach is the *acknowledge* method. One way to implement this method is to add extra acknowledge arcs from consuming to producing node. These acknowledge arcs ensure that no arc will ever contain more than one token and the graph is therefore safe. One arc provides space for one token. In a manner too complicated to show here, the proper addition of dummy nodes and arcs can transform a reentrant graph into an equivalent one allowing overlap of consecutive iterations in a pipelined fashion. The acknowledge method therefore allows more concurrency than the lock method, but at the cost of at least doubling the number of arcs and tokens. Through proper analysis, however, a substantial part of these arcs can be eliminated without impairing the safety of the graph [Mont80, Broc79]. Both of these methods can also be implemented at the architecture level by modifying the enabling rule. In some machines locking is implemented by specifying that nodes in a reentrant subgraph can only be enabled a second time after all tokens of a previous activation have left the subgraph. The architectures of other machines implement acknowledgement by enabling a node only after all its output arcs are empty.

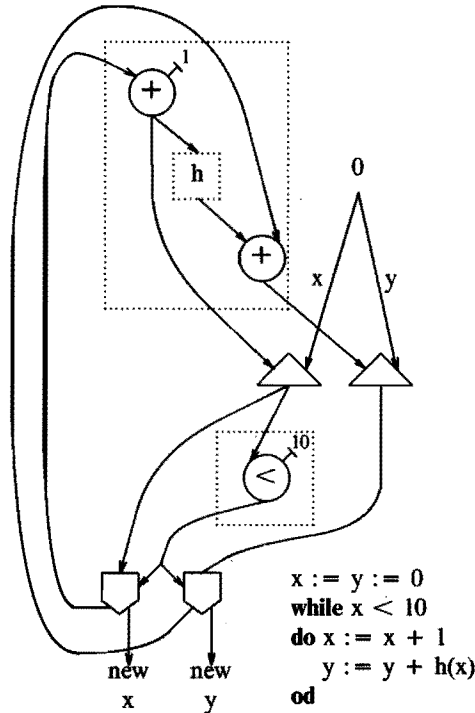


Figure 2.9. An unsafe way to implement a loop.

A new token may arrive at the input of subgraph h before the previous one is absorbed.

A much higher level of concurrency is obtained when each iteration is executed in a separate instance (or copy) of the reentrant subgraph. This *copy* method requires a machine with facilities to create a new instance of a subgraph and to direct tokens to the appropriate instance. A more efficient way to implement the copy method is to share the node descriptions between the different instances of a graph without confusing tokens that belong to separate instances. This is accomplished by attaching a *tag* to each token that identifies the instance of the node it is directed to. These so called tagged architectures have an enabling rule that states that a node is enabled if each input arc contains a token *with identical tags*. Safety in these machines means that no port will ever contain more than one token with the same tag. A tag is sometimes referred to as a *color* or a *label*.

The tagged nature of the architecture shows up in the program in the form of nodes that modify tags. Figure 2.10 shows the implementation of the example of figure 2.9 on a tagged architecture. The proper execution of nested loops requires that the tags used within a loop are distinct from those in the surrounding expression. A new area in the tag space is therefore allocated at the start of the loop; within the area tags are ordered. Tokens entering the loop receive the first tag and tokens for consecutive iterations receive consecutively ordered tags within the allocated area. On tokens that exit the loop, the tag corresponding to the surrounding expression is restored. This method can lead to a much higher level of concurrency, because the cycle for x can safely send a whole series of tokens with different tags into subgraph h , with each token initiating a separate and possibly concurrent execution of h .

0

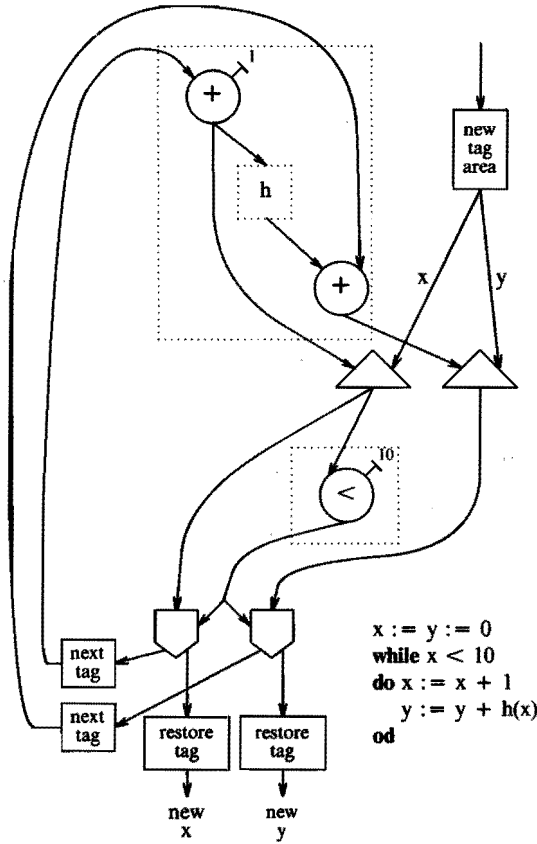


Figure 2.10. An implementation of a loop using tags.

At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations receive consecutive tags within this area. On tokens that exit from the loop the tag from before the loop is restored; this operation requires an extra input arc that has been omitted from the illustration.

PROCEDURE INVOCATION

The invocation of a procedure introduces similar problems with reentrancy, to which the methods described above can also be applied. An extra facility is required to direct the output tokens of the procedure activation back to the proper calling site. This is usually implemented as shown in figure 2.11. A token is sent into the procedure body that contains a reference to a node at the calling site. This token is then used by the output nodes of the procedure body to direct the return values to the proper places. Since these output nodes can thus send tokens to nodes to which they have no static arc, these are known as *dynamic* nodes.

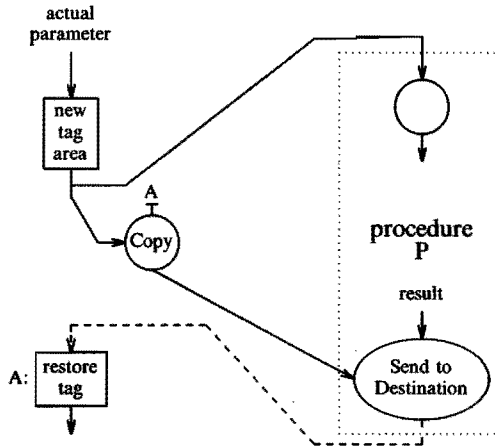


Figure 2.11. Use of dynamic nodes to return procedure results.

On the left a call of procedure *P* whose graph is on the right. *P* has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of *P* and concurrently a token containing address *A* is sent to a node with a dynamic output arc. This SEND-TO-DESTINATION node transmits its first input token to a node of which the address is contained in the second token. The effect is that, when the return value of the procedure becomes available, the dynamic node sends the result to node *A*, which restores the tag belonging to the calling expression.

2.3. The Architecture of Dataflow Machines

This section describes dataflow machines at the level that directly supports the machine language. First the basic execution mechanism of a processing element is described and then the overall structure of a dataflow multiprocessor.

A PROCESSING ELEMENT

A typical dataflow machine consists of a number of processing elements, which can communicate with each other. Figure 2.12 shows a functional diagram of a processing element.

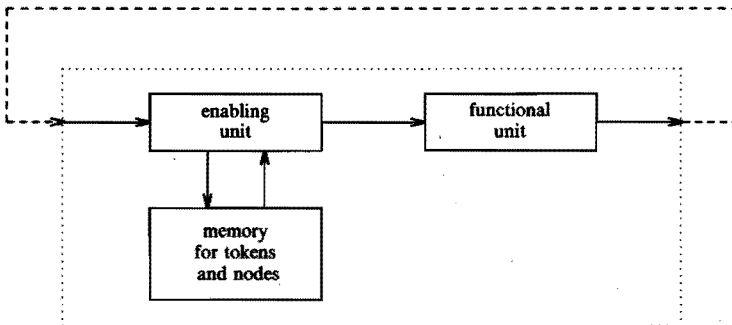


Figure 2.12. Functional diagram of a processing element.

The enabling unit accepts tokens from the left and stores them at the addressed node. If this node is enabled, an executable packet is sent to the functional unit where it is processed. The output tokens, with the destination addresses, are sent back to the enabling unit. Modules dedicated to buffering or communication have been left out of this diagram.

The nodes of the dataflow program are often stored in the form of a *template* containing a description of the node and space for input tokens. The node description consists of the operand-code (a shorthand for the mapping from input values to output values) and a list of destination addresses (the outgoing arcs). We can think of the movement of a token between two nodes as the progress of a locus of activity. A node that produces more tokens than it consumes increases the number of concurrent activities. Concurrent activities interact at nodes that consume more than one token. Coordination has to take place at these nodes. In dataflow machines coordination therefore amounts to the administration of the enabling rule for those nodes that require more than one input. The unit that manages the storage of the tokens we call the *enabling unit*. It sequentially accepts a token and stores it in memory. If this causes the node to which the token is addressed to become enabled (i.e. each input port contains a token), its input tokens are extracted from memory and, together with a copy of the node, formed into a packet and sent to the functional unit. Such an *executable packet* consists of the values of the input tokens, the operand-code and a list of destinations. The *functional unit* computes the output values and combines them with the destination addresses into tokens. Tokens are sent back to the enabling unit, where they may enable another node. Since the enabling and the functional stage work concurrently, this is often referred to as the circular pipeline.

Dividing a processing element into two stages is just one of the possibilities. In some machines the processing elements do not have to be so powerful and they just consist of a memory connected to a unit that handles both token storage and the execution of nodes. In other machines the circular pipeline consists of more concurrent stages, as for instance in most machines that use the tag method to protect reentrant code. Since in such a machine nodes are shared between different instances of a graph, the space in a template to be reserved for storage of input tokens may become arbitrarily large. This makes it impractical to store tokens in the nodes themselves. Token storage is therefore separated from node storage and the enabling unit is split into two stages: the *matching unit* and the *fetching unit*, usually arranged as shown in figure 2.13.

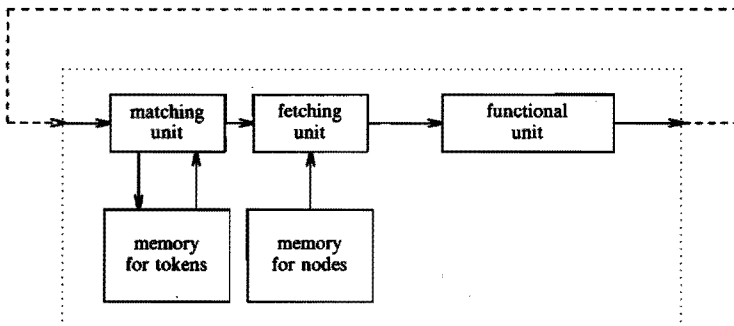


Figure 2.13. Functional diagram of a processing element of a tagged machine.

The matching unit stores tokens in its memory and checks whether an instance of the destination node is enabled. This requires a match of both destination address and tag. Tokens are stored in the memory connected to the matching unit. When all tokens for a particular instance of a node have arrived, they are sent to the fetching unit, which combines them with a copy of the node description into an executable packet to be passed on to the functional unit.

For each token that the matching unit accepts, it has to check whether the addressed node is enabled. In most tagged machines this is facilitated by limiting the number of input arcs to 2 and providing each token with an extra bit that indicates the number of tokens the addressed node requires. The matching unit only has to check whether its memory already contains a matching token, i.e. a token with the same destination and tag. Conceptually, the matching unit simply combines destination and tag into an address and checks whether the location denoted by the address contains a token. The set of locations addressed by tag and destination forms a space that we call the *matching space*. Managing this space and representing it in a physical memory is one of the key problems in tagged dataflow architectures.

Although not apparent at first, the problem of matching space management is quite similar to the problems encountered in code copying machines and in fact involves problems that have plagued parallel architectures from the beginning. At the entrance to a loop, and during procedure invocation, a unique tag area has to be allocated. Guaranteeing uniqueness in a parallel computer is problematic. The fundamental trade-off is between the bottleneck created by a centralized approach and the communication overhead or inefficient use of space offered by a distributed approach. In [Arvi77] an extremely distributed approach is proposed in which the uniqueness of a new tag area can be deduced from the existing tag. Since a tag in this scheme effectively encodes the calling stack of a procedure invocation, its size grows linearly with calling depth. Many partly distributed solutions have been proposed. They all amount to statically distributing the matching space over a set of managers, each of which manages the allocated area locally. An example is a centralized counter per processing element, which together with a unique identification of the processing element provides a unique tag. To prevent the local areas from becoming exhausted the matching space must be large and, consequently, at any given time sparsely occupied. Large sparsely occupied spaces cause several problems. Firstly, addressing an item requires many bits. Secondly, implementing the space involves a difficult trade-off between storage waste (e.g. a sparsely occupied array) and access time overhead (e.g. a linked list). Hashing techniques offer a compromise. Actual implementations of the approaches just described are far too few to come to any conclusion yet.

It is interesting to note that the trade-offs for code copying machines are virtually identical. When a copy of a subgraph needs to be created a storage area has to be allocated. A virtual memory scheme with sparse allocation can be used, but addresses become large and an efficient mapping to physical memory is needed. Paging techniques that exploit locality in instruction execution may be useful. A good memory manager would avoid these problems but has the same drawbacks as described above. Efficient distributed allocators and resource managers should therefore be a focal point of dataflow research. The applicability of mechanisms that have been developed to solve similar problems in sequential machines (cache, virtual memory management) should also be studied.

In one variety of dataflow machines each node that fires has been loaded into the memory of a processing element before the computation starts. Nodes are statically allocated not only to a processing element but also to a physical memory address. In these so called *static* machines destination addresses are fixed before the computation starts and do not have to be calculated dynamically. These machines do not support concurrent execution of a loop or procedure body. Such concurrency requires facilities to implement the copy or the tag method. Machines of this type are called *dynamic*.¹

1. This is not related to the concept of dynamic nodes and arcs described previously.

Static machines are much simpler than dynamic machines, since they do not need mechanisms for copying or matching of token tags, but for most algorithms their effective concurrency is lower. Algorithms with a predominantly pipelining type of parallelism, however, execute efficiently on static machines with acknowledging.

DATAFLOW MULTIPROCESSORS

Figure 2.14 shows a schematic view of the structure of a complete dataflow machine. Although each description of a dataflow machine in the literature seemingly presents a different picture, most designs conform to one of the three structures illustrated.

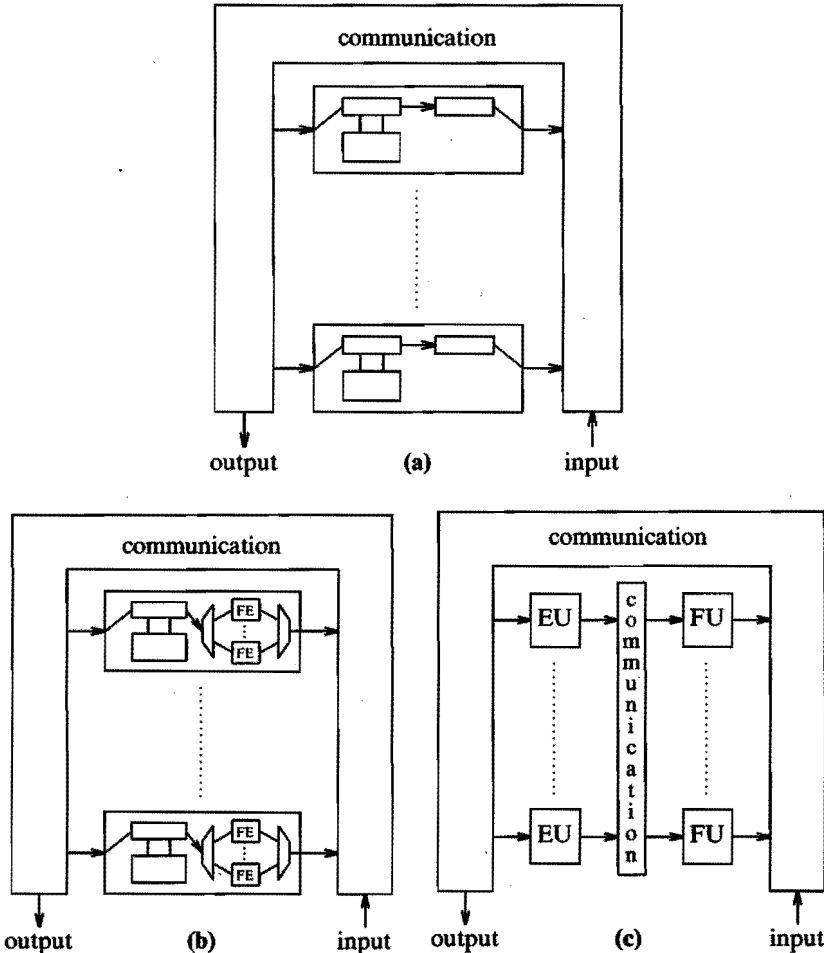


Figure 2.14. Overall structure of various dataflow multiprocessors.

- (a) One level dataflow machine. Communication facilities deliver tokens that are produced by a functional unit to the enabling unit of the correct processing element, as determined by the destination address and the allocation policy.
- (b) Two level dataflow machine. Each functional unit consists of several functional elements (FE), which concurrently process executable packets.
- (c) Two stage dataflow machine. Each enabling unit (EU) can send executable packets to each functional unit (FU).

In a *one level* dataflow machine there is only one level of concurrency in the execution of instructions. Instructions are executed in the processing elements and the resulting tokens are used in the same processing element or communicated to other processing elements. The other two structures exploit the fact that the processing of executable packets is independent and can be done in any order or concurrently, since they contain all the information that the functional unit needs to fire the node and to construct the output tokens. In a *two level* machine each functional unit consists of many *functional elements*, which process executable packets concurrently. Scheduling is trivial: an executable packet is allocated to any idle functional element. By adjusting the number of functional elements the power of the functional unit can be tuned to that of the rest of the processing element. In a *two stage* machine the processing elements are split into two stages and between the two stages there is an extra communication medium that sends executable packets to functional elements. This two stage structure is advantageous if the functional stage is heterogeneous, for instance when some functional elements have specialized capabilities.

COMMUNICATION

Figure 2.14 is merely intended to indicate that there is a way to communicate between different processing elements without suggesting any particular topology. In an actual machine the communication medium can have the structure of a tree, a ring, a binary n -cube, or an equidistant $n \times n$ switch. An even more important difference lies in the nature of the connections that the communication medium provides. Just as there are circuit switching and packet switching networks, a dataflow machine can have a direct communication or a packet communication architecture.

In *direct communication* machines adjacent nodes in the graph are allocated to processing elements that have a direct connection with each other. An important property of a direct communication architecture is that the communication medium delivers tokens in the same order as they were received. If the communication medium is equipped with queues, unsafe graphs (dataflow graphs in which arcs can contain more than one token) can be executed without impairing determinism.

Packet communication offers the greatest opportunity for load distribution and parallelism in the communication unit, since it can be constructed from asynchronously operating packet switching modules, with parallelism and redundancy in this critical resource. Such a module can accept a token and forward it to another module depending on its destination address. The order of packets is not necessarily maintained, and consequently the arcs of the graph do not behave as FIFO queues. Deterministic execution on these machines can therefore only be guaranteed for safe graphs. The best structure for the communication unit and its limitations in size and performance are a matter of debate among dataflow architects. One approach is to have a large number of slow and simple processing elements connected to a high bandwidth communication unit. A one level machine structure is usually appropriate for this approach. Other architects claim that as soon as the machine contains more than a few dozen processing elements, insurmountable bottlenecks in the communication unit are created. They therefore concentrate on the construction of powerful processing elements, which almost always involves a two level design. These architects tend to postpone the design of the higher level until later, and sometimes one processing element is presented as a complete machine. The performance of one processing element, however, is limited by the inherent bottlenecks in the enabling section.

DATA STRUCTURES

In a dataflow graph values flow from one node to another and are, at least at that level of abstraction, not stored in memory. If a value is input to more than one node, a copy is sent to each node. Conceptually, data structures are treated in the same way as other values. A retrieve operation, for instance, consumes a complete structure and an index and produces a copy of the retrieved element. Directly implementing this concept is known as *copying*. Copying is appropriate for small structures. In a tagged machine with limited token size a complete structure can be sent to a node by packaging each element as a separate token distinguished by subsequent tags. Unfortunately, data structures tend to be large and implementing these by the conceptually simple copying method would place an unacceptable burden on the machine. Many machines therefore have a facility to *store* structures. In such machines an element can be retrieved by sending a request to the unit where the structure has been stored.

The dataflow equivalent of a selective update operation (changing one element of a structure) is an operation that consumes the old structure, the index, and the new value and produces a completely new structure. This involves the copying of structures even when they are stored. There are several ways to reduce excessive copying. Structures that are not shared do not have to be copied before an update. A reference count mechanism can be used to detect this, and is helpful for garbage collection. For shared structures copying can be further reduced by storing the structure in the form of a tree and copying only the updated node and its ancestors.

Another approach is to provide restrictive access primitives in the programming language. This led to the concept of *streams*, which are structures that can only be produced and consumed sequentially. These may be processed more efficiently in some machines and so increase the effective parallelism, because elements of a stream can be consumed before the stream is completed. This increase in parallelism can also be achieved by treating the structures *non-strictly*, i.e. allowing access to elements before the structure has been completely created.

2.4. A Survey of Dataflow Machines

This section presents a survey of most of the dataflow machines described in the literature. The extent of such a survey is not immediately clear, since there is no sharp definition of dataflow machines in the sense of a widely accepted set of criteria to distinguish dataflow machines from all other computers. For the sake of this survey we consider as dataflow machines all *programmable* computers of which the hardware is optimized for *fine grain data-driven* parallel computation. *Fine grain* means that the processes that run in parallel are approximately of the size of a conventional machine code instruction. *Data-driven* means that the activation of a process is solely determined by the availability of its input data. This definition excludes simulators as well as non-programmable machines, for instance those that implement the dataflow graph directly in hardware, an approach that is popular for the construction of dedicated asynchronous signal processors. We also exclude data driven computers that use coarse grain parallelism such as the MAUD system [Leco79], and computers that are not purely data-driven [Trel82a].

The concept of data-driven computation is as old as electronic computing. It is ironic that the same von Neumann, who is sometimes blamed for having created a bottleneck that dataflow architecture tries to remove, made extensive study of neural nets, which have a data-driven nature. Realization of such devices was not feasible at the time. Asynchronously operating I/O channels, introduced in the 1950's, which

communicate according to a ready/acknowledge protocol, are among the first implementations of data-driven execution. The development in the 1960's of multi-programmed operating systems, such as MULTICS, provided the first experience with the complexities of large scale asynchronous parallelism. The intractability of these systems has led to the emergence of new models for the design of parallel systems. After exposure to these problems in the MULTICS project [Denn69] Dennis at MIT developed the model of Dataflow Schemas, building on work by Karp&Miller [Karp66] and Rodriguez [Rodr69]. These dataflow graphs, as they were later called, evolved rapidly from a method for designing and verifying operating systems to a base language for a new architecture. The first designs for such machines [Denn74, Rumb75] were made at MIT. The first dataflow machine became operational in July 1976 [Davi79] and several have been built since.

A clear view of the common properties of different dataflow machines is sometimes obscured by trivial matters like differences in terminology, choice of illustrations, or emphasis. Comparisons of dataflow machines have appeared elsewhere, but they were mostly limited to a few machines [Denn80a, Hazr82]. A more extensive list can be found in [Trel82b]. An interesting comparison of machines for the execution of functional languages recently appeared in [Vegd84].

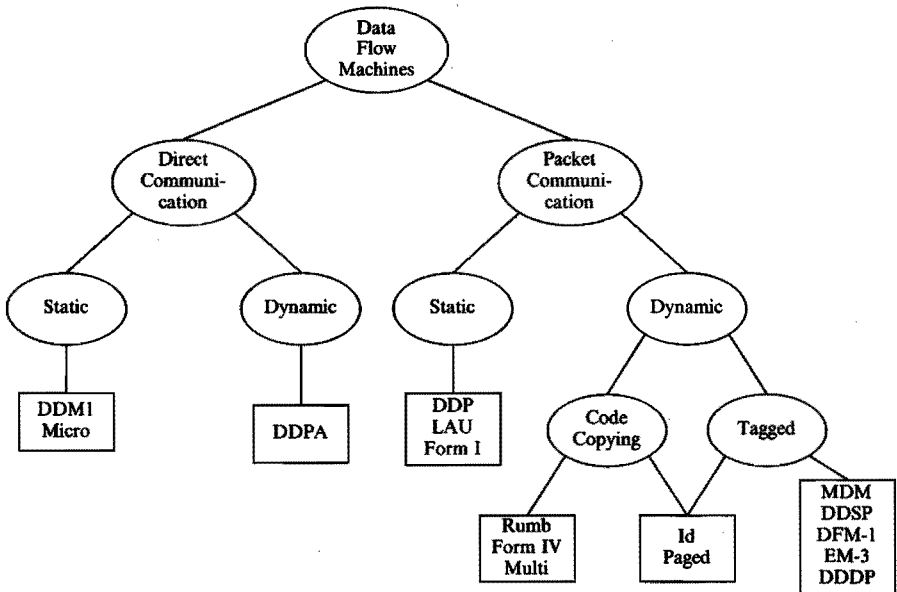


Figure 2.15. A survey of dataflow machines, categorized according to their architecture and implementation.

The keys in the boxes refer to the machines that are summarized in figure 2.16.

Figure 2.15 illustrates our classification of dataflow machines. Its form is chosen for reasons of clarity and gives an impression of which machines are most similar, although it does not do justice to all the important properties of a particular machine. The choice of properties used for the classification is limited by the fact that many descriptions (and some designs) are vague and incomplete. In figure 2.15 dataflow machines are categorized according to the nature of the communication unit and the architecture of the processing elements. The topology of the communication unit is not

used as a criterion, since it does not really help to characterize a dataflow machine and is often left unspecified. In the rest of this section all machines appearing in the figure are described separately, using the common terminology established in the previous two sections. A few features of some designs are summarized in the table at the end of this section.

Key	Machine	Group	Start Project	Operational
Direct Communication Machines				
DDM1	Data-Driven Machine #1	Davis, Utah	1972	1976
Micro	Micro-Programmed	Marczyński, Warsaw		
DDPA	Data-Driven Processor Array	Takahashi, Tokyo		1983
Static Packet Communication Machines				
DDP	Distributed Data Processor	Cornish, Texas Instruments	1976	1978
LAU	LAU System Prototype #0	Syre, Toulouse	1975	1980
Form I	Prototype Basic Dataflow Processor	Dennis, MIT	1971	1982
Dynamic Packet Communication Machines				
Rumb	Dataflow Multiprocessor	Rumbaugh, MIT	1974	-
Form IV	Dynamic Dataflow Processor	Misunas, MIT	1976	
Multi	Multi-User Dataflow Machine	Burkowski, Winnipeg		
Id	Id Machine	Arvind, MIT	1974	1984
Paged	Paged Memory Dataflow Machine	Caluwaerts, Leuven	1979	-
MDM	Manchester Dataflow Machine	Gurd&Watson, Manchester	1976	1981
DDSP	Data-Driven Signal Processor	Hogenauer, ESL		
DFM-1	List-processing-oriented Dataflow Machine	Amamiya, Tokyo	1980	1983
EM-3	ETL Data-Driven Machine-3	Yuba, ETL		1984
DDDP	Distributed Data-Driven Processor	Kishi, Tokyo		1982

Figure 2.16. A summary of the dataflow machines that are described in the text.

The dates are in most cases estimates and are merely meant as an indication of the relative chronology.

DIRECT COMMUNICATION MACHINES

The main drawback of direct communication machines is that for many graphs it is difficult to find a good mapping onto the network (*allocation*.) For applications that have predictable and regular communication patterns matching the machine's topology, this may be a fruitful approach, however. The most important member of this class is the oldest working dataflow machine, the DDM1 [Davi77, Davi79]. The processing elements of this machine are arranged as a tree. Allocation is simplified by preserving the hierarchical tree structure of the program. Any internal node of the processing tree can allocate a part of its program (a subtree) to any of its descendants. Allocation is simple and distributed, but far from optimal with respect to even load distribution over the processing elements. Another, less elaborate, example is provided by a machine developed in Warsaw, in which the processing elements receive the node descriptions in the form of micro-programs [Marc83].

In Japan an interesting dynamic direct communication machine has been developed for large scale scientific calculations, such as solving partial differential equations [Taka83]. The processing elements are arranged on a two-dimensional grid and use tags to distinguish tokens belonging to different activations. To avoid the necessity to allocate unique tag areas dynamically, the input language is somewhat restricted (no general recursion) so that static allocation is possible. A hardware simulator, consisting of 4×4 processing elements, each connected to 8 neighbors, has been used to study small applications. It confirmed analytical predictions that communication delay does not seriously degrade performance provided that programs have enough parallelism. A prototype is now under construction.

STATIC PACKET COMMUNICATION MACHINES

The first packet communication dataflow machine that became operational is the Distributed Data Processor [Corn79, John80], built at Texas Instruments. The references suggest that DDP uses a locking method to protect reentrant graphs. Although the compiler may create additional copies of a procedure to increase parallelism, this copying occurs statically. It is a one level machine with a ring structured communication unit, augmented with a direct feedback link for tokens that stay within the same processing element. A prototype comprising four processing elements has been built.

Around the same time the LAU project in Toulouse, France, designed another static dataflow machine [Syre80, Comt80, Syre77]. LAU stands for *Langage à assignation unique* (single assignment language). This group first designed a high-level language and then a machine for its efficient execution. The group concentrated on the construction of a powerful processing element and left the higher level structure more or less unspecified. In 1980 the LAU system prototype #0, a processing element with 32 functional elements, was completed. Most functional elements are built around a conventional micro-processor. The machine is not programmed by pure dataflow programs as described in section 2.2. There is a separate program and data memory and programs are represented as conventional control flow programs, in which control flow arcs have been replaced by additional pointers in data memory to all consuming instructions. This requires a multi-phase communication between functional unit and token memory and it also complicates the communication with other processing elements. Safety is guaranteed by a hardware supported locking mechanism. As in the DDP, the programmer can instruct the compiler to create copies of reentrant subgraphs to increase parallelism. The instruction set includes nodes that manage all copies of a subgraph and choose the copy to be used dynamically.

Dennis and his colleagues at MIT have been in the vanguard of the dataflow field and produced the first designs for dataflow machines. The earliest design [Denn74] had a two stage structure, with each enabling unit (called an instruction cell) dedicated to one node and with heterogeneous functional units. This design was later extended into a series of machines differing in the way they handled reentrancy and data structures. They ranged from the elementary *Form I* processor, which was static and could only handle elementary data, to the full fledged *Form IV* processor, which had extensive structure facilities and which could copy subgraphs on demand (see below). When it was discovered that an unsafe graph might deadlock the machine and acknowledge arcs had to be introduced, it became clear that it was wasteful to dedicate the processing power needed in one instruction cell just to one instruction. They were therefore shared between a group of nodes and called cell blocks. A prototype has now been built in which the different parts are emulated by micro-programmable micro-

processors [Denn80b]. Since this single unit can emulate both a cell block and a functional unit, the prototype has the single stage structure of figure 2.14. The prototype that is now operational consists of 8 processing elements and an equidistant packet routing network built from 2×2 routing elements.

MACHINES WITH CODE COPYING FACILITIES

The dataflow machines with potentially the highest parallelism are the dynamic dataflow machines; they employ either code copying or tags to protect reentrant graphs. It is characteristic for a code copying machine that it cannot always be determined statically what the physical address of a node will be. The first detailed design of a dataflow machine was of this type [Rumb75]. Allocation in this machine is per procedure: all the nodes and intermediate results of one procedure are stored in the memory of one processing element. There is a fast connection from the output to the input port of a processing element such that a circular pipeline is created. Tokens stay within this pipeline unless they are directed to another procedure, in which case they are routed to a special processing element called the scheduler. This scheduler sends a copy of the called procedure and its input values to an idle processing element. If there is no idle processing element, it waits until a processing element becomes dormant and then saves its state (i.e. all the unprocessed tokens) and declares it idle.

The MIT *Form IV dataflow processor* is not one machine, but refers to a whole family of designs: there have been a number of articles from the dataflow group at MIT each specifying part of a full fledged dataflow machine. They are all based on an extension of the basic architecture originally described by Dennis and Misunas [Denn74], but include special units to store data structures in the form of a tree using hardware supported reference counts. There have been different proposals for the handling of reentrancy. Misunas [Misu78] rejected locking and acknowledgement, because it limits parallelism and proposed to program the machine without iteration. Procedure bodies would be stored just like data structures and presumably the invocation of a procedure would result in the storing of a copy of the procedure in the cell blocks. Weng [Weng79] is more specific about this mechanism. Miranker [Mira77] suggests a sort of virtual memory for nodes. Translation from virtual to physical address is handled by a relocation box, which manages both the physical and the virtual space. A node is copied into physical memory when it receives its first token. A procedure call generates a unique suffix, which identifies a particular activation. The relocation mechanism ensures that all tokens in that invocation receive the same suffix. This is similar to the tag method. All nodes in a procedure are relocated, not only those that get executed. Code copying is needed because in all machines of this family tokens and nodes are stored together as templates.

A proposal that is surprisingly similar to this is presented by Burkowski [Burk81]. He produced a detailed hardware design for the static Form I processor, including the acknowledge scheme to protect reentrant graphs, but added memory management facilities, so that the machine can safely be shared between independent tasks. This feature makes it into a dynamic machine, since nodes can be allocated and removed under program control. Although this makes code copying at procedure invocation feasible, no reference to this can be found in the description.

MACHINES WITH BOTH TAG AND CODE COPYING FACILITIES

Arvind and Gostelow began their study of dataflow languages and architectures at the University of California, Irvine, almost a decade ago. They designed the language Id (Irvine Dataflow), which introduced many interesting concepts. Independently from similar work in Manchester, they developed the concept of tags (originally known as colors) and showed that it helped to extract more of the parallelism available in a dataflow graph [Arvi77]. Simulation studies were also carried out [Gost80]. The machine they designed has interesting data structure facilities implementing so called *I-structures* (for incomplete structures). These structures are non-strict: fetching of already written elements is allowed before the structure is complete. This increases the effective parallelism of a program and facilitates the asynchronous activation of parts of a procedure (i.e. non-strict procedure call). Special hardware is included to defer fetches of elements that are not yet available. Arvind and his group, now at MIT, are in the process of constructing a prototype. Original plans called for the implementation of the processing elements in VLSI, but this has been postponed until after the construction of a prototype comprising 64 Lisp machines. These machines will each emulate one processing element, and can communicate through a packet routing network consisting of 64 switching elements. The physical connections between these switching elements favor a binary 7-cube topology, but the network can be programmed to emulate other topologies. Since the paths between processing elements are unequal in length, with the path from a processing element back to itself the shortest, the allocation of nodes and structures can have a great influence on the performance of the machine. Since elaborate facilities are needed to make this allocation as flexible as possible, allocation of memory and of tags is under control of a software manager. An advantage of the combined managing of these two resources is that dynamic trade-off is possible. The tag space (limited by the maximum size of a tag) is kept small and is used rather densely. When the tag supply is exhausted, new copies of a subgraph are allocated.

In Leuven, Belgium, a machine has been designed, with an elaborate memory management scheme [Calu82]. Each processing element has its own memory manager, but they can also communicate with each other, so that the total memory space is shared. A procedure call results in the allocation of a fresh memory area for the tokens belonging to the new invocation. A pointer to this area serves as the tag. To facilitate an even load distribution the area is allocated in a neighboring processing element. Therefore, when a node is enabled, its description must be fetched from another processing element. Caches are used to create local copies. In fact memory is paged and complete pages are copied. An interesting feature of the memory system is that it treats data structures in the same way as programs, just as in the Form IV processor, and that they can be converted into each other. This makes the implementation of higher order functions feasible.

TAGGED MACHINES

The first tagged dataflow machine built was the Manchester Dataflow Machine [Gurd80, Wats82]. A prototype processing element became operational in 1981. This machine will be treated in detail in the next section.

A similar machine but optimized for signal processing is the Data Driven Signal Processor (DDSP) developed by ESL Inc [Hoge82], which can accommodate a maximum of 32 processing elements. The optimization is probably due to a special allocation algorithm combined with an unorthodox communication topology, that appears to be a combination of a ring and a tree.

In Japan several tagged dataflow machines are in various stages of construction. The machine constructed at the Electrical Communication Laboratory of NTT is optimized for list processing [Amam82]. The processing elements are divided into two classes: control modules, which provide storage for nodes and tokens, and structure memories, which provide storage for structures. Functional units are integrated with the structure memories rather than with the control modules, since most nodes are expected to operate on structures. The design is guided by the primitive operations available in pure Lisp and all structures are lists. The central structure operation *cons* is implemented "lenient": a pointer token is generated before its arguments are available. This provides the same advantages as other non-strict structures such as I-structures.

Non-strict data structures are also supported by the Electro Technical Data-Driven Machine-3 (EM-3), another LISP based machine [Yama83]. This non-strict mechanism is extended to increase the concurrency of a procedure call. At the start of a procedure invocation *pseudo-results* are sent to the consumers of the results of the procedure call. Concurrent with the execution of the procedure body, most nodes will process these pseudo-results just as if they were normal tokens. When a node requires the actual value, its execution is delayed until it becomes available. This mechanism seems to provide the same computational capability as lazy evaluation. A hardware prototype composed of 8 processing elements is under construction.

The Distributed Data-Driven Processor built at Systems Laboratory [Kish83] is distinguished by a centralized tag manager. Although this manager may introduce a bottleneck, it uses the tag space rather densely and simplifies the restoration of tags after a procedure invocation. Token matching is by means of a hardware hashing mechanism similar to the one described in the next section. The machine has a dedicated unit for non-strict structures. A prototype comprising 4 processing elements communicating through a two-way ring has been constructed. The study of simple hand-coded benchmarks revealed that simple allocation results in a reasonable utilization, which can be markedly improved by more sophisticated allocation schemes.

The table in figure 2.17 summarizes some of this information for the most important dynamic machines.

Feature	FormIV	Id	Paged	MDM	DFM-1	EM-3	DDDP
MS	2S	1L	O	2L	O	1L	1L
Top	E	C	?	E	?	E	B
Power	L	M	H	H	H	M	M
Data	St	NS	St	no	NS	NS	NS
Dyna	C	CT	CT	T	T	T	T
Space	H	M	H	S	?	S	H

Figure 2.17. A comparison of some interesting dynamic machines.

The features are as follows:

- MS Machine structure is one level (1L), two level (2L), two stage (2S), or other (O).
- Top Topology of communication unit is equidistant (E), bus (B), or cube (C).
- Power Computational power per processing element is high (H), medium (M), or low (L).
- Data Hardware data structure support for streams (St) or general non-strict data structures (NS).
- Dyna Dynamic mechanism uses code copying (C) and/or tags (T).
- Space Space management is static (S), hardware supported (H), or by means of a software manager (M).

2.5. The Manchester Data Flow Machine

Around 1976 John Gurd and Ian Watson started a research project on data flow computing at the University of Manchester. They conceived a two level machine as shown in figure 2.14(b). Since they believe that the construction of an asynchronously operating packet communication network serving more than a few dozen processing elements is not realistic at present, the emphasis of their work has been on constructing a powerful processing element. This machine is described in detail, since it is the target machine for the compiler to be described in chapter 8. The description in this section is based on [Gurd80, Kirk81, Wats82, Silv83] and on personal communication.

2.5.1. OVERVIEW

The group developed the tag concept to increase parallelism for reentrant graphs, independently from similar work elsewhere. The structure of their processing element (figure 2.18) is similar to that shown in figure 2.13. It is a pipeline of four units: token queue, matching unit, fetching unit, and functional unit. Each unit works internally synchronous, but they communicate via asynchronous protocols. More than thirty packets can be processed simultaneously in the various stages of the pipeline. To maximize communication speed the data paths are all parallel (up to 166 bits wide) transmitting a complete packet at a time. Consequently the sizes of packets, and thus of tokens, are fixed.

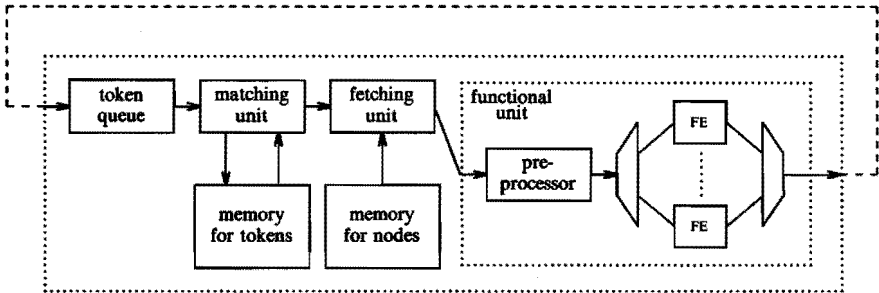


Figure 2.18. Functional diagram of a processing element in the Manchester Dataflow Machine.

The *token queue* is a simple FIFO buffer currently accommodating 32 K tokens. It serves to smooth the irregular output rates of two other units in the pipeline: the matching unit and the functional unit.

The *matching unit* accepts tokens from the token queue and sends complete sets of input tokens to the fetching unit. Currently it can store 1 M tokens. Since in this machine the number of input arcs of a node is limited to two, the destination node is either a *single-input* or a *dual-input* node. Each token carries information to distinguish the two cases. In the former case the token is simply passed on to the fetching unit (a *bypass*.) In the latter case a *match operation* is performed, as described below. A match operation may or may not result in the production of an output packet and this accounts for the variable rate of this unit.

The *fetching unit* combines the set of input tokens with the description of the destination node into an executable packet. The node space is divided into segments to provide rudimentary protection in case of multi-programming. The prototype currently accommodates 64 K nodes. Each node may contain up to two destination descriptions, each consisting of an address and an indication whether the destination node is single-

or dual-input. One of the descriptions may be replaced by a *literal*, a constant input token for one of the two input arcs. Such a carried node is then single-input.

The *functional unit* consists of a preprocessor and a set of functional elements connected via a distributor and an arbiter. The preprocessor executes instructions that require access to a counter memory. Most counters are used to monitor performance. One counter, called the *activation name* counter, is used for the generation of unique tag areas and can be manipulated by the program proper. Although this is not a functional operation, the instruction set is such that this in itself cannot lead to non-functional programs. The functional elements are micro-programmed bit-slice processors. The processing time per instruction varies from 3 to 30 micro-seconds, with an average of 6 micro-seconds. This variation, combined with the fact that an instruction may produce 0, 1, or 2 tokens, accounts for the irregular rate of the functional unit.

The prototype is connected via a rudimentary communication network to a VAX 11/780, which serves as host computer. Since the loading of the node memory and the micro-programs for the functional units and several other control functions are all accomplished through the use of special packets, no other communication paths than the ones shown in figure 2.18 are needed.

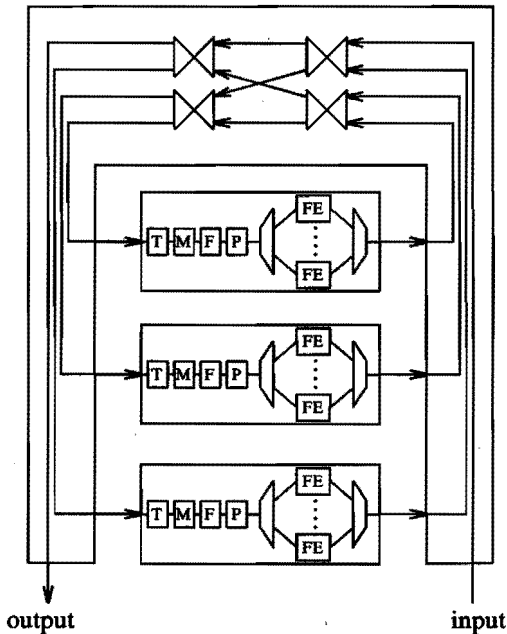


Figure 2.19. The Manchester Dataflow Machine with three processing elements.

Figure 2.19 illustrates the structure of a multiprocessor with 3 processing elements. The communication unit consists of 2×2 routing elements, each of which may accept a token from one of its input lines and send it to one of its output lines. For $n - 1$ processing elements $\log_2 n$ layers of $\frac{1}{2} n$ routing elements each pair are needed. The communication unit is equidistant: the distance between any pair of processing elements is the same as that between the output and input of one processing element. The routing of tokens is determined by the destination address and/or the tag, depending on which allocation strategy is chosen. Since the communication unit has no locality properties that the allocation policy could take advantage of, only an even load

distribution over the processing elements has to be ensured. At present a pseudo random distribution is envisioned, implemented by hashing on both address and tag.

2.5.2. THE MATCH OPERATION

When a token destined for a dual-input node arrives at the matching unit, it performs a match operation, i.e. searches its memory for a token with the same destination and tag. In a dataflow machine matching implements the synchronization of and the communication between concurrent threads of execution. An efficient implementation is crucial for the performance of the whole machine. The introduction of matching functions, described below, requires the matching unit to support the storage of data structures. These two factors make this unit the most interesting part of the machine. The unit can be considered to implement a sparsely occupied virtual memory with the pair <destination,tag> as memory address. The search consists of retrieving the addressed memory cell. If it is empty the match *fails*. If the cell contains a token destined for the same input port, a fatal condition, known as *token clash*, has arisen due to unsafety of the graph and the execution is aborted. If the cell contains a token destined for the other port, the two tokens are partners and the match *succeeds*. They are combined into a packet and sent to the fetching unit. What extra action is to be taken in case of failure or success is determined by a field in the incoming token, known as the *matching function*.

Matching Functions.

There are four success actions:

- | | |
|-----------|--|
| Extract | The token is removed from memory. |
| Preserve | The token is left in memory. |
| Increment | The value of the token in memory is incremented. |
| Decrement | The value of the token in memory is decremented. |

The four fail actions are as follows:

- | | |
|----------|--|
| Wait | The incoming token is stored at the memory location. |
| Defer | The incoming token enters a "busy-wait" cycle: it is passed as a special packet through the rest of the processing element and the communication unit, until it reaches the matching unit again, where the match operation will be repeated. |
| Abort | The incoming token is combined with a special "empty" token into a packet and sent to the fetching unit. |
| Generate | As Abort but a copy of the incoming token is placed in memory as if it were a preserved partner. |

Not all combinations are allowed. Normally only the combination Extract-Wait is used, i.e. the partner is removed if present, otherwise the incoming token is stored in the matching store. The matching function Preserve-Defer may be used to implement a memory function (see figure 2.20). Data structures can thus be stored without a separate structure memory. Storing large data structures, however, may burden the matching unit considerably. The storage facility is quite primitive: reference counts and garbage collection have to be implemented in software. A separate structure store that provides these facilities directly is being implemented. The other combinations are not discussed in detail. Suffice it to say that Increment-Defer and Decrement-Defer are useful when an indivisible semaphore action is needed as in e.g. resource management. Extract-Abort allows testing of the state of the matching unit, whereas Preserve-Generate is useful in case special action needs to be taken the first time an arc is

traversed. The indication that a token should bypass the matching unit because the destination is a single input node is sometimes also referred to as the matching function *Bypass*.

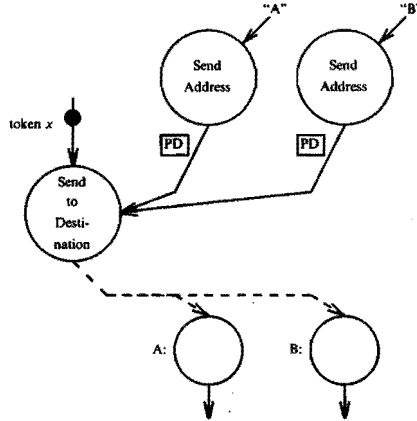


Figure 2.20. The storage of a token.

Token x is "stored" by sending it to the first input port of a dynamic node (see also figure 2.20). The address tokens entering this node at the other port carry a Preserve-Defer matching function (PD). The Preserve action makes them into requests to send a copy of the stored token to the designated node. The Defer action is needed since several requests may arrive before token x is stored. If the stored token becomes garbage it has to be collected by means of a request to send the token to a sink. This request should carry a normal Extract-Wait matching function.

The extra success actions are optimizations (see figure 2.21), and do not add to the power of the machine. The extra fail actions, however, introduce the possibility of non-deterministic graphs, where the output is dependent on the relative timing of node firings. It also allows the construction of safe but non-functional (i.e. history sensitive) graphs. The Defer action in fact changes the concept of safety: more than one token with the same tag are allowed on a port as long as they have Defer matching functions. Deferment is essential for the efficient implementation of data structures, but, if busy-waiting occurs, taxes the resources of the machine. Böhm [Bohm84] has shown that all special matching functions can be simulated with a so called "there box", which is equivalent to an Extract-Abort matching function.

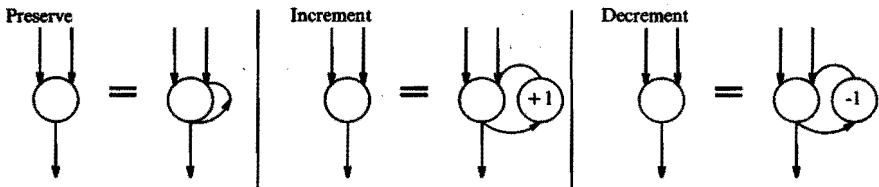


Figure 2.21. Equivalence of matching functions and cycles.

The success actions of the special matching functions can all be implemented by cyclical graphs.

Realizing the Virtual Matching Memory.

Since the virtual matching memory is occupied so sparsely it cannot be implemented directly, but has to be mapped onto a physical memory of realistic size. An associative memory could be used but it was determined that simulating this by means of a hardware hashing mechanism is more cost-effective. The 54 bit matching key (18 bits for the destination and 36 bits for the tag) is hashed to a 16 bit address to access a 64K memory. Each cell has room for one token including destination address, tag, and an extra bit to indicate an empty cell. If the accessed cell is empty the match fails. If it contains a token, its address, tag, and port are compared with those of the incoming token leading to either success, failure, or token clash. When the incoming token has to wait upon failure it has to be stored at the same address. At present 20 of these memory banks work in parallel, so 20 tokens that hash to the same address can be accommodated simultaneously. When a token needs to be stored for which all 20 slots are occupied, it is diverted to the *overflow unit* (presently simulated by the VAX 11/780), which handles the tokens in a conventional manner. The matching unit uses an extra 64K bit memory to indicate which hash keys have overflowed and routes each failing token hashed to an overflowed address to the overflow unit to continue its search for a partner. Other tokens can be processed concurrently, since the order in which matching occurs does not affect the computation.

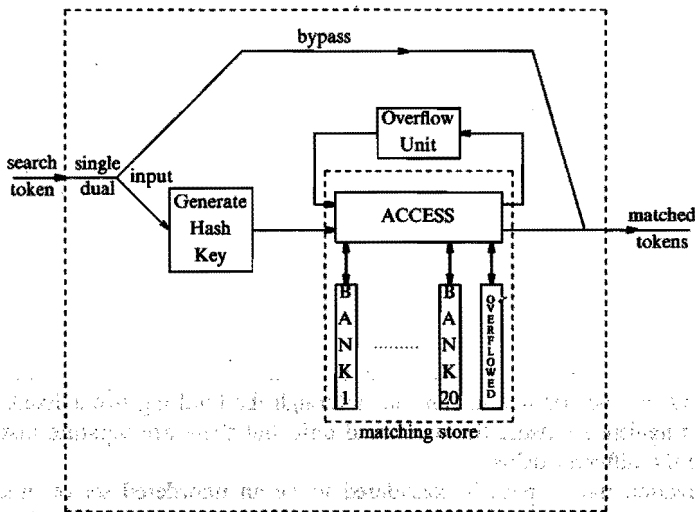


Figure 2.22. Matching Unit.

For each token destined for a dual input node a hash key is generated based on destination address and tag. In the second stage of the pipeline the 20 slots of the memory banks are accessed in parallel. If the partner is present the match succeeds. If the match fails the token is stored unless all slots are already occupied. In that case the token is diverted to the overflow unit and an overflow bit is set. A token for which the match fails and the corresponding overflow bit is on is always sent to the overflow unit.

Since the processing of overflowed tokens is slower than normal matching, a small fraction of overflowed addresses (less than 1%) may have a considerable effect on the overall performance. When such level of overflow is reached, 60% of the memory is occupied on average.

2.5.3. INSTRUCTION SET

To give an impression of the architecture of a typical dataflow machine as perceived by the machine language programmer the instruction set is presented in some detail. More information on the instruction set can be found in chapter 8.

Operators.

Operators are instructions that implement arithmetical, logical, and relational operations that could also be found in a conventional architecture. Since each token carries a type indication, polymorphic operators (such as an addition operation that can handle reals and integers) could have been included. Instead the architects decided to implement strong typing, where operators check whether the input tokens conform to rather strict type restrictions. They hoped that its inherent redundancy would lead to a more robust system. As a consequence the machine has a large set of standard types and a great number of operators (two thirds of the 77 instructions listed in [Kirk81]). In a parallel computer it is not so simple to abort a computation when an error condition is detected. Instead a standard type Error is included, which is transmitted by each operator throughout the graph and will eventually appear in the output.

Flow Control.

Characteristic for a dataflow machine are the flow control instructions. The `DUPLICATE` instruction, which simply copies its input to two outputs, is essential since the number of output tokens per node is limited to two. The `KILL` instruction acts as a sink. Conditional flow is directed by `BRANCH` instructions, which send their first input token to one of the output arcs depending on the value of the second input token. The various branch instructions direct tokens of type Error to a fixed output port, making it possible to write programs that terminate rapidly whenever an error is detected. The implementation of procedure returns requires *dynamic* nodes with an output arc that is not fixed but determined at run time (see figure 2.11).

Tag Manipulation.

The tag is divided into three fields. The *iteration level* is used to separate subsequent activations of a loop body, the *index* is used to separate elements of a data structure, and the *activation name* is used to separate tag areas. Through clever encoding the sizes of these fields are determined at run time although the total tag size is fixed. The fields are not distinguished outside the functional unit, but there are separate instructions to manipulate the different fields.

The activation name space is considered to be an unordered set of unique names. The `GENERATE-ACTIVATION-NAME` instruction generates a new activation name by causing the preprocessor to increment its activation name counter and prefixing its value by the processing element identifier. Consequently the activation names are unique, but their supply is rather limited (programs with too many procedure calls cannot be supported). Other schemes have been proposed that do not make use of the central counters, but generate and recycle activation names locally in the graph [Catt81].

Since tokens of this type may not be converted to any other type, the non-functionality of this instruction is harmless. The iteration level and index may be set to an integer and they may be subject to arithmetic. The special operations on iteration level can be seen as optimizations of the more general activation name operations, taking advantage of the restrictions that make iteration equivalent to tail recursion.

Data Structures.

A data structure can be sent over an arc with each element as a separate token distinguished by the index field of the tag. The elements can be produced and accessed in any order and concurrently. Retrieving a single element of a data structure in copying mode (acceptable for small structures) is accomplished by sending all tokens of a structure to a node that transmits the token with the proper index field and discards all other tokens. The storage of data structures is accomplished by matching functions as shown in figure 2.20. Many algorithms exhibit a pipeline type of parallelism, calling for implementation with streams, which are produced and consumed in order. With the aid of special instructions (see [Bowe81]) this type of processing can be made quite efficient. Other instructions facilitate the interaction between subsequent iterations and subsequent data structure elements.

2.5.4. STATE OF THE PROJECT

The first prototype processing element, which became operational in the fall of 1981, has been subjected to numerous performance studies, and unsatisfactory parts have been improved. For a set of benchmark programs a performance of one to two million instructions per second has been reached [Gurd85]. Since the prototype is implemented in medium performance technology, an upgrading to around ten million instructions per second for one processing element seems feasible. Before an expansion to a four processing elements machine is attempted, an emulator is being constructed to study the behavior of the communication unit. The emulator consists of 16 pairs of micro-processors, each pair emulating one processing element, connected via a synchronously operating packet switching network. A separate structure store has been constructed and is now being installed. In the following section some conclusions are drawn based on the experiences gained so far.

2.6. Feasibility of Dataflow Machines

We saw in the previous section that a processing element for a dataflow machine can be constructed with a speed of close to ten million instructions per second. Since dataflow machines are in principle extensible, a machine consisting of more than a hundred processing elements could conceivably reach a speed in the range of a billion instructions per second. It is too early to tell whether this potential can indeed be realized; much work needs to be done on allocation schemes and experience needs to be gained with data structure support and networks that connect many processing elements. But even if a machine with such a performance could be constructed, the question remains whether the amount of hardware needed for such a machine would not be better used by an alternative architecture. In fact most of the objections raised against the dataflow approach are concerned with factors that are believed to reduce the effective utilization of a dataflow machine to an unacceptably low level. A well argued case is made by Gajski et al. [Gajs82]. They claim that most programs do not contain enough parallelism to utilize a realistic dataflow machine except when large arrays are processed in parallel. They also claim that the handling of large data structures involves considerable overhead in the form of either excessive storage or excessive processing requirements. With several prototypes operational the validity of such objections can now be judged on the basis of actual experience. In this section this question is addressed with respect to the Manchester Dataflow Machine, concentrating on underutilization and overhead. To facilitate the description, these will be treated together as *resource waste*. Roughly speaking wasted resources are considered to be those that are needed beyond those in a reasonably high performance sequential

computer.

Most of the hardware of the Manchester Dataflow Machine can be classified as being used either for *processing* or for *storage*. All functional elements together form the processing hardware. Storage consists of data and instruction memories in the token queue, the matching unit, and the fetching unit. The rest of the hardware we classify as being used for *communication*. The total resource waste in this machine can be estimated if we know the relative sizes of the three categories and the level of waste within each category. As a rough measure of the amount of hardware we use the number of printed circuit boards ignoring differences in board and chip density.

A multiprocessor consists of a number of processing elements connected with a communication switch. The amount of hardware in the switch per processing element grows logarithmically with the size of the machine. A machine containing a few dozen processing elements would require about 2 printed circuit boards per processing element for the switch alone. One processing element is currently implemented with about 15 printed circuit boards for processing, 22 for storage, and 9 for internal communication. We can say that about 50 % of the hardware is devoted to storage, 30 % to processing, and 20 % to communication.

The amount of communication hardware is relatively small, especially considering that most of it is needed for the asynchronous communication between the units. Since the same architecture could have been implemented synchronously, we concentrate on the other two categories.

2.6.1. PROCESSING

Processing power is wasted either because a functional element is idle or because it is performing overhead computation, i.e. computation that would not be needed in a sequential implementation. We treat these two factors in order.

Underutilization of Functional Elements.

A functional element is idle because of a poor hardware balance, lack of parallelism in the program, or poor distribution over the processing elements. Balancing the hardware amounts to adjusting the number of functional elements to the speed of the matching unit and providing enough buffering to smooth irregularities. This has been done by analysis and by experiment [Gurd83, Gurd85] and it has been concluded that the functional unit should contain between 12 and 20 elements.

In such a configuration there are 30-40 stages in the pipeline that can concurrently be active. The parallelism in a program should thus be at least 30 per processing element to avoid starvation of functional elements and preferably more to accommodate the smoothing buffers. Experiments with simple programs run on one processing element indicate that an average parallelism of 50 is sufficient. A reasonably sized multiprocessor would therefore need programs with an average rate of parallelism close to a thousand. Experience so far suggests that realistic programs can indeed achieve such rates of parallelism. Programs with a regular type of parallelism, for which the average rate of parallelism is close to the maximum rate, do not create problems. Such programs, however, run well or even better on more conventional parallel computers. For programs with irregular parallelism the amount of parallelism is occasionally so high that the resources of the machine gets flooded by intermediate results, i.e. the matching unit overflows. While originally the extraction of parallelism out of programs was an important research topic, currently attention has shifted to the opposite: the search for a *throttle*, a mechanism to dynamically limit parallelism if resources tend to get overloaded. In the Manchester processing element this could be

implemented by replacing the FIFO token queue by a more sophisticated mechanism that would classify tokens in different categories and favor a particular category depending on machine load. A suggestion for this also appears in [Veen80]. An effective classification would need assistance from the compiler.

Distributing the work load over the processing elements is in general a complicated allocation problem that needs to take the locality of instruction and data access into account. In the Manchester machine the problem is simplified, since all communication paths are of equal length so there is no physical locality that the allocator needs to exploit. The architects expect that a pseudo random distribution based on a similar hashing technique as used in the matching store will provide an even distribution.

Overhead Computation.

Even if the functional elements are sufficiently utilized, processing power can still be wasted if many instructions are in fact overhead. One source of this type of overhead mentioned in [Gajs82] is the distributed nature of flow control. A manifestation of this problem is the separate branch instructions that need to be executed for each data item that enters a conditional expression compared to the single jump instruction in a control flow computer. Nested conditionals aggravate the problem considerably. Another manifestation in a tagged architecture is the tag manipulation instruction that is needed for each data item entering a reentrant subgraph. Possibly the largest source of overhead computation is in the handling of large data structures. Whenever a complete data structure is transmitted where a pointer to a stored copy could have been used, as many overhead instructions are executed as there are elements in the structure.

For certain numerical programs an indication of the amount of overhead computation is provided by the *floating point fraction*, i.e. the fraction of executed instructions that perform floating point operations.¹ Studies of benchmark programs run on conventional super computers at Lawrence Livermore National Laboratory showed that assembly language programmers achieve a floating point fraction of 30 %, whereas FORTRAN compilers reach 15-20 % [Gurd85]. Straightforward compilers for the Manchester Dataflow Machine achieve a floating point fraction of 3 % for large programs. There is, however, much room for optimization and a good compiler can reduce this overhead considerably. Recent work on optimization in Manchester has achieved floating point fractions of 15 % for realistic programs [Bohm85]. We will return to this issue in chapter 9.

2.6.2. STORAGE

An even distribution of the work over a multi-processor is greatly simplified if each instruction is available on each processing element. Because of all the copies of the program, most instruction storage would be wasted. This waste is, however, insignificant compared to the waste in data storage.

The processing element that is currently operating contains an enormous amount of memory, practically all of it situated in the matching unit. The total hardware cost of the machine is dominated by the cost of this 15 M byte high speed data memory. This memory is so large because large data structures (and sometimes several copies) need to be accommodated and because its effective utilization is less than 20 percent.

1. The Manchester Dataflow group calls the inverse of this figure the MIPS/MFLOPS ratio.

The latter is due to a combination of two factors:

- Each token carries a destination and a tag in addition to its data. Two thirds of each cell is thus dedicated to overhead.
- The occupancy needs to be limited to less than 60 percent to avoid serious performance degradation due to overflow.

Many of the large data structures that have to be accommodated have a long life time. It would be much more efficient to store all elements of such a structure consecutively without tags and destination. An access to an element would then require a pseudo-associative access to the structure and within that area a conventional access to the element. This would practically eliminate the first overhead. A separate structure store based on this principle is now being installed [Sarg85]. It will appear in the multiprocessor as an extra processing element specialized in structure operations. This structure store needs to allocate memory only during structure creation, a relatively infrequent operation. This allows for efficient memory management which will significantly reduce the second source of storage waste.

2.6.3. CONCLUSIONS

The major resource waste occurs in the data memory due to the per token mapping of virtual to physical matching memory. The pseudo-associative memory that is needed for this, with its relatively slow overflow mechanism, necessitates a far too low utilization. A secondary problem is the amount of control information accompanying each data item. The structure store will probably alleviate both of these problems. If this undertaking is successful, the token memory could be greatly reduced in size. It is interesting to speculate on the effects. If the amount of data storage could be reduced to a quarter of what is currently needed, the situation would change considerably: half of the hardware of the machine would then be devoted to processing with the rest evenly divided between storage and communication. The 25 % overhead in communication seems acceptable as long as the functional elements are utilized efficiently. Three factors are most important for this: sufficient parallelism, efficient code (i.e. few overhead instructions), and an even distribution over the processing elements. The first two issues depend greatly on the compiler. Even distribution needs extensive research in allocation schemes. Static allocation (i.e. allocation that does not take the current load distribution into account) probably requires a good compiler that provides locality information. Some experience with static allocation is reported in [Kish83]. It seems probable that in a large general purpose machine a dynamic allocator will be needed. Elaborate allocation schemes that exploit locality have been proposed by Arvind [Arvi83]. A mechanism to dynamically adjust the activity in a processing element (a throttle) seems essential. Such a mechanism could also benefit greatly from information provided by the compiler about the structure of the program.

In summary, allocation and distribution schemes should be a focal point of further research. The quality of compilers could also greatly effect the performance. We come back to this in the next chapter on programming.

References

- Amam82. AMAMIYA, M., R. HASEGAWA, O. NAKAMURA, AND H. MIKAMI (Jun 1982). A List-Processing-Oriented Data Flow Machine Architecture, *AFIPS National Computer Conference 82*, 143-151.
- Arvi77. ARVIND AND K.P. GOSTELOW (1977). A Computer Capable of Exchanging Processors for Time, *Information Processing 77*, 849-853, North Holland.
- Arvi83. ARVIND, ET.AL. (1983). *The Tagged Token Dataflow Architecture*, Technical Report, MIT - Laboratory for Computer Science.
- Bohm84. BÖHM, A.P.W. (Feb 1984). *Dataflow Computation*, dissertation, Mathematical Centre CWI Tract 6, Amsterdam.
- Bohm85. BÖHM, A.P.W. AND J. SARGEANT (Sep 1985). Efficient Dataflow Code Generation for SISAL, *Parallel Computing 85*.
- Bowe81. BOWEN, D.L. (Apr 1981). *Implementation of Data Structures on a Data Flow Computer*, Ph.D. Thesis, Dept. of Computer Science - Victoria University of Manchester.
- Broc79. BROCK, J.D. AND L.B. MONTZ (Jul 1979). *Translation and Optimization of Data Flow Programs*, CSG Memo 181, MIT - Laboratory for Computer Science.
- Burk81. BURKOWSKI, F.J. (May 1981). A Multi-User Data Flow Architecture, *Eighth International Symposium on Computer Architecture*.
- Calu82. CALUWAERTS, L.J., J. DEBACKER, AND J.A. PEPPERSTRAETE (Dec 1982). Implementing Code Reentrancy in Functional Programs on a Dataflow Computer System with a Paged Memory, *International Workshop on High-Level Language Computer Architecture*.
- Catt81. CATTO, A.J. (Jun 1981). *Nondeterministic Programming in a Dataflow Environment*, Dissertation, Dept. of Computer Science - Victoria University of Manchester.
- Comt80. COMTE, D., N. HIFDI, AND J.C. SYRE (Oct 1980). The Data Driven LAU Multiprocessor System: Results and Perspectives, *IFIP80*, 175-180.
- Corn79. CORNISH, M. ET.AL. (Nov 1979). The TI Data Flow Architectures: The Power of Concurrency for Avionics, *Third Conference on Digital Avionics Systems*, 19-25.
- Davi77. DAVIS, A.L. (1977). *Architecture of DDMI: A Recursively Structured Data Driven Machine*, Technical Report, University of Utah, Salt Lake City, Utah.
- Davi79. DAVIS, A.L. (Jun 1979). A Data Flow Evaluation System Based on the Concept of Recursive Locality, *Proceedings National Computing Conference*, 1079-1086, AFIP.
- Denn69. DENNIS, J.B. (1969). Programming Generality, Parallelism and Computer Architecture, *Information Processing 68*, 484-492.
- Denn74. DENNIS, J.B. AND R.P. MISUNAS (Dec 1974). A Preliminary Architecture for a Basic Data Flow Processor, *Second International Symposium on Computer Architecture, Computer Architecture News*, 3.4, 126-132.
- Denn80a. DENNIS, J.B. (Nov 1980). Data Flow Supercomputers, *Computer*, 13.4, 48-56.
- Denn80b. DENNIS, J.B., G.A. BOUGHTON, AND C.K.C. LEUNG (May 1980). Building Blocks for Data Flow Prototypes, *Seventh International Symposium on Computer Architecture*, 1-8.

- Gajs82. GAJSKI, D.D., D.A. PADUA, D.J. KUCK, AND R.H. KUHN (Feb 1982). A Second Opinion on Data Flow Machines and Languages, *Computer*, 15.2, 58-69.
- Gold72. GOLDSTINE, H.H. (1972). *The Computer from Pascal to von Neumann*, Princeton University Press.
- Gost80. GOSTELOW, K.P. AND R.E. THOMAS (Oct 1980). Performance of a Simulated Dataflow Computer, *IEEE Transactions on Computers*, C-29.10, 905-919.
- Gurd80. GURD, J. AND I. WATSON (Jun & Jul 1980). A Data Driven System for High Speed Parallel Computing, *Computer Design*, 9.6&7, 91-100 & 97-106.
- Gurd83. GURD, J. AND I. WATSON (1983). Preliminary Evaluation of a Prototype Dataflow Computer, *Ninth IFIP World Computer Congress*, 545-551.
- Gurd85. GURD, J.R., C.C. KIRKHAM, AND I. WATSON (Jan 1985). The Manchester Prototype Dataflow Computer, *Communications of the ACM*, 28.1, 34-52.
- Hazr82. HAZRA, A. (Oct 1982). A Description Method and a Classification Scheme for Data Flow Architectures, *Third International Conference on Distributed Computing Systems*, 645-651.
- Hock81. HOCKNEY, R.W. AND C.R. JESSHOPE (1981). *Parallel Computers: Architecture, Programming and Algorithms*, Adam Hilger, Bristol.
- Hoge82. HOGENAUER, E.B., R.F. NEWBOLD, AND Y.J. INN (Aug 1982). DDSP - A Data Flow Computer for Signal Processing, *International Conference on Parallel Processing*, 126-133.
- John80. JOHNSON, D. ET.AL. (1980). Automatic Partitioning of Programs in Multiprocessor Systems, *Spring COMPCON 80*, IEEE.
- Karp66. KARP, R.M. AND R.E. MILLER (Nov 1966). Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing, *SIAM Journal of Applied Mathematics*, 14.
- Kirk81. KIRKHAM, C.C. (May 1981). *Basic Programming Manual of the Manchester Prototype Dataflow System*, 2nd Edition, Dataflow Research Group - Manchester University.
- Kish83. KISHI, M., H. YASUHARA, AND Y. KAWAMURA (Jun 1983). DDDP: A Distributed Data Driven Processor, *Tenth international Symposium on Computer Architecture*, 236-242.
- Leco79. LECOUFFE, M.P. (Apr 1979). MAUD: A Dynamic Single-Assignment System, *Computers and Digital Techniques*, 2.2, 75-79.
- Marc83. MARCZYŃSKI, R.W. AND J. MILEWSKI (Jun 1983). A Data Driven System Based on a Microprogrammed Processor Module, *Tenth International Symposium on Computer Architecture*, 98-106.
- Mira77. MIRANKER, G.S. (1977). Implementation of Procedures on a Class of Data Flow Processors, *International Conference on Parallel Processing*, 77-86.
- Misu78. MISUNAS, D.P. (1978). *A Computer Architecture for Data Flow Computation*, Technical Memorandum 100, MIT - Laboratory for Computer Science.
- Mont80. MONTZ, L.B. (Jan 1980). *Safety and Optimization Transformations for Data Flow Programs*, Technical Report 240, MIT - Laboratory for Computer Science.
- Rodr69. RODRIGUEZ, J.E. (Sep 1969). *A Graph Model for Parallel Computation*, Technical Report 64, MIT - Project MAC.
- Rumb75. RUMBAUGH, J. (1975). A Data Flow Multiprocessor, *Sagamore Computer Conference on Parallel Processing*, 220-223.

- Sarg85. SARGEANT, J. (Apr 1985). *Efficient Stored Data Structures for Dataflow Computing*, Ph.D. Thesis, Dept. of Computer Science - Victoria University of Manchester.
- Silv83. SILVA, J.G.D. DA AND I. WATSON (Jan 1983). Pseudo-Associative Store with Hardware Hashing, *IEEE Proceedings Pt. E*, 130.1, 19-24.
- Smit78. SMITH, B.J. (1978). A Pipelined Shared Resource MIMD Computer, *International Conference on Parallel Processing*.
- Swan77. SWAN, R.J., S.H. FULLER, AND D.P. SIEWIOREK (1977). Cm* - A Modular, Multi-Microprocessor, *National Computer Conference*, 637-644.
- Syre77. SYRE, J.C., D. COMTE, AND N. NIFDI (Aug 1977). Pipelining, Parallelism and Asynchronism in the LAU System, *International Conference. on Parallel Processing*, 87-92, IEEE.
- Syre80. SYRE, J.C. (1980). *Etude et Realisation d'un Systeme Multiprocesseur MIMD en Assignment Unique*, Thèse, Université Paul Sabartier de Toulouse.
- Taka83. TAKAHASHI, N. AND M. AMAMIYA (Jun 1983). A Data Flow Processor Array System: Design and Analysis, *Tenth International Symposium on Computer Architecture*, 243-250.
- Trel82a. TRELEAVEN, P.C., R.P. HOPKINS, AND P.W. RAUTENBACH (Feb 1982). Combining Data Flow and Control Flow Computing, *Computer Journal*, 25.1.
- Trel82b. TRELEAVEN, P.C., D.R. BROWNRIDGE, AND R.P. HOPKINS (Mar 1982). Data-Driven and Demand-Driven Computer Architecture, *Computing Surveys*, 14.1, 93-143.
- Veen80. VEEN, A. (1980). Data Flow Computers, in *Colloquium Hogere Programmeertalen en Computerarchitectuur - Syllabus 45*, 99-132, ed. P. Klint, Mathematical Centre, (in dutch).
- Veen81. VEEN, A. (Oct 1981). *A Formal Model for Data Flow Programs with Token Coloring*, IW 179, Mathematical Centre.
- Vegd84. VEGDAHL, S.R. (Dec 1984). A Survey of Proposed Architectures for the Execution of Functional Languages, *IEEE Transactions on Computers*, C-33.12, 1050-1071.
- Wats82. WATSON, I. AND J. GURD (Feb 1982). A Practical Data Flow Computer, *Computer*, 15.2, 51-57.
- Weng79. WENG, K.S. (May 1979). *An Abstract Implementation for a Generalized Data Flow Language*, Technical Report 228, MIT - Laboratory for Computer Science.
- Yama83. YAMAGUCHI, Y., K. TODA, AND T. YUBA (Jun 1983). A Performance Evaluation of A LISP-Based Data-Driven Machine (EM-3), *Tenth International Symposium on Computer Architecture*.

Chapter 3

Dataflow Programming

Programming a parallel computer efficiently requires a subtle skill: a small, semantically inconsequential modification can make a program run many times faster. This is unfortunate since it makes efficiency considerations overly important, which is not conducive to a clear programming style. Moreover a considerable effort is often required to bring a parallel computer to an acceptable level of performance. There are two strategies to facilitate the construction of efficient programs.

- Enrich the *programming language* with constructs for which particularly efficient translations are available and remove constructs that tend to degrade performance. A frequently chosen form of language improvement is the creation of a library of standard functions that have been coded efficiently by other means. The most radical approach is to design a completely new language specifically tailored towards the particular machine.
- Construct a *compiler* that performs an analysis sufficiently sophisticated to generate efficient code. This approach is the popular one for commercially available machines. All pipelined vector computers, for instance, have FORTRAN compilers that *vectorize*, i.e. recognize certain array operations within loops that can be executed efficiently by vector instructions. The patterns that such a compiler recognizes should cover broad categories, because if they are restricted to special cases, programming may become even more complicated. Programs then need to be in a form that will trigger the optimizations and the idiosyncracies of the compiler need to be mastered in addition to those of the machine.

The next two sections give examples of both approaches in the context of dataflow machines. Since dataflow machines have been designed specifically for the efficient support of a new way of programming, the emphasis has been on the development of special languages. These are treated in the first section. The next section considers the merits of using a sophisticated compiler to translate imperative languages. The concluding section compares the two approaches.

3.1. Declarative Languages

The original impetus for the development of dataflow machines came from concern about the inadequacies of existing languages to deal with concurrency. Consequently, the architecture of dataflow machines is to a large extent language based. However, dataflow graphs, the languages they were originally based on, are too low a level for practical programming and higher level equivalents, called *dataflow languages*, were developed. The following restrictions make it easy to translate these languages into dataflow graphs.

- They are all *single assignment languages*: an identifier appears only once as the target of an assignment. In most dataflow languages the single assignment rule is a *static* restriction: an assignment within a loop or a recursive procedure that gets executed repeatedly, is acceptable. An exception to the rule is sometimes made for initializing loop variables. Conditional assignment such as in “if *test* then $x := 7$ fi” is usually not allowed, since x would not be defined if *test* fails. Conditionals only appear as part of the expression on the right-hand side of a definition. A consequence of the single assignment rule is that a data structure has to be created in a single expression. It cannot be modified, although parts of it can be retrieved and used to create new data structures.

An identifier is thus a short-hand for the value computed by the expression on the right-hand side of the assignment. In fact variable and assignment are misnomers and one rather speaks of *value name* and *definition*. An advantage of the single assignment rule is that a value name can be uniquely associated with an output port of a node in the dataflow graph.

- Functions are strictly functional, i.e. two applications of a function with the same arguments deliver the same value. There are no hidden communication channels between function applications. Constructs that can maintain a global state, such as own or global variables, are therefore not allowed.

An important consequence of these restrictions is that the evaluation of an expression is *free of side-effects*, i.e. each result of an expression has to be explicitly indicated by the programmer. Dataflow languages belong to the family of *declarative languages*.¹ Declarative languages are not very strict about the order of definitions: reordering definitions in a syntactically correct program may sometimes transform it into an incorrect one, but never into a syntactically correct but semantically different one.

An example of a dataflow language is VAL, developed at MIT [Acke79]. It introduces a powerful iterative construct containing *reduction operators*, which allow concise expression of certain operations that occur frequently in numerical applications. In VAL the difficult topic of error handling has been thoroughly worked out. It lacks, however, important features like recursion and I/O primitives.

Handling I/O, and especially interactive I/O, is problematic in a declarative language, since it involves communication with a non-functional environment in which order of actions is important. The designers of the dataflow language ID solved this problem elegantly by means of *streams* and *resource managers* [Arvi78]. The data structure *stream* is similar to a one-dimensional array except that its elements can only be produced and consumed in consecutive order. A *resource manager* is a non-functional procedure with internal state similar to a SIMULA class object. Interaction with resource managers is by means of messages that can be non-deterministically merged into streams. This makes it possible to write programs for operating systems,

1. In the literature this group of languages is sometimes called *applicative* or *functional*.

data base managers and interactive I/O, which all require non-deterministic primitives.

Other examples of dataflow languages are LAU [Comt80], LAPSE [Glau78], MAD [Bowe81], and VALID [Amam82]. In fact dataflow languages proliferated to the point where almost each machine design was accompanied by a new language. Fortunately, the last couple of years have seen a concentration of this effort in language design culminating in the definition of SISAL. We first treat this language in some detail and continue with a discussion on the implementation on dataflow machines of the closely related group of functional languages.

3.1.1.1. SISAL

SISAL (Streams and Iteration in a Single Assignment Language) is a result of a collaboration between the University of Manchester Dataflow Group, Lawrence Livermore National Laboratory, Digital Equipment Corporation, and Colorado State University. It is meant as a common high level language for numerical programs to be run on a variety of uni- and multi-processors. It is syntactically similar to PASCAL. A compiler for the Manchester Dataflow Machine has been completed and compilers are planned for a CRAY, a HEP, and a VAX. This would allow easy portability between these machines and greatly facilitate comparative performance studies. SISAL is derived from VAL but includes recursion and streams. The description below is taken from [Glau84] with some additional information from [McGr83].

SISAL provides three data structuring facilities. A **record** is like its PASCAL equivalent except that the complete record has to be defined at once, since the selective update of a field would violate the single assignment rule. Instead there is an operation that creates a copy of a record with one field replaced by a new value. There are similar restrictions for an **array**: instead of updating one element, a new array has to be created that is a copy of the old array with one element replaced. A **stream** is an array on which only a restricted set of operations is defined such that it is guaranteed to be created and accessed in order. This has the advantage that in certain implementations an expression that consumes a stream may overlap in execution with the expression producing the stream (pipelined parallelism). Streams in SISAL are always finite.

Each expression delivers a value or a sequence of values. The most simple expression is the **let** expression, consisting of two sections: the *defining* section contains local definitions to be used in the *result* section. For example

```

root1 , root2 :=
  let
    d := sqrt( b * b - 4 * a * c);
    t := 2 * a
  in
    (-b + d) / t , (-b - d) / t
  end let

```

computes two roots of a quadratic equation. The value names used in the result section should be defined in the defining section or previously in the surrounding expression. A value name can be defined only once (single assignment rule) and should follow normal sequencing constraints (no use before a definition) to facilitate the detection of cyclic dependencies.

In conditional expressions all branches have to be present and have to deliver the same number of values:

```
small,big := if a < b then a,b else b,a end if
```

A powerful iterative construct is provided in two varieties. In the most general form all iterations are conceptually performed in sequence. Values computed in the previous iteration are accessible by prefixing the value name by **old**. These so called *loop names* are initialized in a separate **initial** section of the expression. Each loop returns a value, specified in a **returns** section. This value may be a stream or an array, each element of which is to be provided by one iteration. The power of the **returns** section is significantly extended by the provision of *reduction* operators, which may specify that the result is the **sum**, the **product**, the **least**, or the **greatest** of a series of values, each of which is computed in one iteration. Since these reduction operators are based on associative and commutative operations, they can be executed in the order that is most efficient on the target machine. If the operation must be treated as non-associative (because of rounding errors, overflow, or underflow) the order may be specified. The reduction operators make possible concise expression of many numerical algorithms.

A program to compute some Fibonacci numbers can be specified as follows:

```
fibnumbers :=
  for
    initial
      fib1 := 1 ; fib2 := 1
    repeat
      fib1, fib2 := old fib2, old fib1 + old fib2
    while
      fib2 < max
    returns array of fib2
  end for
```

If the number of iterations is determined before the loop starts and iterations are not dependent on each other (i.e. the body does not contain **old**), the more restrictive variety of the iterative construct, which is equivalent to a *forall* expression, can be used. In this variety loop names range over a fixed set such as the elements of an array. The primitives available to specify this range, together with the reduction operators, facilitate the use of the *forall* variety in a wide range of circumstances by removing the need to use the qualifier **old** in the loop body. As an example, the computation of innerproduct and sum of two vectors *A* and *B* may be described as follows:

```
InnerProduct, SumVector :=
  for ElemA in A dot ElemB in B
    prod := ElemA * ElemB;
    sum := ElemA + ElemB
  returns
    value of sum prod,
    array of sum
  end for
```

The first line of the **for** expression specifies the ranges of loopnames; the keyword **dot** specifies that the elements of *A* and *B* should be distributed over *ElemA* and *ElemB* in pairs.

The language is currently under revision. A compiler for the Manchester Dataflow Machine has been written, which provides a fairly complete implementation of the version just described. In this first compiler the code generator is straightforward: for most constructs a simple implementation is chosen and no attention is paid to optimizations. A more efficient implementation is presently being developed.

3.1.2. FUNCTIONAL LANGUAGES

Another group of declarative languages is formed by the functional languages,¹ which enjoy a growing popularity, at least in academic circles. In these languages the evaluation of expressions is also free of side-effects due to the exclusion of global variables and multiple assignments. The main difference, compared with dataflow languages, is that functions play a more central role and appear as objects that can be manipulated. It is usually possible to define a higher order function: i.e. a function that produces a function as a result. Iterative constructs are seldom provided. Non-strict data constructors, i.e. built-in functions that do not require all arguments to be evaluated, make computation on infinite data structures possible. For example, if *append* is a non-strict operator that places an item at the head of a list and we define

$$\text{IntegersFrom}(n) = \text{append}(n, \text{IntegersFrom}(n + 1))$$

then “IntegersFrom(1)” represents the infinite list of integers. However, calculating the sum of the first 10 elements of this list is a finite operation. Non-strict operators require demand driven (or lazy) evaluation in which only those operations that are necessary to produce the required output are performed. The necessary operations are determined by a process known as *demand propagation*.

Since there is considerable interest in the execution of functional languages on parallel machines, we take some time to describe their implementation on dataflow machines. This description is based on [Rich82] and [Ping83]. In such an implementation a second graph is superimposed on the dataflow graph, similar to the first graph but with all its arcs reversed. It is called the *demand graph*, and the tokens flowing through it are known as *demands*. The difference between demands and normal tokens is conceptual; for the machine they are indistinguishable. An initial demand is sent to the root node of the demand graph. Demands then propagate through the demand graph until they reach constants or input expressions where they initiate normal data driven execution. Figure 3.1 shows examples for the demand subgraphs of some strict and non-strict operators. Note the complicated mechanism needed for a shared expression, i.e. an expression whose value is used by more than one expression. If the demands were simply propagated the expression would be evaluated more than once (string reduction). This can be avoided if demands are shared between expressions (graph reduction). Complications arise, however, since it is not clear whether all demands will eventually arrive, due to the conditional propagation of demands by non-strict operators. The mechanism illustrated in figure 3.1(d) is expensive compared to the simple DUPLICATE node in data driven execution and has the additional disadvantage that tokens may be left in the matching store when the program terminates.

1. Functional languages are also known as *applicative languages*.

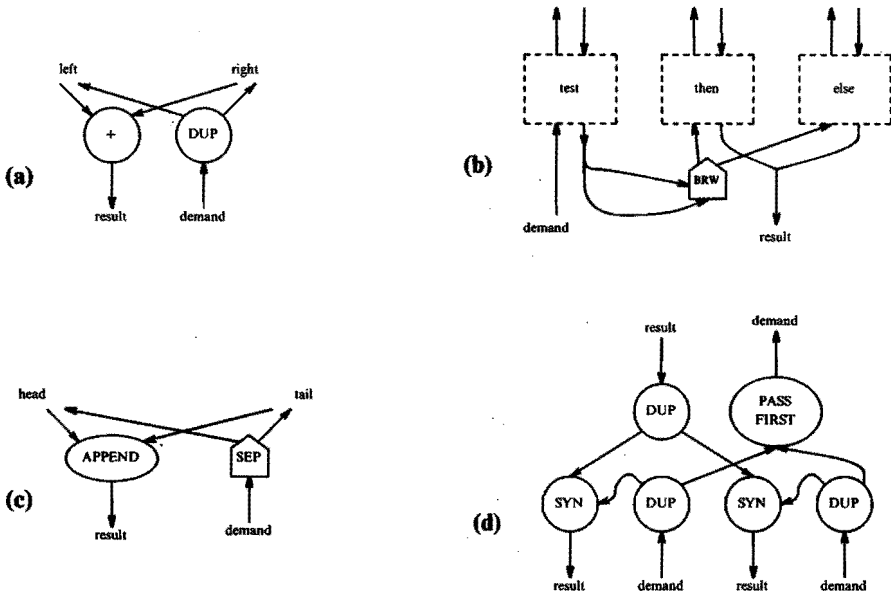


Figure 3.1. Demand subgraphs for some operators.

These are subgraphs as used in the *sasl* implementation on the Manchester Dataflow Machine.

(a) For a strict operator a one-node subgraph is created that simply distributes the demand to its operands.

(b) A demand for a conditional expression is sent to the *test* expression, which will be evaluated. Its result determines to which subexpression a demand will be propagated.

(c) The non-strict *APPEND* operator forms a new list by prefixing an element *head* to an existing list *tail*. Its demand subgraph consists of a *SEPARATE* node. Assuming that demands for a list element are tagged with the sequence number of the required element, the *SEPARATE* node sends a demand for the first element to the left and other demands to the right with their sequence number decremented.

(d) The subgraph for shared values. By means of special matching functions the *PASS-FIRST* macro propagates the first demand to arrive and absorbs the next one. A copy of the computed value waits at the *SYNCHRONIZE* node for the second demand to arrive.

In the implementation described so far, a portion of the demand graph is constructed for each operator in the dataflow graph. Pingali&Arvind [Ping83] have looked at optimizations. If large strict expressions transmit a demand directly to their inputs, the demand graph can be substantially reduced in size. This can be accomplished by *strictness analysis*. The demand graph for most conditional expressions could also be optimized by transferring the *BRANCH* nodes in the demand path to the dataflow path of the input arcs, in the same way as for a normal data driven implementation (see figure 2.5). In fact this normal implementation has a demand driven nature at conditional points: tokens entering one of the branches are stopped until the condition has been evaluated. For other non-strict operators similar optimizations are possible, thereby reducing conditional propagation of demands and the need for an expensive mechanism for shared values. The work so far indicates that an efficient implementation of a functional language on a dataflow machine requires a sophisticated compiler.

It is interesting to note that the situation for reduction machines, especially designed for the execution of functional languages, is not all that different. The architects of the

ALICE machine [Darl81], for instance, expect that a realistic implementation requires a sophisticated compiler that uses data driven execution except where demand driven execution is mandatory due to infinite data structures. Since the efficient implementation of data structures on a fine-grain parallel machine seems to be problematic in general, it is not clear whether infinite data structures complicate the problem fundamentally.

3.2. Imperative Languages

Dataflow graphs originated from dissatisfaction with attempts to incorporate concurrency into existing languages. Most of the work on implementations of high level languages on dataflow machines has focused on languages that were expected to be easy to translate into dataflow graphs. Not much attention has been paid to the question whether any of the languages that were the source of the original dissatisfaction could be implemented efficiently on dataflow machines. Although there is general agreement about the value of such a translation, it is commonly assumed to be too complex to be practical. The source of this complexity is to be found in the *imperative* nature of these languages.

Imperative languages have well developed mechanisms (assignments, pointers, global variables) to facilitate the use of *side-effects*: not all inputs and outputs of a statement need to be explicitly indicated. The evaluation of an expression may, for instance, involve the evaluation of a procedure that changes the value of a global object. Practically all widely used programming languages are imperative: FORTRAN, COBOL, BASIC, PASCAL, ADA, and even the commonly used varieties of LISP. A dataflow machine that does not accept any of these languages can hardly be called a general purpose computer. Since continuity is often as important as efficiency, a machine that would render all existing software useless would not be very attractive.

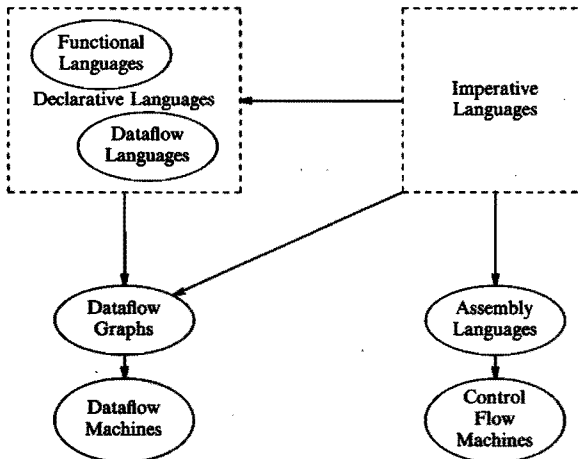


Figure 3.2. Relationship between dataflow machines and high level languages.

Dataflow graphs, developed as a means to express concurrency, lead to the development of dataflow machines for their execution and to dataflow languages for the effective expression of algorithms. A declarative program is much easier to translate into dataflow graphs than an imperative program.

The facilities in an imperative language that make side-effects attractive are, roughly speaking, the same that make the translation of an imperative program into a dataflow graph problematic. The following features are frequently encountered in imperative languages:

Jumps

The regular control flow patterns implied by structured statements as conditional, iteration, and procedure call may be disturbed by escapes or *goto*'s.

Aliasing

One memory location can be addressed and modified through different access paths.

The multiple paths can be created by pointers, call-by-reference parameters, explicit aliasing (e.g. the *EQUIVALENCE* statement in *FORTRAN*) or through array indexing (“*a[i]*” may address the same location as “*a[j]*”).

Multiple Assignment

A variable can appear as the target of several assignments.

Global Objects

Through global objects a nested procedure invocation may exchange information with another one without this being visible at intermediate levels.

Selective Modification of Data Structures

A selective update operation may replace a single element of a large data structure.

The jumps in a program can be removed by transforming them into conditional or iterative constructs, but at the cost of possibly introducing many superfluous dependencies between statements. Multiple assignment and global objects by themselves are not hard to deal with, but the presence of recursion and especially aliasing complicates the problem considerably. The efficient implementation of data structures requires recognition of the access patterns.

Despite the recognized value of a translation of imperative programs into dataflow graphs, the literature on the subject is quite limited.

- Long before dataflow graphs were introduced, Miller&Rutledge described how a program can be transformed into a specification for a hardware device, which is equivalent to a dataflow graph. Their method, which is applicable to assembly languages as well as higher level imperative languages without recursion, breaks the program into basic blocks and constructs a data flow segment for each basic block. The segments are connected with gates inserted at places where conditional control flow occurs. Concurrent execution of loop iterations is prevented by locking. All accesses through “computed addresses” (e.g. arrays) are sequentialized.
- Whitelock [Whit78] constructed a compiler for the Manchester Dataflow Machine that accepts a quite restricted subset of *PASCAL*. The most essential among the restrictions are the exclusion of jumps, aliasing, indirect recursion, pointers, and data structures. Attempts to extend the compiler so it could accept arrays have unfortunately been abandoned.
- Allan et al. defined a new language based on *PASCAL*, excluding all “features incompatible with the notion of functionality” [Olde78, Alla80]. Among the casualties were jumps, pointers, global variables, and data structures that can be modified. They defined and simulated a conceptual model of a dataflow machine, which they used as the target for a compiler. Their compiler cannot handle recursion and is rather conservative in its data flow analysis.
- The work of Ottenstein [Otte81] is not focused on one particular language, but gives a rather comprehensive treatment of the features common in imperative languages. The method he describes is similar to the one to be presented in chapter 5. It transforms a program into a representation in which both control flow and data

dependency is encoded. He suggests the possibility of generating code for dataflow machines and provides a few examples, but no implementation of this suggestion has been reported.

- Kuck and his colleagues [Kuck81] have worked for many years on the analysis of FORTRAN programs and the generation of high quality code for parallel machines. Their analysis could be a good starting point for an implementation on a dataflow machine, but their interest has so far been confined to vector processors.

3.3. Imperative versus Declarative Languages

With the staggering number of programming languages already available, the reasons for introducing yet a new one should be very strong indeed. Declarative languages have been introduced largely because they are supposed to make the construction of correct and clear programs much easier. This section does not discuss this hotly debated issue, but considers the merits of two additional advantages often cited by advocates of the programming of dataflow machines in declarative languages: declarative programs are easier to translate and contain more inherent parallelism than imperative programs.

A declarative program is indeed easier to translate into an efficient dataflow graph than an imperative program. This is due to the difference in their “underlying computational model”, i.e. the way the meaning of a program is most naturally described. A program in an imperative style, i.e. one which relies strongly on side effects, is best understood as a *sequential* composition of statements each of which affects the *computational state*, i.e. the function that maps variables to values. A declarative program is best described by associating a function with each statement, and interpreting the composition of statements into a program as function composition. While the computational state corresponds directly to the memory in a conventional computer, function composition corresponds to the way a dataflow graph is composed out of subgraphs.

An imperative program can be transformed into a declarative one by replacing each jump by an appropriate conditional or loop and adding the computational state to the interface (all inputs and outputs) of each statement. If such a transformed program would be directly translated into a dataflow graph an almost linear graph would result: each statement would be dependent on its predecessor and there would be little parallelism. A type of analysis, called *data flow analysis*, is needed to determine the real interface. Part of the interface of an imperative statement may be hidden due to side-effects. To limit the amount of analysis two statements are sometimes assumed to be dependent, while further analysis could determine that they are not. Such assumptions introduce superfluous data-dependencies, which reduce the parallelism of the translated program. A good analyzer therefore spends a lot of effort to avoid superfluous data-dependencies.

In the first approximation no such problems are encountered when a declarative program is translated: the interface of each statement is explicitly specified. In a way the analysis has already been done by the programmer and no data flow analysis is required to generate a dataflow graph with reasonable parallelism. Since *wide* interfaces (i.e. many input and output variables) are bothersome to deal with, programmers tend to formulate their algorithms in a way that minimizes the width of interfaces. If no data structures are involved, this tendency reduces data dependencies between statements. This is the reason why it is often claimed that the average declarative program is inherently more parallel than its imperative equivalent. Inherent parallelism is, however, not a well-defined concept: a programmer working with vector processors

will often attribute a different level of inherent parallelism to a program than an architect of a tagged dataflow machine. The actual parallelism that a program exhibits during execution depends on the machine and on the compiler. When the inherent parallelism of a program is assessed, a type of machine and a type of compiler is tacitly assumed. Inherent parallelism is in the eye of the beholder, and the beholder often assumes a straightforward compiler. However, already a modest amount of dataflow analysis can often raise the level of parallelism substantially.

When a declarative program that manipulates large data structures is to be translated into an efficient dataflow graph, analysis similar to that needed for imperative programs is required. This is not surprising, since the computational state could be represented by a data structure that can be treated just as a conventional memory, if the right operators are available to structure, copy, and manipulate data structures. A declarative program can therefore have an "imperative nature". A good indicator is the *volume* of the interface of the average statement (i.e. the amount of data flowing through it): in a program with an imperative nature this is high compared to the number of primitive operations that the statement represents. In such a program most input items of each statement are transmitted to its output without modification: the real interface is much smaller than the one explicitly indicated. Data flow analysis is needed to determine the real interface, just as for the translation of imperative programs. The operations on data structures that are available and the way they are implemented are therefore of crucial importance for the behavior of declarative languages on dataflow machines.

In summary, a translator for imperative programs clearly needs to analyze its input program extensively, whereas a translator for declarative programs would also benefit from an analysis of data structure accesses. In the next few chapters a new method for flow analysis and its use for the translation of imperative programs into dataflow graphs is presented, but we start with a short review of existing methods of flow analysis.

References

- Acke79. ACKERMAN, W. AND J.B. DENNIS (Jun 1979). *VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual*, Technical Report 218, MIT - Laboratory for Computer Science.
- Alla80. ALLAN, S.J. AND A.E. OLDEHOEFT (Sep 1980). A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language, *IEEE Transactions on Computers*, C-29.9, 826-831.
- Amam82. AMAMIYA, M., R. HASEGAWA, O. NAKAMURA, AND H. MIKAMI (Jun 1982). A List-Processing-Oriented Data Flow Machine Architecture, *AFIPS National Computer Conference 82*, 143-151.
- Arvi78. ARVIND, K.P. GOSTELOW, AND W. PLOUFFE (Dec 1978). *An Asynchronous Programming Language and Computing Machine*, Technical Report 114a, University of California, Irvine, Information and Computer Science Dept.
- Bowe81. BOWEN, D.L. (Apr 1981). *Implementation of Data Structures on a Data Flow Computer*, Ph.D. Thesis, Dept. of Computer Science - Victoria University of Manchester.
- Comt80. COMTE, D., N. HIFDI, AND J.C. SYRE (Oct 1980). The Data Driven LAU Multiprocessor System: Results and Perspectives, *IFIP80*, 175-180.

- Darl81. DARLINGTON, J. AND M. REEVE (Oct 1981). ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Conference on Functional Programming Languages and Computer Architecture*, 65-75.
- Glau78. GLAUERT, J.R.W. (Jan 1978). *A Single Assignment Language for Data Flow Computing*, M.Sc. Dissertation, Dept. of Computer Science - Victoria University of Manchester.
- Glau84. GLAUERT, J.R.W. (Aug 1984). High Level Dataflow Programming, in *Distributed Computing*, 43-53, ed. G.P. Jones, Academic Press.
- Kuck81. KUCK, D.J., R.H. KUHN, D.A. PADUA, B. LEASURE, AND M. WOLFE (Jan 1981). Dependence Graphs and Compiler Optimizations, *Eighth Annual Symposium on Principles of Programming Languages*, 207-218.
- McGr83. MCGRAW, J. ET.AL. (Jul 1983). *SISAL: Streams and Iteration in a Single-Assignment Language - Language Reference Manual Version 1.1*, Lawrence Livermore National Laboratory, Livermore.
- Olde78. OLDEHOEFT, A.E., S. ALLAN, S. THORESON, C. RETNADHAS, AND R.J. ZINGG (Mar 1978). *Translation of High Level Programs to Data Flow and their Execution on a Feedback Interpreter*, Technical Report 78-2, Department of Computer Science - Iowa State University.
- Otte81. OTTENSTEIN, K.J. (Oct 1981). *An Intermediate Program Form Based on a Cyclic Data-Dependency Graph*, CS-TR 81-1, Department of Mathematical and Computer Science - Michigan Technological University.
- Ping83. PINGALI, K. AND ARVIND (Sep 1983). *Efficient Demand-driven Evaluation (I & II)*, Technical Memo 242-243, MIT - Laboratory for Computer Science.
- Rich82. RICHMOND, G. (Oct 1982). *A Dataflow Implementation of SASL*, M.Sc. Dissertation, Dept. of Computer Science - Victoria University of Manchester.
- Whit78. WHITELOCK, P.J. (Oct 1978). *A Conventional Language for Data Flow Computing*, M.Sc. Dissertation, Dept. of Computer Science - Victoria University of Manchester.

Chapter 4

Program Flow Analysis

Due to the changing nature of efficiency demands, program analysis will be an important subject in the years to come. The efficient production of software is not the same as the production of efficient software and as long as neither the cost of programming nor the cost of computing power can be ignored there will be a need for both types of efficiency. With the ratio of programming versus processing cost constantly increasing, emphasis has shifted from computing efficiency to programming efficiency and modern software methods call for programs that are easy to understand, verify, and maintain. Many efforts have been directed towards producing tools to bring software practice closer to this goal without sacrificing too much computing efficiency. The availability of cheap microprocessors has led to a flurry of activity in the development of new tools. Programs that embody knowledge about the programming language have been developed to facilitate editing, testing, debugging, verification, and documentation. All these activities can be performed more easily within a programming environment that has more structural information available about the program being worked upon than merely its syntactic structure. Some form of flow analysis is usually required to obtain this structural information.

Program analysis also plays an essential role in bridging the gap between language and architecture. A program with a complete set of input data specifies a computation. The computation can be performed by an interpreter, but such a direct interpretation is, in general, highly redundant in the sense that the same result could be achieved by a much shorter computation. Usually intermediary programs, which we call *language processors*, are employed to reduce this redundancy. A compiler, for instance, transforms the program into a form which can be executed more efficiently. A so-called optimizing compiler is nothing more than a compiler that carries the transformation a bit further. Reducing the redundancy of the interpretation process involves the transformation into a more abstract form that is closer to the "meaning" of the program. This is the basic mechanism of program analysis.

The first phase, lexical and syntactic analysis, is well understood. A class of languages has been identified for which efficient parsers can be written and the syntax

of most programming languages is confined to this class. If a more elaborate transformation is required, a more powerful but less well understood analysis has to be performed. In most traditional methods two separate phases can be distinguished: *control flow analysis*, which is concerned with the order in which instructions are to be executed, and *data flow analysis*, which is concerned with the data dependencies in a program. Because the latter phase is the more complicated one, the complete analysis is often simply referred to as data flow analysis. In the method to be presented in the next chapters, and in other recently developed methods, this separation into two phases is dropped.

Before we can compare different methods of program flow analysis we need to have some notion of its applications. The next section elaborates on this concept of application by giving an example and a general model. The final section is a short overview of some of the existing methods. As a preliminary, we give a condensed presentation of the standard graph terminology used extensively in the rest of this thesis.

GRAPH TERMINOLOGY

A (directed) graph is a pair $\langle N, A \rangle$, where N is a finite nonempty set of *nodes* and A is a relation on N . Each pair $\langle x, y \rangle \in A$ is called an *arc from node x to node y* ; x is the *tail* and y the *head* of the arc. x is a *predecessor* of y and y a *successor* of x . A node without predecessor is called a *source*; a node without successor a *sink*. All other nodes are *interior*.

A *path* is a finite sequence of two or more nodes, such that there is an arc between each pair of subsequent nodes in the path. If there is a path from x to y , we say x is an *ancestor* of y and y is a *descendant* of x . In a *connected* graph each pair of nodes has a path between them or has a common ancestor or descendant. A *tree* is a connected graph, where no node has more than one predecessor. For trees the words *root*, *leaf*, *parent*, and *child* are used instead of source, sink, predecessor, and successor.

A *cycle* is a path in which the first and last nodes are the same. A graph is *acyclic* if it has no cycles. Each graph can be uniquely partitioned into subsets, where two nodes belong to the same subset if and only if there is a cycle to which they both belong. Such a subset is called a *strongly connected component*. The *acyclic condensation* of a graph is obtained when each strongly connected component and its internal arcs are replaced by a single node. A graph is *irreducible* if it contains three nodes x , y , and z , such that there is a path from z to x not containing y and a path from z to y not containing x . All other graphs are reducible.

4.1. Applications

Program flow analysis is usually applied to obtain an answer to a specific question. Constant propagation, for instance, is concerned with the question: "Which expressions can be evaluated independently of the input of the program?" Live-dead analysis is concerned with the question: "What will be the life span of each value created during program execution?" These separate concerns we will call *applications of flow analysis* or simply *applications*. In the literature this notion is sometimes referred to as a *technique* or *problem*.

EXAMPLE OF AN APPLICATION

Before discussing applications in general terms a simplified version of the *Value Approximation* application will be presented as an example. This application, which will receive a more elaborate treatment in chapter 7, is concerned with the question "What is the value, or range of values, of each particular variable occurrence?" However, this range is, in general, not effectively computable, so a reasonable upper bound has to take its place. In this simplified version we assume that data can only be of three types and that the value domains that appear in figure 4.1 are sufficient to describe the desired information. To make the application more interesting we assume that we are dealing with a language in which not only the value but also the type of a variable may vary. The analysis should label each item with the value domain that most precisely describes the range of values that an item can have.¹ If, for instance, an item could take on the values 5 and 8, its value domain should be *Integer*. If it could also be 6.5 its value domain should be *Numeric*.

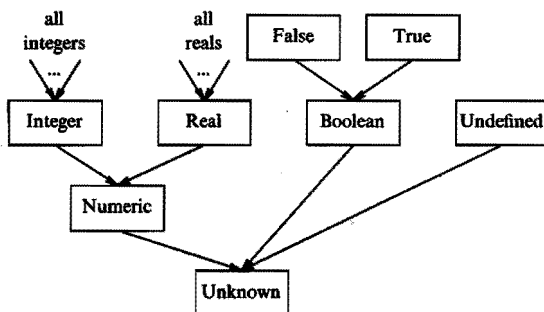


Figure 4.1. The assertion semilattice for the Value Approximation application.

Each box represents a possible value domain. A phrase like *all reals* represents an infinite set of boxes. The partial order defined on the value domains is indicated by the arrows. The value-domain *Unknown*, which is less than all other elements, is called the *bottom element*.

The analysis of a program starts with associating initial *assertions* with all arcs in a graph representation of the program. These assertions contain information about the values of variables and they should be valid regardless of which execution path is followed. Each assertion has the form "When control reaches this point variable V_1 is X_1 , ..., variable V_n is X_n ", in which each X_i is a value domain. The initial assertions contain only local information, which can be deduced from considering the operation represented by its tail in isolation. In fact, most of them contain no information.

The aim of the analysis is to obtain assertions for each arc as a result of the interaction of the initial assertions. The final assertions should not be weaker than the initial ones. This notion assumes a partial ordering of the assertions, which is also illustrated in figure 4.1. A set with a partial order and a bottom element is called a *meet semilattice*. The meet operation maps a pair of elements to their greatest lower bound. A *chain* is a monotonically increasing sequence of elements. A semilattice is *bounded* if all its chains are finite. The infinite semilattice in figure 4.1 is bounded.

The interaction between assertions is expressed in *propagation rules*, which are associated with nodes and specify how assertions are transformed when an operation is

1. In some cases even this approximation is not computable or would require unduly sophisticated analysis. In that case a less precise value domain is accepted.

executed. Figure 4.2 shows one such propagation rule and the assertions whose interaction it specifies.

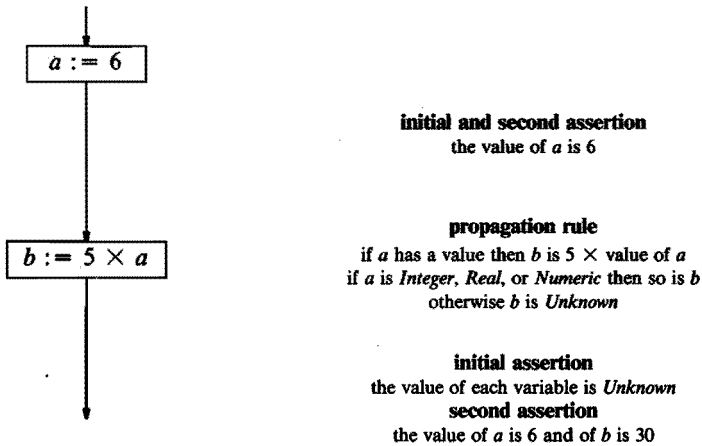


Figure 4.2. A segment of a (condensed) program graph.

Assertions and propagation rules are from the Value Approximation application. The second set of assertions is derived from the initial assertions by one application of the propagation rule.

The assertions described so far are *global* assertions: each assertion describes the total “state” of the program: when control reaches a particular point, it asserts something about every program variable. The initial assertions do not contain much information and can thus be encoded compactly, but when the information is propagated and combined, the assertions grow. The global assertions are wasteful in that every point receives information about the total state whereas only a small part of that information is relevant locally. The method described in the next chapter avoids this problem by choosing a different set of assertions that only encodes information that is of local interest. Not only do global assertions require a lot of storage, they also take up considerable processing time. One way of coping with this complexity is to limit the information per assertion to a few independently computable bits per variable. These can then be encoded by bit vectors for efficient storage and processing. Most of the applications considered traditionally lend themselves to this kind of representation and are therefore called *bit vector type* applications. In these applications propagation rules can be expressed in so called data flow equations, that specify the relation that should hold between the bit vectors of each node in the control flow graph. Solving the global flow problem is then equivalent to finding some solution satisfying the set of data flow equations. When comparing the computational complexity of different methods, it should be kept in mind that the size of the bit vectors (and consequently the cost of a bit vector operation) is usually proportional to the program size.

ABSTRACT APPLICATIONS

In the early days of optimization, particular applications of flow analysis and their relative merits were focal points of research. Later [Kild73] it was realized that many applications share some of the most important problems and the emphasis shifted to research in methods that can be useful for many applications. Such *generalized* analysis methods need some general notion of an application.

Each flow analysis problem can be seen as the association of a set of assertions concerning local properties with particular points in a program, and the propagation of this information through the program so that it can be checked for consistency or combined into more global assertions. We will consider an application to be a pair $\langle A, P \rangle$, where A is a set of *assertions* and P a set of *propagation rules*. Each assertion provides information about a particular property of a program and a propagation rule specifies the interaction between assertions.

Assertions are associated with arcs and propagation rules with nodes. In the general case the propagation rules associated with a node with p incoming and q outgoing arcs is a function from $A^{p+q} \rightarrow A^{p+q}$. The inputs and outputs of the function are the old and new assertions associated with the arcs. Two special cases are distinguished. In a *forward* application the information flows in the same direction as control and each propagation rule is a function from $A^{p+q} \rightarrow A^q$. In a *backward* application the information flows in the opposite direction and each propagation rule is a function from $A^{p+q} \rightarrow A^p$.

A solution consists of the association of a (final) assertion with each arc in the program satisfying all propagation rules. Not all solutions are good ones: a trivial solution for the problem illustrated in figure 4.2 could be the minimum assertion "The value of each variable is Unknown". The information contained in the initial assertions should not be lost, and to capture this notion a partial ordering is associated with the set of assertions and it is usually assumed that it forms at least a meet semilattice. This implies that there is a minimum assertion, which is implied by all other assertions and a meet operation, which extracts the information that two assertions have in common. A *good* solution is one which implies all initial assertions. It is desirable to obtain not just a good solution, but a *maximum* one, i.e. a good solution that is not implied by any other solution.

One way of obtaining a solution is by propagating information through the graph, each time using the propagation rule of a node to update the assertions on the associated arcs, until a stable situation is reached. In such an *iterative* method only individual assertions are changed and the propagation rules remain untouched. If assertions are never replaced by smaller ones (guaranteed if all propagation rules are order-preserving) and if the assertion lattice is bounded, it is certain that a good solution will be reached. A maximum solution will be reached when the application is distributive [Kild73]. Other, so called *elimination* methods summarize the effect of a whole subgraph by replacing a set of propagation rules by a new one. These methods are usually faster than iterative methods, but the class of applications that they can handle is more restricted. The set of propagation rules has to be closed under functional composition and pointwise meet. Cycles present problems because the effect of unbounded paths must be expressible as a propagation rule and it must be computable in a bounded number of steps. Rosen and Graham&Wegman have investigated the minimum requirements that guarantee a good solution using such a method [Grah76, Rose80].

4.2. Existing Methods

As indicated in the previous section, a flow analysis problem is solved in two steps.

- Assertions and propagation rules are associated with certain points in the program.
- Information is propagated through the program by combining assertions and/or propagation rules into new ones until a stable situation is reached.

The initial assertions and propagation rules describe the local effect of separate operations. This is trivial for atomic operations, but the local effect of a procedure call

can only be determined through extensive analysis. Flow analysis that does not concern itself with the relationship between procedures is called *intraprocedural*, all other analysis is *interprocedural*. If interprocedural analysis is omitted a conservative approximation of the effect of a procedure call must be used, which limits the quality of the information that can be obtained. In the rest of this chapter strategies for interprocedural and intraprocedural analysis are discussed separately.

4.2.1. INTERPROCEDURAL ANALYSIS

Interprocedural analysis is an active area of research and we give only an indication of its problems rather than attempt to survey its present state. Important articles in this field are [Alle74, Bart78, Rose79].

A normal procedure call (i.e. not a coroutine call) consists of two transfers of control: from the calling to the called procedure and back to the calling procedure. These jumps are not independent, since a call will never be followed by a return to another procedure. One consequence of this is that not every path through the call graph (the graph that expresses calling relationships between procedures) is a valid control path. The challenge of interprocedural analysis is to exploit this information about the control flow patterns to obtain a better solution. A simple but expensive method is *in line expansion*: each procedure call is replaced by a copy of the procedure body and only the intraprocedural analysis of the root procedure is required. Its obvious drawbacks are that much analysis is duplicated and that recursion cannot be handled.

A popular approach is to split the analysis into two phases. In the first phase a summary of the effect of each procedure is constructed by a rough analysis of its body, ignoring any procedure calls. A transitive closure algorithm is then used to incorporate all direct and indirect procedure calls into the summaries. In the second phase the final analysis is performed using the summary information whenever a procedure call is encountered. The quality of the method depends on the quality of the information gathered in the first phase, which in turn is limited by the fact that the local effect of a procedure call is necessarily overestimated.

In [Shar81] two methods are described which aim at removing this deficiency. The *functional approach* analyzes each procedure and expresses its effect in a set of relations between assertions at entry and exit points. Since these relations are interdependent, iteration is required to arrive at a fixed point. This method belongs to the elimination methods and is only useful for a restricted class of applications (see previous section). In the *call string approach* procedure call and return are treated as separate jumps, but an identification of each procedure call encountered during information propagation is tagged onto the propagated information. When a return is encountered this call string tag is used to select the correct control path. A generalization of both methods is described in [Jone82].

Most methods simplify the problem of interprocedural analysis by excluding those language features that lead to serious complications. One complication is *aliasing*, which arises when different access paths (such as variable names) refer to the same object. It can occur if the language allows pointer values or call-by-reference parameters. A second complication arises when it is statically (i.e. during analysis) difficult to determine which procedure is being called. This can occur if the language allows variables or parameters to have procedures as values or when operators and procedure names are overloaded. An extensive treatment of these problems appears in [Weih80] where it is shown that obtaining precise information in the presence of procedure variables is P-Space hard.

4.2.2. INTRAPROCEDURAL ANALYSIS

The many strategies that have been proposed for flow analysis fall into groups distinguished by the level of program representation operated upon. It is still a matter of debate which level is most appropriate. The choice is between the source text, the generated code, or any of the levels in between. Analysis of the source text always incorporates some form of lexical and syntactical analysis. Analysis of the generated code is the natural domain for machine dependent optimization; the work that has been done in this area is rather ad hoc and does not have much general applicability. Therefore, most general methods operate on some intermediate level. Ideal would be a representation in which all information that is not helpful for the analysis has been removed and all information that can be helpful is easily retrievable. Although many intermediate representations can be devised, two levels are of particular importance to flow analysis:

- In a *branch level* representation the hierarchical structure of the program has been lost and the control flow is entirely encoded by jumps. An example is the representation in three address code, where each instruction corresponds to a typical machine instruction; the difference with assembly level is that register allocation has not yet been performed. Analysis methods that work on this level are called *low level*.
- In a *syntax level* representation the program has the form of a graph supplemented with tables. The graph is usually a tree such as a parse tree. The nesting of statements, which has an important influence on the analysis, is directly reflected in the graph structure, which is not cluttered by lay-out, variable names, and other details not relevant to flow analysis. Analysis methods that work on this level are called *high level*.

As Rosen, who first coined the terms for this distinction [Rose77], points out, there has traditionally been a bias towards low level methods. This is partly due to the fact that flow analysis was almost always aimed at optimization and the programs whose optimization was most crucial were written in FORTRAN. The relation between control flow hierarchy and syntax is virtually absent in FORTRAN IV and the structure encoded in a parse tree is therefore of little help for flow analysis.

Both representations contain references to variables. Some of these represent an update of the value of the corresponding memory location; this is called a *definition*. Other references refer to the present value of the memory location; this is called a *use*. During program execution the outcome of each use is determined by at most one definition: the last definition that assigned a value to the particular variable. This does not have to be the same definition at every run of the program. Associating each use with all definitions that could affect its outcome is called *use-definition chaining*. For languages that allow a variable to be updated at more than one location in a program this is not a trivial process and is called use-definition or *data-dependency* analysis. An elaborate data-dependency analysis can serve as the backbone of flow analysis, as will be shown in the next chapter.

Low Level Methods.

In a branch level representation each procedure is represented by a list of instructions. Some of these are labeled and some prescribe a transfer of control to a labeled instruction of the same procedure. This representation can thus be treated as a graph, called the *program graph*, in which each instruction is a node and each arc a possible transfer of control, including the default transfer to the next instruction. Each procedure can be partitioned into a set of *basic blocks*, where each block is a set of

consecutive instructions guaranteed to be executed in a strictly linear fashion. In the *control flow graph* of a procedure each node corresponds to a basic block and each arc to a possible transfer of control.

Most low level methods are remarkably alike. The most general method for intraprocedural analysis consists of the following steps:

Control Flow Analysis:

Partitioning of each procedure into basic blocks

Construction of the control flow graph

Analyzing the control flow graph

Data Flow Analysis:

Intrablock analysis

Global data flow analysis

Control flow analysis is needed because in a branch level representation the control flow structure is not explicitly available: the transfer of control is exclusively indicated by unrestricted jumps and the control flow patterns that this may lead to have to be uncovered by analysis. The partitioning of each procedure into basic blocks and the construction of the control flow graph is rather straightforward. Analysis may then be performed on this graph to obtain structural information to be used in the global data flow analysis. The data flow analysis phase is initialized by attaching assertions and propagation rules to arcs and nodes of the program graph. The analysis within a basic block is straightforward and usually all blocks are first analyzed separately and assertions attached to each block summarizing the information of all instructions in the block. The methods differ in the way global data flow analysis is then performed in the control flow graph.

The simplest method is the *arbitrary order iteration* in which all basic blocks are processed in arbitrary order and the information for each block is updated to take into account all incoming and outgoing arcs. The whole process is repeated until no information is updated in one complete iteration. Under reasonable conditions (the assertion lattice is bounded and all propagation rules are order preserving) this process is guaranteed to terminate and to produce a good solution, although not necessarily the maximum one. In the worst case n iterations are needed, each consisting of a number of assertion updates on the order of n .¹ One usually says that the complexity of this method is quadratic, counting only the number of assertion updates.² So much theoretical work has been focussed on finding an alternative with a lower worst case complexity for this part of the algorithm, that it has made the development of the field somewhat lopsided. Kennedy gives a good survey of this work [Kenn81].

Most other methods are only applicable to programs that have a reducible control flow graph. Well-structured programming languages guarantee reducible control flow graphs, but even in a language such as FORTRAN almost all programs are reducible [Cock77]. A variation of the arbitrary order iteration, but still an iterative method, is described in [AhUI76]. It is shown that for reducible control flow graphs an order of processing called *node listing* can be found that reduces the complexity to the order $n \log n$. If the set of propagation rules for the application is rich enough, elimination

1. In complexity measures n will refer to the size of the program in some reasonable metric, e.g. the number of lexical tokens.

2. It should be stressed that often the cost of an update of a global assertion is also of the order of n , in which case the complexity of this method is more properly characterized as *cubic*.

type methods, which operate directly on propagation rules, can be used. The best known example is *interval analysis* [Alle76] : to uncover its loop structure the control flow graph is structured into nested subgraphs called intervals. For forward applications the nested intervals are then processed from the inside out, each time replacing a whole interval by one node with assertions and propagation rules that summarize the complete effect of the interval. For backward applications the order is reversed. The worst case complexity of this method is still quadratic.

Faster methods have been designed, but we will limit ourselves to citing their references. Graham and Wegman [Grah76] developed the *path compression* method, which has a worst case complexity of order $n \log n$, but is in practice usually linear. Tarjan [Tarj81] and Rosen [Rose80] introduced restrictions on the class of applications that allow an almost linear algorithm. For a somewhat more restricted class of control flow graphs, linear methods using *graph grammars* are available [Farr75].

High Level Methods.

In a syntax level representation each procedure is represented by a graph. The obvious representation is a *parse tree*, where each interior node corresponds to an application of a rule in the grammar of the language. If the language contains no jumps, all information about the structure of the control flow is available directly in this tree. Consider, for instance, the node corresponding to a **while** statement: the descendants of this node form exactly the set of instructions that belong to the cycle in the control flow induced by the **while** statement. For such languages the parse tree can thus serve the same role as the analyzed control flow graph in low level methods. Even if there is no complete correlation between the control flow and the parse tree, because the language contains jump statements, a high level method might still be advantageous. In some cases the jumps are restrained in a way that requires only slight adjustments to the algorithm (e.g. no backward jumps). In other cases the language is rich enough to make it a reasonable assumption that difficult jumps occur so infrequently that an expensive analysis can be afforded for each occurrence. A high level method dealing with frequent unrestricted jumps is described in [Born84].

For applications that are meant to generate messages for the programmer, choosing a representation close to the source text offers another advantage besides the virtual disappearance of control flow analysis. If such an application is implemented as a low level method, expressing the information in terms of the source text would require some form of transformation back to the source text.

A convenient way to operate on a tree is to process its nodes during a *recursive descent* traversal. All nodes of the tree are processed depth first starting at the root. The algorithm applied at each node contains a recursive application of the same algorithm to each of its children. The nature of processing at a node in the parse tree is usually determined by the type of the node (its syntactic category). Since the tree is determined by a grammar, the description of the algorithm can conveniently be combined with the grammar to obtain an *attribute grammar*.

Rosen has given an extensive theoretical treatment of high level methods. In [Rose77] it is shown that the *a priori* assumptions made in recursive descent methods are valid for all programming languages without backward jumps. In such languages the effect of each operation can be summarized by a graph whose structure is determined by the type of operation. When arbitrary jumps are allowed the structure of a graph sometimes has to be derived during analysis. Although this increases the cost, it does not effect the structure of the method.

Despite the intuitive appeal of high level methods, the pertinent literature is limited. The first full scale application of the recursive descent method was the optimizing compiler for the BLISS language [Wulf75], a well-structured language with only forward jumps. In that compiler the detection of feasible optimizations occurs during the construction of the parse tree and the actual optimization during a second traversal of the tree.

The application of attribute grammars was first investigated by Jazayeri&Babich. In [Babi78] one forward and one backward application are described for a simple language with iteration and unconstrained jumps. The anomalies in the control flow induced by these language features are dealt with by repeating the complete tree traversal until the assertions stabilize. It is shown that the number of iterations is bounded by the number of loops and backward jumps. Also the MUG2 compiler generating system [Wilh81] uses (modified) attribute grammars extensively.

Ferrante&Ottenstein [Ferr83] recently developed a high level method that transforms the program into a new representation. For each procedure a so called *extended data flow graph* is constructed, which encodes both data flow and control flow dependencies. The incoming arcs of each node represent either a operand or the predicate that controls its execution. They show that this is an attractive program representation by describing four applications: code motion, constant propagation, common subexpression elimination, and detection of induction variables. Their method is being extended to include unrestricted jumps, but so far it has been limited to intraprocedural analysis. Their approach is quite similar to the one described in the next chapters.

References

- AhU176. AHO, A.V. AND J.D. ULLMAN (Dec 1976). Node Listings for Reducible Flow Graphs, *Journal for Computing Systems Science*, 13.3, 286-299.
- Alle74. ALLEN, F.E. (Aug 1974). Interprocedural Data Flow Analysis, *Proceedings IFIP Congress 74*, 398-408.
- Alle76. ALLEN, F.E. AND J. COCKE (Mar 1976). A Program Data Flow Analysis Procedure, *Communications of the ACM*, 19.3, 137-147.
- Babi78. BABICH, W.A. AND M. JAZAYERI (1978). The Method of Attributes for Data Flow Analysis, *Acta Informatica*, 10, 245-272.
- Bart78. BARTH, J.M. (Sep 1978). A Practical Interprocedural Data Flow Analysis Algorithm, *Communications of the ACM*, 20.9, 724-736.
- Born84. BORN, R. VAN DEN (Feb 1984). *Struktuur Behoudende Data Flow Analyse op Programma's met GOTO-Statements*, Noot CS-N8401, Centre for Mathematics and Computer Science, Amsterdam, (in dutch).
- Cock77. COCKE, J. AND K. KENNEDY (Nov 1977). An Algorithm for Reduction of Operator Strength, *Communications of the ACM*, 20.11, 850-856.
- Farr75. FARROW, R.K., K. KENNEDY, AND L. ZUCCONI (Nov 1975). Graph Grammars and Global Program Flow Analysis, *Seventeenth Annual IEEE Symposium on Foundations of Computer Science*.
- Ferr83. FERRANTE, J. AND K.J. OTTENSTEIN (Jan 1983). A Program Form Based on Data Dependency in Predicate Regions, *Tenth Annual Symposium on Principles of Programming Languages*, 217-236.
- Grah76. GRAHAM, S.L. AND M. WEGMAN (Jan 1976). A Fast and Usually Linear Algorithm for Global Flow Analysis, *Journal of the ACM*, 23.1, 172-202.

- Jone82. JONES, N.D. AND S.S. MUCHNICK (Jan 1982). A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures, *Ninth Annual Symposium on Principles of Programming Languages*, 66-74.
- Kenn81. KENNEDY, K. (1981). A Survey of Data Flow Analysis Techniques, in *Program Flow Analysis - Theory and Applications*, 5-54, ed. S.S. Muchnick & N.D. Jones, Prentice Hall.
- Kild73. KILDALL, G.A. (Oct 1973). A Unified Approach to Global Program Optimization, *First Annual Symposium on Principles of Programming Languages*, 194-206.
- Rose77. ROSEN, B.K. (Oct 1977). High-Level Data Flow Analysis, *Communications of the ACM*, 20.10, 712-724.
- Rose79. ROSEN, B.K. (Apr 1979). Data Flow Analysis for Procedural Languages, *Journal of the ACM*, 26.2, 322-344.
- Rose80. ROSEN, B.K. (Feb 1980). Monoids for Rapid Data Flow Analysis, *SIAM Journal on Computing*, 9.1, 159-196.
- Shar81. SHARIR, M. AND A. PNUELI (1981). Two Approaches to Interprocedural Data Flow Analysis, in *Program Flow Analysis - Theory and Applications*, 189-234, ed. S.S. Muchnick & N.D. Jones, Prentice Hall.
- Tarj81. TARJAN, R.E. (Jul 1981). Fast Algorithms for Solving Path Problems, *Journal of the ACM*, 28.3, 594-614.
- Weih80. WEIHL, W.E. (Jan 1980). *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables*, RC 8060, IBM.
- Wilh81. WILHELM, R. (1981). Global Flow Analysis and Optimization in the MUG2 Compiler Generating System, in *Program Flow Analysis - Theory and Applications*, 132-159, ed. S.S. Muchnick & N.D. Jones, Prentice Hall.
- Wulf75. WULF, W., R.K. JOHNSON, C.B. WEINSTOCK, S.O. HOBBS, AND C.M. GESCHKE (1975). *The Design of an Optimizing Compiler*, Elsevier North-Holland, New York.

Chapter 5

The Demand Graph Method

This chapter introduces a new method for flow analysis called the *demand graph method*. The name is derived from the representation of a program as a so-called *demand graph*, a data structure that plays a central role in the analysis. An analyzer that uses this method consists of four phases: syntactic analysis, demand graph construction, demand propagation, and extraction. The first two phases can be shared by analyzers that implement different applications. These therefore constitute the *general* part and the remaining two phases the *application specific* part.

The first section explains why a new method was developed and how it is related to other methods for flow analysis. The second section gives an outline of the whole process.

5.1. Evolution of the Demand Graph Method

As we saw in the previous chapter, early work in flow analysis was exclusively concerned with optimization; the emphasis was on experimenting with specific program transformations. Later work concentrated on improving the methods for obtaining the required information. It was soon realized that the study of flow analysis algorithms could be separated from the study of particular applications. This separation, however, was not visible in the implementation. Although implementations for different applications have much in common, extracting the general part of one implementation so that it could be used in a new application was hard because the general and the specific parts were intimately intertwined. An analogy from the related field of parsing may clarify this. Although two recursive descent parsers for two different languages have much in common, it is hard to extract the general part from one such parser to use it in the construction of a new one. A new methodology, generating a parser on the basis of the grammar, was needed to separate the general from the specific. Similarly, in the field of flow analysis, the demand graph method provides a general framework in which the implementation of various flow analysis applications can be smoothly integrated.

The borderline between general and application specific is to some degree arbitrary. Since most ambitious applications require a more or less elaborate data-dependency analysis (or *use-definition analysis*), it was decided that the general part of the method should perform such an analysis and express its result in a data structure that would be convenient for a wide range of applications. The contribution of the method is mainly this new data structure and the separation of use-definition analysis from the rest of an application. It makes applications easier to program, but, of course, difficult problems in certain applications do not disappear simply because another program representation has been chosen.

It would have been desirable if the experiment would not have been tied to a particular language; yet developing a language independent method was considered too ambitious a project. Instead a locally developed and mostly locally known programming language, called SUMMER, is used, but the techniques developed in implementing the method for this language are transferable to implementations for other imperative languages. Since SUMMER has no unrestricted jump, an exception is made for languages such as FORTRAN IV, in which *goto* is the dominant control flow instruction.

For the reasons described in the previous chapter, a high level method was chosen. In summary, the input languages considered interesting are well-structured and the method should be useful for a wide variety of applications, including those that express their results in messages to the users. Since propagation rules and initial assertions in a high level method are related to the syntax rules of the language, attribute grammars form an attractive formalism: the description of the flow analysis algorithm is intertwined with the grammar, such that each production rule is followed by a set of *attributes* and a set of *attribute rules*. The attributes describe the set of assertions that can be attached to nodes of that type in the parse tree. The attribute rules are the propagation rules that specify how attributes are influenced by other attributes attached to parent or child. Attribute grammars are attractive because they describe the flow analysis algorithm purely in terms of local effects. There is a reasonable match between the connectivity of the parse tree and the locality properties of many applications, but major discrepancies still remain. It is not uncommon to find a long path between the node in which information originates and the nearest node in which it is used, forcing all intermediary nodes to retain and transmit information that is irrelevant to them. In use-definition analysis, for instance, when a use is encountered, the information about all previous definitions in the program has to be available and therefore transmitted through all nodes. Ganzinger [Ganz74] attacks this problem by proposing *modified attribute grammars*, which allow global attributes. This constitutes a relaxation of the strong modularization imposed by the original attribute grammars, akin, in purpose as well as consequence, to the introduction of global variables in a well-structured programming language. One consequence is that in this method the programmer needs to be more specific about the evaluation order in order to guarantee a correct maintenance of global attributes.

The approach used in the method proposed here is to construct a graph, whose connectivity, compared to that of the parse tree, is in better accordance with the locality properties of many applications. For a well-structured language the structure of the parse tree reflects to a great extent the control flow. Many applications, however, are more sensitive to data dependencies in the program and therefore a representation more directly reflecting the flow of data is more appropriate. One such representation is the data flow graph described in section 2.2.

The step from tree to graph is important. In the type of languages that we are considering the order of statements is significant: interchanging two statements in a program may alter its meaning. However, in general this evaluation sequencing is overspecified: sometimes statements may be reordered without affecting the meaning of the program. The meaning is therefore better expressed by a partial ordering of statements. Such a partial ordering can easily be expressed in a data flow graph.

In the demand graph method a program is represented as a data flow graph with all the arcs reversed. I have called this representation a *demand graph*. Roughly speaking a demand graph can be obtained from a parse tree by adding appropriate arcs from uses of variables to their definitions and removing all unnecessary sequencing constraints. Not all data-dependencies can be determined during static analysis, since they are influenced by conditional control flow. We refer to this problem as (*static*) *ambiguity*. In those cases, rather than a simple data-dependency a more elaborate subgraph is inserted that is attached to all relevant definitions. The nodes of this subgraph encode the static ambiguity.

5.2. Language-Independent Aspects of the Demand Graph Method

This section presents an overview of the four phases of the demand graph method. The description in this section limits itself to the general structure of the method and to the analysis of constructs that are commonly found in imperative languages. Features of the implementation due to peculiarities of the input language will show up in the next two chapters, which are devoted to a more detailed description of the most important phases (i.e. demand graph construction and demand propagation).

5.2.1. SYNTACTIC ANALYSIS

The lexical and syntactic analysis is standard and any parser that converts a program text into a parse tree representation is suitable. The present implementation uses the existing parser, which produces a condensed form of a parse tree, called the (*abstract*) *syntax tree*.¹ A syntax tree is a more convenient starting point for the analysis than a parse tree, because many of the nodes that are artifacts of the particular grammar and that are irrelevant to the meaning of the program have been removed. Figure 5.1 illustrates the difference by depicting an expression, its parse tree, and the corresponding syntax tree.

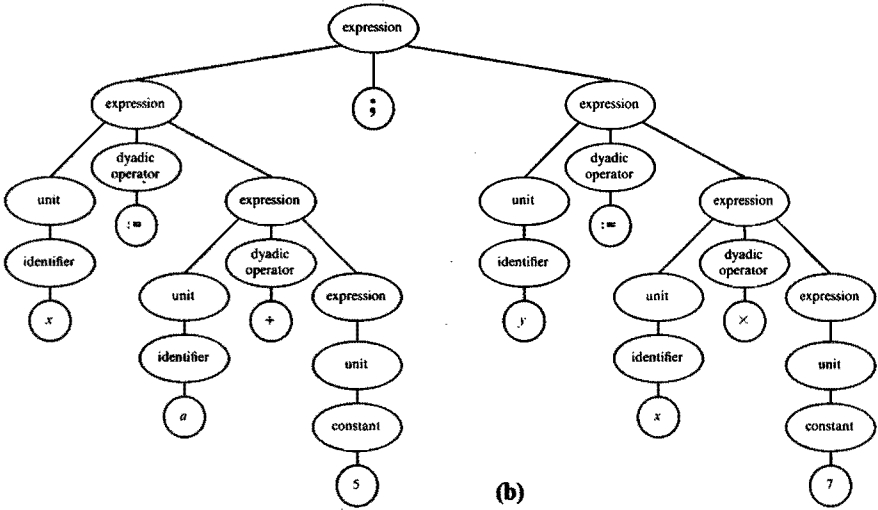
During syntactic analysis some rudimentary information may be collected to facilitate demand graph construction. In the current implementation the call graph is constructed and the syntax tree of each procedure is descended to record uses and definitions of global variables. At the end of the syntactic analysis the transitive closure of this information is computed to be used during the construction of demand graphs for recursive procedures (see below).

1. In the literature the two types of trees and their names are often confused. We will follow the terminology used by Aho&Ullman [AhU177].

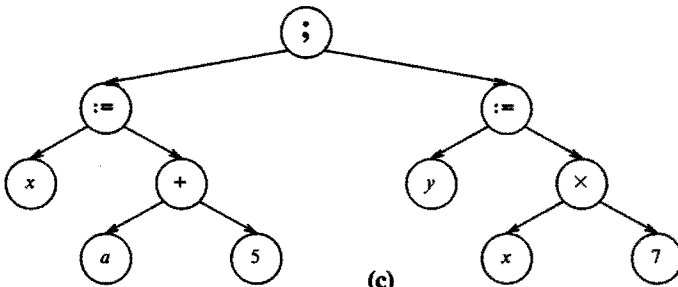
$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle (; \langle \text{expression} \rangle)^* |$
 $\langle \text{unit} \rangle [\langle \text{dyadic-operator} \rangle \langle \text{expression} \rangle]$.
 $\langle \text{dyadic-operator} \rangle ::= ' := ' | '+' | '\times'$.
 $\langle \text{unit} \rangle ::= \langle \text{constant} \rangle | \langle \text{identifier} \rangle$.
 $\langle \text{constant} \rangle ::= '5' | '7'$.
 $\langle \text{identifier} \rangle ::= 'a' | 'x' | 'y'$.

$x := a + 5 ;$
 $y := x \times 7$

(a)



(b)



(c)

Figure 5.1. From source text to syntax tree.

- (a) Expression and grammar: a fragment of a program source and the relevant part of a grammar.
- (b) Parse tree. For each application of a production a node is produced. Each interior node is labeled with a non-terminal and leaves are labeled with a terminal. The exact program and the relevant part of the grammar can be reconstructed from this representation.
- (c) Syntax tree. Interior nodes are labeled with operators and leaves are labeled with operands.

5.2.2. DEMAND GRAPH CONSTRUCTION

A syntax tree can be converted into a demand graph by adding extra nodes and arcs that encode data dependencies, and by removing control flow nodes and arcs that are not essential to the meaning of the program. Since an elaborate data-dependency analysis already implies the detection of unnecessary sequencing constraints, the latter part of the transformation is simple. The new data-dependency connections are such that superfluous nodes and arcs are not reachable from the *source-of-demands*, which is the common ancestor of all nodes corresponding to output expressions. The demand graph is defined as all nodes and arcs reachable from the source-of-demands, so code that does not contribute in any way to the output of the program is left out automatically. For reasons of symmetry all nodes of the demand graph have a common descendant, the *sink-of-demands*. Nodes that do not in any way construct a new value are not part of the demand graph. This is fully determined by their type: VARIABLE and ASSIGN nodes, for instance, are left out, while a PLUS node constructs a new value and may therefore become part of the demand graph.

Figure 5.2 shows an example of the transformation of a syntax tree into a demand graph.

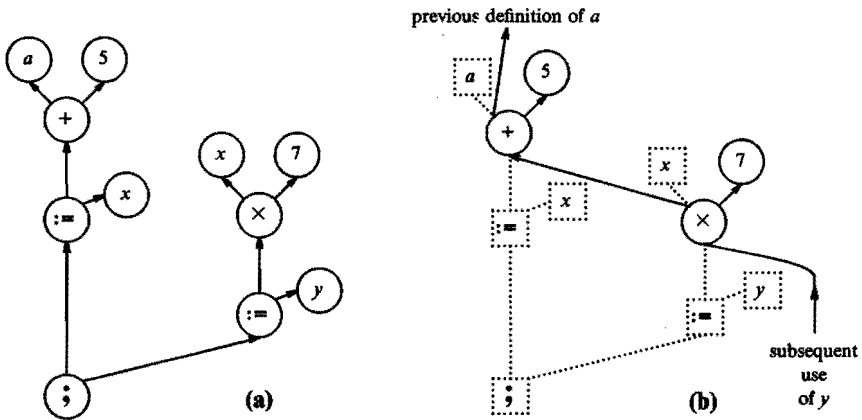


Figure 5.2. From syntax tree to demand graph.

(a) The same syntax tree as in figure 5.1, but now drawn with the leaves on top and the root at the bottom and with the right-hand side of an ASSIGN node (nodes marked with ':=') on the left and the left-hand side on the right. In this way the normal evaluation order of expressions is in better correspondence with a left-to-right and top-to-bottom order in the illustration.

(b) The corresponding demand graph. Only the solid nodes and arcs are part of the demand graph. The arcs in this figure that are not in (a) are data-dependency arcs, e.g. the new arc from the \otimes node to the \oplus node is the data-dependency arc for x .

It is interesting to note that the demand graph for this expression is indistinguishable from that for the expression $y := (a + 5) \times 7$ (provided that the value of x is not used later in the program). This phenomenon of identical demand graphs for different expressions occurs frequently. The demand graph construction defines an equivalence relation on the set of programs. In that sense it extends the parsing process: abstracting from the representation of a program and drawing closer to the function it represents.

Converting control flow information into data flow information can be an elaborate transformation. The perfect demand graph, i.e. a graph without any superfluous sequencing constraints, may for some programs be incomputable. In practice a safe

approximation has to be constructed, but how close this approximation should be to the ideal depends on practical considerations. Extra analysis can often improve an approximation, but the closer the approximation, the smaller the set of applications for which the extra analysis is beneficial.

The syntax tree is processed with a recursive descent algorithm: it starts at the root of the main program and processes all descendants mostly from left to right, calling the appropriate analysis¹ procedure, depending on the type of the node. Generally speaking the order of processing corresponds to the control flow except when control flow is cyclical.

Chainers.

The complicated part of demand graph construction is the building of appropriate use-definition graphs. This is controlled by a set of objects called *chainers* and *cocoons*. Chainers are created and destroyed in conjunction with cocoons, which are described below. During demand graph construction one chainer is always designated as the *current chainer*. In straight-line code the use-definition chaining is simple and only the current chainer is involved. It contains information about all the definitions encountered so far during the analysis of the straight-line segment. When a definition is encountered the chainer is informed that the current value of the variable can now be obtained from the last encountered node that produces a value (we say the variable now *lives* at this node). When a use is encountered the chainer is requested to construct an arc from the parent node to the node where the variable currently lives.

As an example we will follow the (simplified) processing of the tree fragment of figure 5.2. In this description the term *value node* refers to a node that constructs a new value, while *chainer* refers to the current chainer.

- process (1)
- process first operand (2)
 - process right-hand side (3)
 - process left operand (4)
 - ask chainer to construct arc from parent node (3) to node where *a* lives
 - process right operand (5)
 - inform chainer that a value node is encountered
 - inform chainer that a value node is encountered
 - process left-hand side (6)
 - inform chainer that *x* now lives at the last encountered value node
- process second operand (7)
 - process right-hand side (8)
 - process left operand (9)
 - ask chainer to construct arc from parent node (8) to node where *x* lives
 - process right operand (10)
 - inform chainer that a value node is encountered
 - inform chainer that a value node is encountered
 - process left-hand side (11)
 - inform chainer that *y* now lives at the last encountered value node

Note that arcs are only created from a use to the previous definition of the same variable. If in a straight-line segment the sequence use-definition-use-definition for one variable occurs, the first use is connected to the first definition and the second use to

1. If the meaning is sufficiently clear from the context we use *analysis* as a synonym for demand graph construction.

the second definition. The two definitions are unrelated and the fact that the two groups employ the same variable name has no influence on the demand graph. It is as if the variable name in the second group has been changed to enforce a single assignment discipline. This is the case for a full definition, i.e. a definition that completely replaces the old value of the variable by a new value. The use-definition chains are somewhat different for a *partial definition*, which is a modification of part of a structured object, e.g. an update of an array element. For a partial definition an additional arc is constructed to the previous definition to reflect the fact that not all information, previously stored in the object, has been lost.

Cocoons.

Some expressions need special treatment because of their effect on the use-definition chaining. Examples are conditionals, loops, and procedure bodies. Whenever during the traversal of the syntax tree, one of these special expressions is encountered, a new object, called a *cocoon*, and one or more chainer are created. These objects constitute a new environment in which the subgraph corresponding to the expression can be constructed in isolation from the rest of the demand graph. There are different kinds of cocoons corresponding to the different kinds of special expressions. Each special expression contains one or more subexpressions, called *branches*. For each branch a chainer is created, which is designated as the current chainer when that branch is analyzed. When all branches have been analyzed, a series of separate demand graphs, one for each branch, is available, and a series of chainers, each containing all the necessary information about the "input" and the "output" of a branch. The output information concerns the *exposed definitions*, i.e. definitions that are not obscured by a later definition in the same branch and that consequently may represent a value that is referred to from outside the expression. Input gives rise to an *exposed use*, i.e. a use of a variable that has no previous definition in the same branch.

After all branches have been analyzed the cocoon is *dissolved*, which involves the creation of two series of *interface nodes*, one for the input and one for the output, and the connection of these to the subgraphs and the surrounding graph. For each variable for which some branch contains an exposed use an input interface node is constructed and for each variable for which some branch contains an exposed definition an output interface node is constructed. The type of interface nodes and the way they are connected depends on the type of the cocoon. Figure 5.3 gives a summary.

expression	input	output
if and case	MERGE	BRANCH
while	ENTRY-LOOP	EXIT-LOOP
procedure	PARAMETER	RESULT
procedure call	CALL-IN	CALL-OUT

Figure 5.3. Interface nodes created during demand graph construction.

Conditionals.

When conditional control flow is involved the use-definition chaining becomes less straightforward than suggested in the previous section. After a conditional expression like

if test then $a := 7$ else $a := 9$ fi

it is not clear to which definition a subsequent use of a should be linked. There are not

one but two “previous definitions”, i.e. definitions that for some possible control flow path would be the previous definition of a . This static ambiguity is encoded in a **BRANCH** node to which subsequent uses are linked rather than to the definitions directly. Since these **BRANCH** nodes play a crucial role in the demand graph method we will take a moment to reflect on their significance.

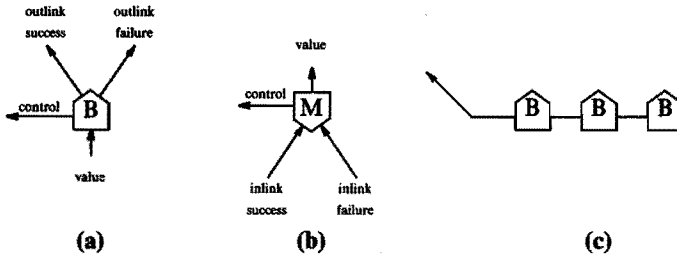


Figure 5.4. **BRANCH** and **MERGE** nodes.

- (a) A **BRANCH** node has an incoming *value* arc, an outgoing *control* arc, which leads to a node that will provide a signal, and a number of outgoing *outlink* arcs, which lead to the previous definitions.
- (b) A **MERGE** node has an outgoing *value* arc, an outgoing *control* arc, and several incoming *inlink* arcs.
- (c) A group of **BRANCH** or **MERGE** nodes connected to the same control may be drawn connected to avoid clutter. *Control* arcs may be drawn on either side of the node.

Most methods for use-definition analysis would treat the previous definitions of a particular variable as an unstructured set and would simply connect a use to each one of them. In this way information about the conditions under which a particular definition from the set will be selected is lost and cannot be used in subsequent analysis. Such an encoding would make it hard, for instance, to uncover the fact that after the expression

$$\text{if } x \leq 0 \text{ then } x := 1 - x \text{ fi}$$

x is positive.

In the demand graph method the set of previous definitions of each variable, the conditions under which a definition is selected, and the relations between them are treated as a whole. The information is coded as an acyclic graph considered to be part of the demand graph with the effect that at each point in the program the set of previous definitions of a variable is always represented by one single node. The algorithm can without ambiguity refer to the “defining node” of a variable: as soon as ambiguity arises it is removed by encapsulating it in a **BRANCH** node. Ambiguities are thus not allowed to propagate, a strategy which has proven to be quite advantageous. Only when aliasing is involved the propagation cannot always be confined, as we shall see later.

Figure 5.5(a) shows the cocoons and chainers involved in the construction of the demand graph for a simple conditional expression. Note that the *test* expression is analyzed outside the cocoon, since, during program execution, its evaluation is at the same level as the surrounding expression, i.e. it is evaluated whenever its surrounding expression is evaluated. Figure 5.5(b) shows the resulting demand graph; note that the **BRANCH** node fully encodes the effect of the conditional construct. The **IF** node is therefore left out of the demand graph.

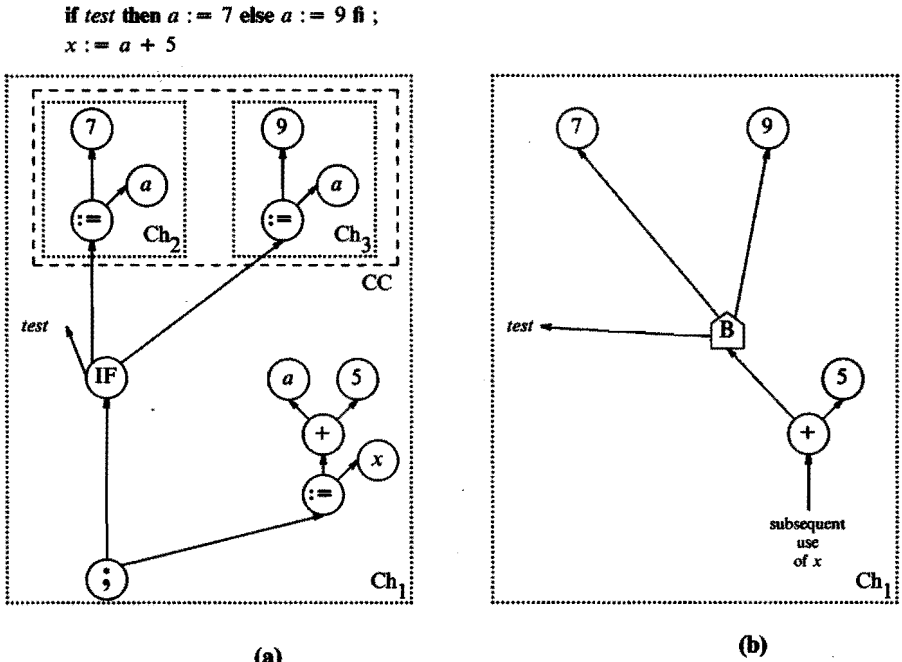


Figure 5.5. Analyzing a simple conditional expression.

(a) The syntax graph with the chainers and cocoons that are involved in its analysis. Ch_1 is the current chainer when, during analysis, the conditional expression (represented by the node marked with IF) is encountered. The conditional cocoon CC (the dashed box in the figure) is then created with the two chainers Ch_2 and Ch_3 , which serve in turn as current chainer during the analysis of the two branches. After the branches have been analyzed Ch_2 and Ch_3 contain the information that the 7 and the 9 nodes are exposed definitions of a . The cocoon CC is then dissolved, which involves the creation of a BRANCH node for each variable that is defined in any of the branches (in this case only a) and connecting it to the exposed definitions in each branch. The surrounding chainer Ch_1 , which is designated as the current chainer again, is then informed that the variable now lives at the newly created BRANCH node.

(b) The resulting demand graph. The left operand of the \oplus node is 7 or 9 depending on the outcome of test. This static ambiguity is encoded by the BRANCH node which has outgoing arcs to the two constants and to the controlling expression that will resolve this ambiguity at run time.

The input interface nodes of a conditional expression are MERGE nodes. They are included in the demand graph for reasons of symmetry and to facilitate its reversal into a data flow graph. Figure 5.6 illustrates the analysis of the conditional expression

```

if test
then  $x := x \times x$ 
else  $x := y ;$ 
        $t := 3$ 
fi
    
```

which has exposed uses for x and y . Since the expression assigns the constant 3 to the variable t in the else branch but does not define t in the then branch, a subsequent use of t should be connected to the definition of t previous to the conditional expression as

well as to the constant 3. To encapsulate this ambiguity in one BRANCH node the conditional expression is considered to use t . The effect is the same as if the dummy assignment " $t := t$ " would have appeared in the **then** branch. This simulated use of t in the **then** branch we call an *induced use*.

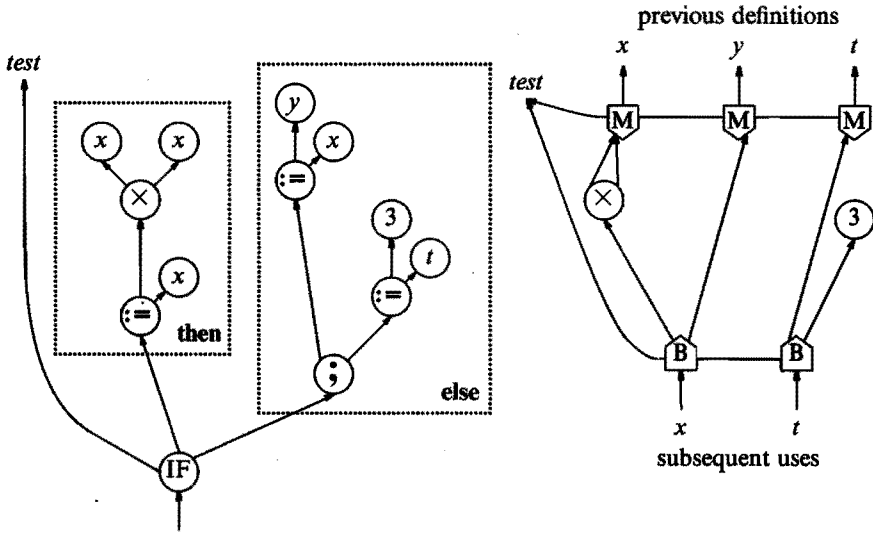


Figure 5.6. A conditional expression with exposed and induced uses.

(a) x is defined in both branches, while t is only defined in the **else** branch. Since the use of y in the **else** branch has no previous definition in the same branch it is an exposed use. Similarly for the uses of x in the **then** branch.

(b) The resulting demand graph. **BRANCH** nodes encode the static ambiguity for x and t . The **BRANCH** node for the latter variable induces an exposed use in the **then** branch, since that branch does not contain a definition for t . Each exposed use is connected to a **MERGE** node, which in turn is connected to the previous definition in the surrounding expression. Multiple exposed uses of the same variable in one branch are connected to the same input port of a **MERGE** node. An input port of a **MERGE** node may have no incoming arc, whereas each output port of a **BRANCH** node has an outgoing arc. Each **MERGE** and **BRANCH** node has an arc leading to the controlling expression $test$.

The treatment of **case** expressions and other conditional expressions is a generalization of the treatment of **if** expressions. Generalized **BRANCH** and **MERGE** nodes, with an arbitrary number of outgoing or incoming arcs, serve as interface nodes. Figure 5.7 shows the general structure of the demand graph for a conditional expression. There is a **BRANCH** node for each variable that is defined within the expression and there is a **MERGE** node for each variable for which any of the branches of the expression contains an exposed use.

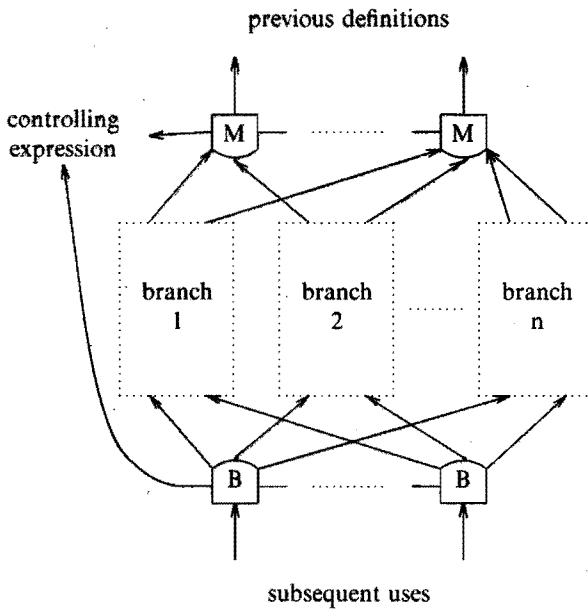


Figure 5.7. General conditional expression.

Each demand path enters a conditional expression through a **BRANCH** node pointing to the controlling expression and to the exposed definitions in one or more branches. For each variable for which a branch contains an exposed use a **MERGE** node is created. Some paths may lead directly from a **BRANCH** node to a **MERGE** node just as in the previous figure. Note that not all input ports of a **MERGE** node need to have an incoming arc.

Loops.

A loop expression is treated almost exactly like conditional expressions. The *test* and the *body* are the two branches for which isolated demand graphs are constructed. The interface nodes are called **ENTRY-LOOP** and **EXIT-LOOP** nodes. Their connections to the subgraphs are such that cycles may be created. Figure 5.8 shows such a cyclic data dependency. The demand path for x may bypass the \otimes node or include it an arbitrary number of times, just as the body of the loop may be executed an arbitrary number of times. As we will see later, cyclic graphs require special mechanisms during demand propagation. Interpreting a cyclic graph with a naive machine model in mind may lead to similar problems.

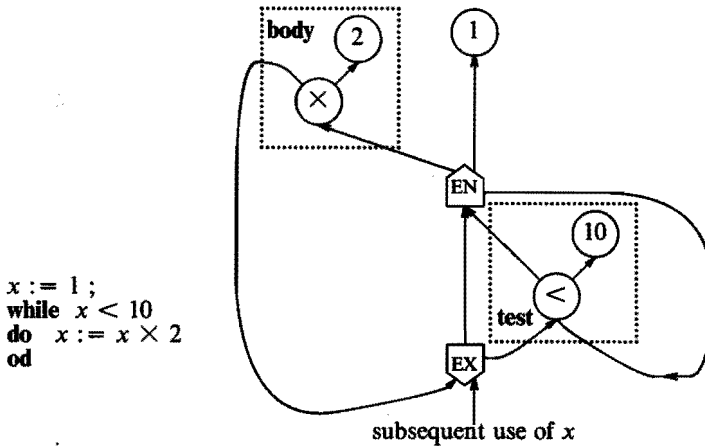


Figure 5.8. The demand graph for a loop.

The subgraph for the body appears on top and that for the test at the bottom. After these branches have been analyzed in separate chainers the cocoon is dissolved, which involves the creation of the interface nodes for the cyclic demand path for x . The arc entering the EXIT-LOOP node (marked with EX) from below corresponds to the value of x after termination of the loop. The EXIT-LOOP node corresponds to a value of x immediately after the test and the ENTRY-LOOP node (marked with EN) corresponds to a value immediately before the test. At the latter point the value of x is either the value before the loop, if the body is not executed at all, or the value defined in the body. The outcome of the test determines which of the two values is taken. The value used in the body is again the value immediately after the test, i.e. the value represented by the EXIT-LOOP node.

Of course, if the value of the constants appearing in this example are taken into account, it can easily be deduced that the body is executed exactly four times. This illustrates the particular point of separation between the general and the application specific part that has been chosen for the current implementation. Through simple constant propagation this whole graph could have been replaced by a $\textcircled{16}$ node. Constant folding, however, is considered to be an application and values of constants have no bearing on the demand graph. All analysis that uses the value of constants is reserved for the application specific part.

Procedure Calls.

The effect of a procedure call on the demand graph depends exclusively on the exposed uses and definitions of the procedure body. Some of these are immediately visible: the input parameters and the return values. When the language contains global variables, or an equivalent mechanism, the complete set of exposed uses and definitions of a procedure body can only be determined through use-definition analysis. Therefore, whenever a call is encountered of a procedure that has not yet been analyzed, the analysis of the calling procedure is suspended and the called procedure is analyzed to determine its exposed uses and definitions. Output interface nodes are called RESULT nodes (see figure 5.9). For each RESULT node a local output interface node is created and the two nodes are connected. Input interface nodes are called PARAMETER nodes, which are connected to local input interface nodes. After the creation of the interface nodes the analysis of the calling procedure is resumed by connecting the local input interface nodes to the appropriate definitions and informing the chainer about the local output interface nodes.

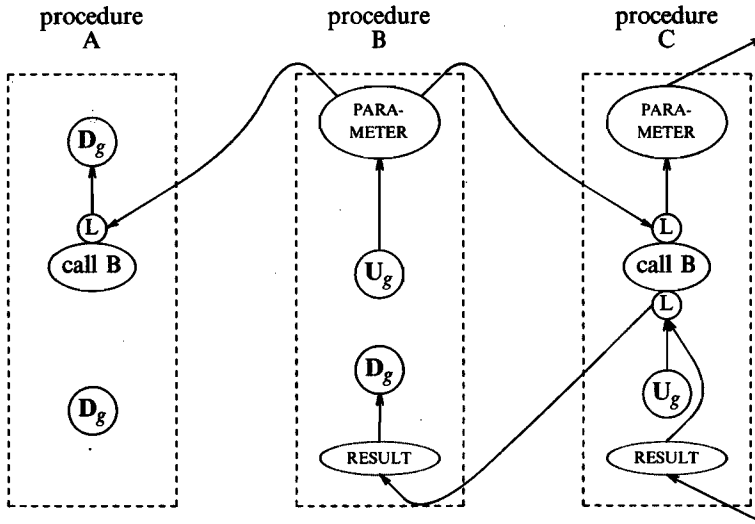


Figure 5.9. Interprocedural demand paths for global variables.

In this illustration D_g and U_g indicate a definition and a use of one and the same global variable g . Procedure B is called by procedure A and C . Since procedure B contains both an exposed use and a definition of global variable g , its cocoon creates a PARAMETER and a RESULT node. These are connected to local interface nodes (indicated by L) at each calling site, which act as local uses and definitions of the same variable. Since procedure C contains no definition of g the local interface nodes for the call of B are in turn exposed use and definition, resulting in a PARAMETER and a RESULT node.

Each procedure is analyzed at most once: when a call is encountered of a procedure that has already been analyzed, only the local interface nodes have to be created and connected to the PARAMETER and RESULT nodes. In the end each PARAMETER node has arcs leading to all calling sites. When the program is recursive, at some point a call is encountered of a partially analyzed procedure. Repeating the partial analysis of the called procedure is of no use, since no new information would be obtained, but continuing the analysis of the called procedure requires information about the use and definitions of global variables. In this case the information about global variables, collected in the previous phase (see section 5.2.1), is used to create the interface nodes when they are needed. This information is a (safe) approximation to the exposed uses and definitions of the called procedure. The analysis of the calling procedure is resumed and eventually the analysis of the called procedure will be completed at which point its demand graph is connected to the already created PARAMETER and RESULT nodes. The information that is used is approximate in the sense that it considers each occurrence of a global variable as an exposed use. This may lead to unnecessary PARAMETER nodes, but, fortunately, they do not become part of the demand graph, since they are not reachable from the source-of-demands.

Escapes.

An escape from an expression is a forward jump to the point just beyond the expression being escaped from. Many languages have a mechanism to escape from a loop body or a return expression to escape from the current procedure. SUMMER has an escape mechanism with an even broader scope. Since expressions immediately following an unconditional escape are never executed, an escape should always be

embedded in a conditional. The expressions following the conditional expression are executed whenever the escape is not executed. The effect of an escape can therefore be taken into account during analysis by adding complementary conditionals (see figure 5.10). The same effect can be obtained by the creation of conditional cocoons. The controlling expressions of these cocoons, or rather their demand graphs, have to be created. By means of auxiliary names, which we call *pseudo-variables*, the existing cocoon mechanism creates exactly the right subgraphs.

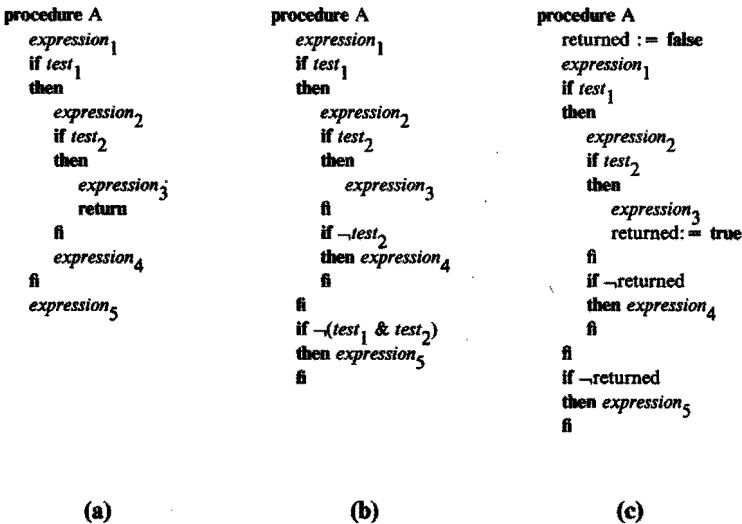


Figure 5.10. Effect of escapes.

(a) Procedure with **return**. Procedure A contains a **return** expression. *expression*₄ and *expression*₅ may not be executed depending on the outcome of *test*₁ and *test*₂. The tests are assumed to be free of side-effects.

(b) The equivalent program without **return**. The controlling expression of the additional expressions can become arbitrarily complex.

(c) Pseudo variables. Using an auxiliary variable simplifies the additional expressions. Obviously, the name of the auxiliary variable should not be in the name space of normal variables.

Aliasing and Indirection.

The combination of aliasing with other features in the language may lead to considerable complications, which have been only partly explored. In the current implementation the aliasing due to first order pointers (no pointers to pointers or to structures containing pointers) in combination with conditionals and loops have been investigated. The complications due to higher order pointers (including cyclic data structures) and interprocedural aliasing have not been studied.

In straight-line code the presence of first order pointers is quite easily dealt with. Indirection is incorporated in the use-definition chaining by admitting objects, in addition to variables, as keys for the chainers. Recall that when a (direct) assignment is encountered the chainer is informed that the *variable* now lives at the last encountered value node. When an indirect assignment is encountered the chainer is informed that the *object pointed to by the variable* now lives at this value node.

Both loops and conditional expressions introduce the possibility of *conditional aliasing*. Conditional assignment to a pointer variable is a serious complication, since the ambiguity introduced by the conditional may be propagated through the analysis indefinitely, necessitating the introduction of ambiguity nodes, whenever the pointer is referenced (see figure 5.11). This would seriously increase the complexity of the graph and consequently the computational complexity of the method. However, an algorithm has been developed that detects and exploits locality properties in the reference patterns to reduce the number of additional nodes due to conditional aliasing. A detailed explanation of this algorithm will appear in section 6.7.

```

a := if test then reference to x else reference to y fi
...
definition of x
indirect definition through a
use of x

```

Figure 5.11. Conditional aliasing.

At the point where x is used it is ambiguous where the variable lives: at the previous definition of x or at the indirect definition through a . The effect of the conditional expression may be propagated to all uses of a and x .

5.2.3. DEMAND PROPAGATION

When the complete demand graph has been constructed, a graph representation of the program is available that lacks most of the irrelevant sequencing specifications of the original program. An analysis in this graph follows the same basic outline as other flow analysis methods described in the previous chapter: propagation rules and initial assertions are associated with the graph and information is propagated by application of the propagation rules. Again, if the assertion lattice is bounded and the propagation rules are order preserving a steady state will be reached. Both the type of assertions and the method of propagation, however, are different.

If an expression directly depends on data computed in another expression, the nodes in the graph corresponding to the two expressions tend to be close together. In those applications in which the assertions are concerned with data, the locality properties of the demand graph can be exploited to reduce the amount of information that has to be retained by each node. Each arc in the demand graph can limit its local information to assertions about the data item it represents. Global assertions, as described in the previous chapter, can be avoided and it becomes feasible to retain much finer information per variable. In the Value Approximation application, for instance, the information to be transmitted over an arc only concerns the data item that it represents (see figure 5.12). For most nodes the assertions associated with incoming arcs are identical. To simplify the description we will assume that for such a node the assertions associated with the incoming arcs are replaced by one assertion associated with the node itself.

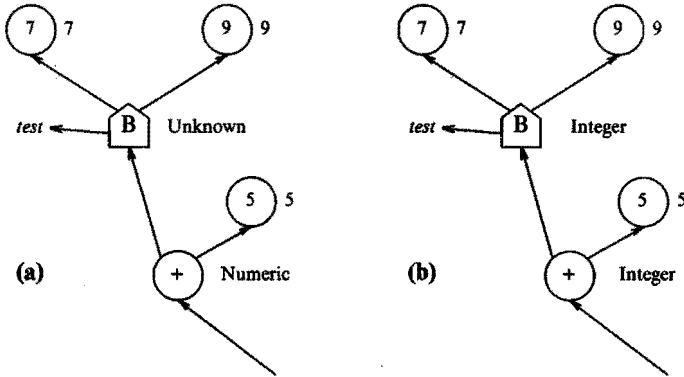


Figure 5.12. Demand propagation in the Value Approximation application.

(a) The demand graph of figure 5.5 with the value domains of initial assertions indicated on the right of each node. At the CONSTANT nodes (7 and 9) the values are known exactly, at the ⊕ node it is known that the value is *Numeric* (*Integer* or *Real*). Note that the assertions only characterize one value, in contrast with the global assertions of figure 4.2.

(b) The final assertions after demand propagation. The value domain at the BRANCH node is *Integer*, which is the meet of 7 and 9 in the semi-lattice of figure 4.1. The final assertion associated with the ⊕ node results from the propagation rule associated with the node.

Information propagation involves the exchange of information between the incoming and outgoing arcs of a node until the total information stabilizes. Just as in the methods described in the previous chapter, there are many ways in which this propagation can be organized. The most simple method, arbitrary order iteration, cannot be used, since the arcs in the demand graph are uni-directional and a node can only initiate a communication with its successors. The information propagation therefore has to start at the source-of-demands. A recursive descent algorithm in a spanning tree of the graph can often be used. The analysis starts by requesting the source-of-demands to deliver the required information about the complete program. Each node reacts to such a request by transmitting requests for information to its successors. Local information may accompany requests, which are called *demands*. In an acyclic graph each chain of demands will eventually reach the sink-of-demands, unless it encounters a node that is able to reply immediately, because it has already consulted its successors in response to a previous demand. The information acquired from the successors is combined with the local information into a reply. All information that passes through a node is accumulated: the locally stored information never decreases.

Demand chains grow in a direction opposite to the control flow. For backward type applications the demand is accompanied by backward flowing information and for forward type applications the forward flowing information is contained in the reply. Some applications have both a forward and a backward component and the two information flows interact. A full scan iteration could be used to accommodate this: the full recursive descent is repeated until the information stabilizes. However, when more than one iteration is needed it is usually due to a few isolated parts of the program where the two information flows interact locally. In the demand graph this locality can be easily exploited by giving priority to local information propagation.

In cyclic graphs special precautions need to be taken, the nature of which depends on the application. Often the recursive descent of a spanning tree of the graph does not produce sufficiently strong assertions. Demand propagation may then be organized in such a way that the demand graph acts as a network of independent objects, which exchange messages with their direct neighbors. Analysis also starts with an initial demand to the source-of-demands and each node may react by sending demands to its successors. The main difference with the recursive descent method is that the order in which outstanding demands are processed is not determined and that the receipt of a demand and the construction of a reply are separate events. A reply can be postponed until all backward flowing information has been received. This postponement may lead to deadlock, so when all activity ceases because all replies are postponed, a central mechanism selects a node to break the deadlock by sending a partial reply. The effect is that subgraphs without complications are processed first and that the complications due to cycles in the remainder of the graph can be resolved more easily with the help of the already collected information.

5.2.4. EXTRACTION

In most applications the information eventually returned by the source-of-demands at the end of the demand propagation, is not the total or even the main result of the analysis: the information stored in the nodes of the graph as a side-effect of the demand propagation is far more important. The final phase of the analysis therefore consists of extracting the relevant information from selected nodes in the graph and transforming this into the form required by the application. Nodes in the syntax tree that are left out of the demand graph may also be used in this process. For conventional code generation, for instance, it is convenient to recursively descend the complete syntax tree, visiting the nodes in the normal evaluation order, and using the information available in the demand graph to optimize the code.

References

- AhUI77. AHO, A.V. AND J.D. ULLMAN (1977). *Principles of Compiler Design*, Addison-Wesley, Reading, Mass..
- Ganz74. GANZINGER, H. (Nov 1974). *Modifizierte Attributierte Grammatiken*, Bericht 7420, Technische Universität München.

Chapter 6

Demand Graph Construction

Constructing the demand graph is the key phase of the demand graph method: it converts a program representation in which control flow is predominant into a representation in which data flow is explicit. Because this transformation requires an interpretation of control flow operators, a detailed description cannot be as language independent as the broad outline in the previous chapter. The first section of this chapter therefore describes the features of **SUMMER** that are referred to subsequently. The rest of the chapter explains the process in more detail than the previous chapter did, but a complete treatment of the algorithm would require a level of detail that would render the description nearly incomprehensible. The algorithm is full of cross-connections due to interaction between features of the input language. A compromise has therefore been struck between clarity and completeness. Readers who would like more detail are referred to appendix II.

After a preliminary discussion of mechanisms used for the translation of all language features, section 6.3 describes demand graph construction for a basic subset of the language. Section 6.4 deals with conditional control flow, which covers a great part of the language, since in **SUMMER** the handling of control flow is distributed over many operators. Interprocedural analysis is covered in the next section, whereas the concluding sections are devoted to arrays and aliasing. The reader who would like to read about the algorithm that handles conditional aliasing without digesting the whole demand graph construction process can skip most of this chapter, skim section 6.6, and then proceed to section 6.7.

6.1. The **SUMMER** Programming Language

SUMMER is both the input and the implementation language of the analyzer. It has been designed and implemented in the late seventies by Paul Klint and Marleen Sint at the Mathematical Centre. It is a well-structured and clearly defined language originally intended for string processing. It includes string handling and pattern matching facilities similar to those in other string processing languages such as **SNOBOL**. An abstract data type mechanism is available to hide the internal representation of data

structures. In the current implementation programs are, for a large part, interpreted, which allows for a friendly interface to the programmer. Unfortunately, the low speed of this interpreter makes it impractical to use SUMMER for problems that require a significant amount of computation. This lack of efficiency is one of the reasons why the use of this language has not spread far beyond where it was conceived. It should thus be considered a research language.

Since some of the original reasons for choosing SUMMER as the input language for the analyzer (see [Veen80]) have lost their relevance, in hindsight I somewhat regret this choice. Nevertheless, the main value of the project is in the development of methods that are useful for the analysis of a variety of languages, including popular well-structured imperative languages. Choosing a research language has the advantage that one does not commit oneself to a choice among the popular languages, provided that the research language is rich enough to be a fair representative of the whole class. SUMMER qualifies in this respect: it includes many of the features that make flow analysis for imperative languages troublesome (and interesting). The following description of the language is not meant to be complete but rather to cover enough of the language to make this and the following chapter intelligible. Readers who would like more details are referred to [Klin80] and [Klin82].

EXPRESSION ORIENTATION

SUMMER has an expression oriented syntax, in the sense that almost every syntactic construct can be viewed as an operator that glues expressions together forming a new expression. This is not only true for ordinary operators like '+' or constructs like *if...then...else...fi*, but also for the ';' and ':=' operators. Almost any expression can be used as an operand in a larger expression. A typical example is

```
x := 2 * case window of
      Open:   val
      Close:  val + 1
      Unknown: 0
    esac
```

Some operators construct a new value (such as the numeric addition '+', or the string concatenation '|'), whereas others yield the value of their right operand (such as ';', ':=', or relational operators). This combined with proper priority rules facilitates concise expressions like:

```
x := y := 5 ;
if a < x < b then ...
```

A complicating factor is that in certain contexts expressions yield "addresses" instead of values. For example in

```
if test then x else y fi := val
```

the *if* expression computes a target for the assignment.

Since ':=' is just another operator, sub-expressions may include assignments. In the following example *ind* is incremented before it is used as a subscript

```
ar[ind := ind + 1] := nextval;
```

The incrementing of the value of *ind* during the evaluation of the subscript is called a *side-effect*. Side-effects may be hidden due, for instance, to the call of a procedure that updates a global variable. This flexibility towards side-effects is a useful feature that, in

some cases, allows more concise and clear programming. It makes, however, SUMMER imperative to the extreme and in fact the reliance on side-effects, by some considered to be detrimental to clear programming, is pervasive in SUMMER programs. Since the language is designed to be as orthogonal as possible, nothing prevents the programmer to abuse this feature to produce horrifying constructs like

```
(x := 53 ; y) := val — if (y := y + 1) < 0
                        then k
                        else x > y
                        fi
```

Because of these side-effects, the order in which expressions are evaluated has to be strictly defined to prevent ambiguity. Evaluation order is left-most inside-out except for an assignment where the right-hand side is evaluated before the left-hand side.

Iteration may be specified by a **while** or a convenient **for** expression. Of course, the controlling expressions may contain side-effects.

FAIL MECHANISM

Although SUMMER does not provide arbitrary jumps (*goto*'s) the evaluation of an expression can be aborted by an escape mechanism, precluding the side-effects of the unevaluated sub-expressions. A familiar escape mechanism is the **return** expression: when such an expression is encountered during execution, the evaluation of the current procedure body is aborted and the evaluation of the calling expression is resumed. In addition to this, SUMMER provides a similar but more powerful escape through its *fail* mechanism. Conditional constructs and loops are not controlled by boolean values but by fail signals. In fact each expression may yield a fail signal instead of a value. If an expression fails (i.e. its evaluation yields a fail signal) the evaluation of the surrounding expression is aborted and the fail signal is transmitted until it is caught by a surrounding construct, such as an **if** or **while** expression. Since this fail signal is also transmitted across procedure calls, one failing operand may cause the abortion of a series of nested procedure invocations.

An expression can only fail if

- It is a relational operator.
- It is a call of a built-in procedure (e.g. the opening of a file may fail).
- It is a call of a procedure that may fail or that contains an **freturn** expression.
- Any of its sub-expressions can fail unless that sub-expression is
 - the *test* branch of an **if** or **while** expression
 - the left operand of the '|' (boolean OR) operator

The ';' operator is special in that failure of its left operand is not transmitted, but results in a run-time error. In all other respects the ';' operator is equivalent to the '&' (boolean AND) operator. The right operand of the '|' operator is evaluated only if its left operand fails. One consequence is that the expression

A & B | C

is equivalent to

if A then B else C fi

provided that B cannot fail. Arbitrary control flow can thus be constructed without using explicit **if** constructs. The flow analysis has to take these escapes into account, which may be hidden in any sub-expression.

Defensive programming is facilitated by the `assert` expression: if its operand fails, an error message is issued and the execution of the program is aborted. It is a natural means for specifying the conditions that should hold before or after an expression.

DATA TYPES

In SUMMER data items are called *objects*. They have a *value* and a *type*. A type can be simple (**real**, **integer**, **string**) or structured (**array**, **table**, or a user defined type). Arrays are one-dimensional sequences of objects which are indexed by their sequence number. They may contain elements of arbitrary and mixed type; the equivalent of a multi-dimensional array can thus be easily constructed. The range of an array may be extended. A table is a generalization of an array: it can be indexed with objects of arbitrary type rather than only integers. It can thus serve as an associative memory.

The data abstraction mechanism is used extensively in the implementation of the analyzer. A class declaration defines a new data type. Each object of such a type contains a fixed number of data fields, which can be selected by means of the *dot* notation. Classes may be “active”: the class declaration may include local procedures that are associated with *procedural* fields. An access to such a field¹ (which from the outside is indistinguishable from a data field) triggers the execution of the associated procedure. Within such a class procedure the object itself is referred to with the keyword *self*.

The subclass mechanism provides a means to share properties between classes. A class that is declared to be a subclass of a previously declared class (called its *superclass*) inherits all the properties of the superclass unless explicitly redeclared. In this way a (tree-like) hierarchy of types can be defined, descending from general to more specific.

ALIASING

Variables have to be declared, but may possess values of different types during their life-time. Each variable is a pointer to an object. The expression ‘*a := 'ape'*’ has the effect that *a* is made to point to a new object of type *string* and value ‘*ape'*’. The effect of a subsequent assignment ‘*b := a*’ is that *b* is made to point to the same object and *a* and *b* are different names (or access paths) to one shared object. They are called *aliases*. For simple data types this aliasing has no consequences, since there are no operations that can change such an object (there is, for example, no operator like *replace-first-character* for a string). Structured objects, however, can be changed and any such change realized along one access path is visible along all others. Since aliasing occurs so frequently the analyzer should handle it efficiently.

OVERLOADING

Since different classes may use the same names for their fields, a particular field selection may refer to any of a series of procedures depending on the type of the object from which the field is selected. If, for instance, the programmer had declared a class SET with the ‘+’ operator² associated with a procedure that computes the union of two sets, the ‘+’ in ‘*a + b*’ may refer to integer addition, real addition, or set union depending on the types of *a* and *b*. We say that the ‘+’ symbol is *overloaded*: one

1. We will refer to a procedural field *p* of class *C* as “field *p* of *C*”, or “procedure *p*” or simply as “*p* of *C*”.
2. Procedures may also be specified as monadic or dyadic operators.

name may refer to different operations depending on the context. At run time the type of the left operand is used to resolve this ambiguity. To resolve this statically the type of the operand would also have to be known statically.

6.2. Overall Structure

The class mechanism in SUMMER proved to be convenient for the implementation of the demand graph method. The information contained in a node and the way it is manipulated is determined by the type of the node. The graph can be viewed as a message exchanging network in which each node may exchange information with each neighbor. Although neighbors may be of arbitrary types, the communication protocol should be standard. These requirements map easily onto the class mechanism. Each type of node has a corresponding class declaration, which specifies procedural fields (for type specific operations) and data fields (for type specific information and pointers to neighbors). A node can send information to a neighbor by calling one of its message receiving fields transmitting information through the parameters. Overloading is essential for the standard protocol, since one name is used for all the procedural fields that implement message receiving.

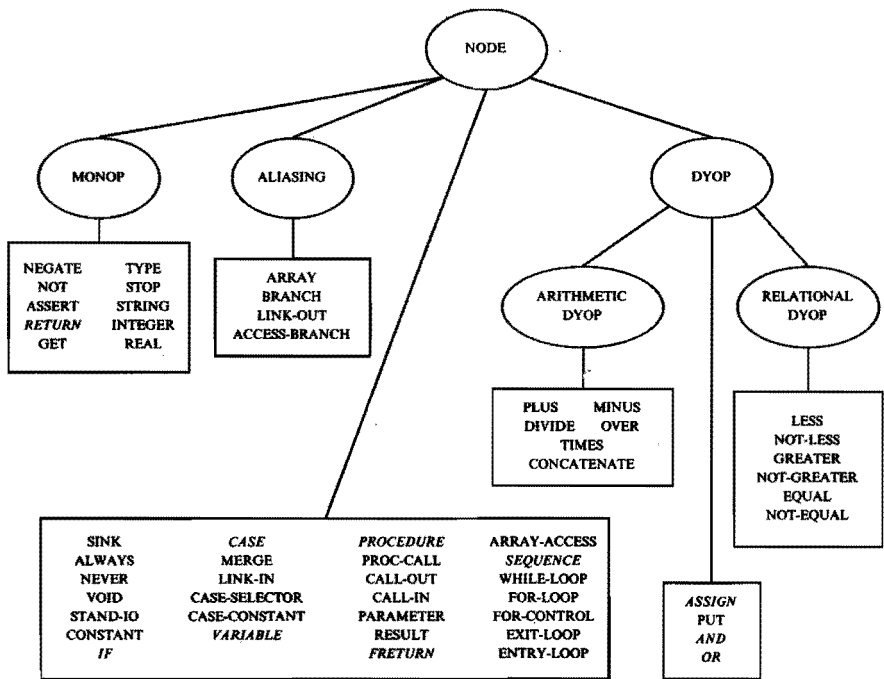


Figure 6.1. Type tree for nodes in the demand graph and syntax tree (simplified).

The connections in this tree indicate subclass relations. NODE is the superclass of all nodes. All leaves (basic types) that are children of the same superclass are depicted together in one box. Nodes that are part of the syntax tree but not of the demand graph are in *italics*.

THE TYPE TREE

The **subclass** mechanism can be used to express the properties that the different nodes have in common. Figure 6.1 depicts the type system used to implement the demand graph. The **subclass** relation between the types induces a tree with the most general type (NODE) at the root and the most specific ones (e.g. PLUS) as leaves. A type system formed as a graph would have been somewhat more convenient, but SUMMER does not allow this.

CONSTRUCTION OF THE SYNTAX TREES

As already mentioned in the previous chapter, the syntactic analysis is performed by the parser of the existing SUMMER compiler. It checks the input program for syntactic correctness and converts it into a forest of syntax trees, one for each procedure. Appendix I gives the syntax of the subset of SUMMER that is accepted by the analyzer as currently implemented and specifies the syntax trees produced by the parser.

During the construction of the syntax trees global summaries of the effect of each procedure are compiled. This information is used during demand graph construction to break recursive cycles in the procedure call graph. Each summary consists of two lists, one for the uses and one for the definitions of global variables. The lists summarize both the effects of the procedure body itself and of any procedure called directly or indirectly. Uses and definitions cannot be distinguished without contextual information, as the following contrived, but legal, expression illustrates:

$$\text{if } U1 = 0 \text{ then } D1 \text{ else } (D2 := U2 ; D3) \text{ fi } := (D4 := U3 ; U4)$$

Each Ux in this example is a use and each Dx a definition. Since the distinction between left-hand side and right-hand side of an assignment is not sufficient in this respect, we speak of *address* context and *value* context. A variable is a definition when it occurs in an address context, and a use when it occurs in a value context. An operand that is expected to deliver neither a value nor an address is in a *void* context.

When the syntax tree of a procedure has been constructed, it is completely traversed to determine the context of each of its nodes by recursively calling the *find-context* procedure of each node. When a use of a global variable is encountered it is recorded in the list of global uses. Since a definition of a variable may induce a use (see figure 5.6), a global variable that gets defined is entered in both global lists. The *find-context* procedures also determine whether a procedure may fail and, if so, record this in the list of global definitions.

The analysis algorithm will be presented in a notation that resembles SUMMER but is somewhat more readable: indentation is used for structuring so that closing keywords and many semicolons can be omitted. Details of algorithms are often replaced by imprecise specification in words. User defined types are in SMALL CAPITALS. All variables are either parameters or belong to the class instance. The *find-context* procedure of global variables illustrate this:

```

find-context(in-context) of GLOBAL-VARIABLE
  add to global uses
  if in void context
    warning 'superfluous variable'
  if in address context
    is-a-use := False
    add to global definitions
  if in value context
    is-a-use := True

```

At the end of this phase, when all syntax trees have been constructed, the transitive closures of the global uses and definitions are computed. This is implemented by a recursive descent algorithm in the call graph similar to the one described in [Tarj72].

ATTACH PROCEDURES

The conversion from syntax tree to demand graph is achieved during a recursive descent of the tree starting at the root of the tree of the main program. The algorithm is best understood if each node is considered to be an active object that can locally alter the graph by adding new arcs and nodes. This process, by which a node of the syntax tree takes its proper place in the demand graph, is called *attaching* the node to the demand graph. It is implemented by a collection of *attach* procedures. All nodes of the syntax tree have an *attach* procedure, including those that will not become part of the demand graph. These *attach* procedures together with the chainer and cocoon mechanism implement the construction of the demand graph. The construction is started by attaching¹ the main program node and proceeds by recursively attaching all of its descendants in an order corresponding to the normal evaluation order.

In the rest of this chapter the *attach* procedures of several types of nodes are described. The next section is limited to the implementation of the basic mechanism, whereas in subsequent sections the implementations are discussed capable of treating complicating features such as control flow, arrays, and aliasing. Because of this incremental presentation some procedures are described more than once. Simplified versions, for which a final version will be presented later, are marked as such. All final versions can be found in appendix II.

6.3. Naive Demand Graph Construction

In this section the basic mechanism for use-definition analysis is explained, ignoring the complications due to procedure calls, conditional expressions, iterations, data structures, and escape mechanisms. The cocoon mechanism is not used for this naive implementation: we assume that the demand graph is constructed in the context of one single chainer. This chainer, responsible for the use-definition administration, contains a table *deflist*, which associates each variable name with its most recently encountered defining node. Each chainer provides two basic fields, which in this section are considered to be implemented as follows:

1. In the rest of this chapter we use "attach" as a synonym for "calling the *attach* procedure of."

```
use(name) of CHAINER (Simplified)
  return deflist[name]
```

```
def(name,node) of CHAINER (Simplified)
  deflist[name] := node
```

Although originally intended for the administration of variables, the same mechanism turned out to be useful for a few other problems. The range of *name* is therefore extended beyond variable names to include a number of *pseudo-names*, some of which we shall encounter shortly.

ASSIGNMENTS, VARIABLES, AND CONSTANTS

Data-dependency analysis for simple expressions without conditional control flow is straightforward. For instance, during the attachment of the expression

```
( y := 5 ;
  x := y )
```

the first ASSIGN node informs the chainer that *y* now “lives” on the CONSTANT node ⑤ by calling

```
def ( 'y', ⑤ )
```

and the second ASSIGN node informs the chainer about the new definition of *x*

```
def( 'x', use( 'y' ) )
```

However, because both sides of an assignment can be arbitrarily complex, the calls on *use* and *def* have to be issued separately by the nodes on both sides. Information has to be transmitted between the two sides and, for reasons that will become clear in the next section, the chainer is used as an intermediary, storing the information under the pseudo-name *Value*. In a first approximation the following scheme will do

```
attach of SEQUENCE
  attach all children in order
```

```
attach of CONSTANT (Simplified)
  def(Value, self)
```

```
attach of ASSIGN (Simplified)
  attach right-hand side
  attach left-hand side
```

```
attach of VARIABLE (Simplified)
  if is-a-use
    def(Value,use(name))
  else
    def(name,use(Value))
```

```
attach of ARITHMETIC-DYOP (Simplified)
  attach left operand
  attach right operand
  def(Value, self)
```

The scheme as described so far cannot handle expressions in which either side of an assignment contains other assignments. Consider, for instance, the expression

$$(x := 5 ; a) := (y := 6 ; y + 1)$$

which assigns 6 to y , 5 to x , and 7 to a in this order. The problem is that during the attachment of the central ASSIGN node other ASSIGN nodes have to be attached. An extra pseudo-name *Address* is therefore introduced, which points to the target of the currently processed ASSIGN node. When the analysis of an ASSIGN node switches from value context to address context, the information in the chainer is transferred from one pseudo-name to the other. By saving the *Address* pointer locally in each ASSIGN node a stack of *Address* pointers is maintained, when descending a complicated tree.

Figure 6.2 depicts the processing of the above expression. Note that all CONSTANT nodes make a connection with the *sink-of-demands*, the common descendant of all nodes in the demand graph. This node is created at the start of the demand graph construction and stored in the chainer under the pseudo-name *Sink*.

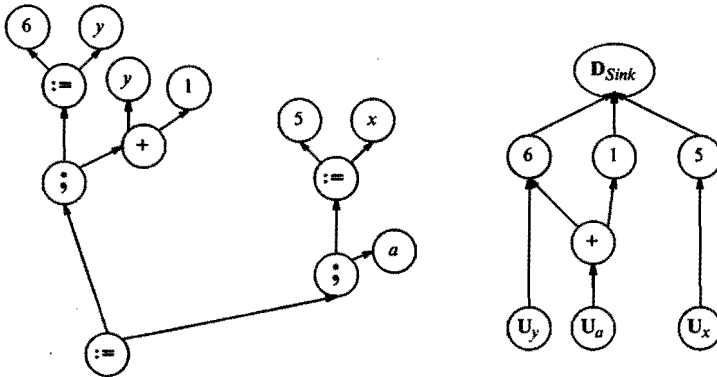


Figure 6.2. Attaching the expression $(x := 5 ; a) := (y := 6 ; y + 1)$.

The syntax tree appears on the left and the resulting demand graph on the right. In this and subsequent illustrations D_x and U_x stand for the previous definition and the subsequent use of variable x .

INPUT AND OUTPUT

A special role is reserved for the output expression *put*. Each part of a program that does not contribute in any way to its output is, in a sense, superfluous. Nodes that are not reachable from any PUT node are in the same sense superfluous. This is easily detected if demand propagation can start at a node that is guaranteed to be an ancestor of all PUT nodes. The demand graph constructor provides this node, known as the *source-of-demands*, by creating a rooted *IO-graph* that links PUT nodes together reflecting their order of execution (see figure 6.3). This graph is constructed with the aid of the pseudo-name *Standard-IO*, which represents the output stream. Output expressions can be viewed as the concatenation of a new string to the output stream produced so far:

$$\text{put}(a) \longrightarrow \text{Standard-IO} := \text{Standard-IO} \parallel a$$

where *Standard-IO* refers to the output stream and \parallel indicates string concatenation. A PUT node is therefore implemented as if it were a dyadic operator with a reference to the pseudo-name *Standard-IO* as its left operand. This arc will point to the previous

PUT node or, as we shall see in the next few sections, to an interface node of an expression that contains such a node.

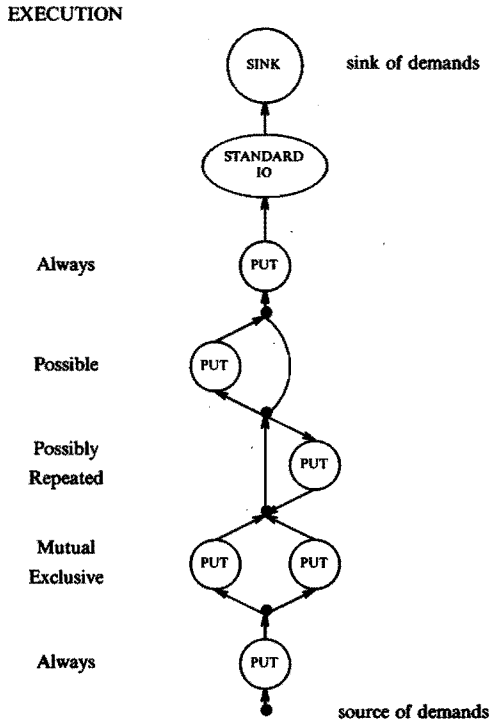


Figure 6.3. Ordering of PUT nodes in a graph.

The graph depicts the reverse of the partial ordering of execution of output expressions. The IO-graph ends in a dedicated node of type STANDARD-IO.

When not only the order of output actions is to be preserved, but interaction (input and output on the same medium) as a whole, the GET nodes (which represent input) have to be part of the IO-graph. For this reason a GET node is treated as if it were a monadic operator with two incoming arcs

$$x := \text{get} \quad \longrightarrow \quad [x, \text{Standard-IO}] := \text{get}(\text{Standard-IO})$$

which indicates that *get* has the current IO-stream as argument and delivers two values: the next input string and the new IO-stream.

6.4. Conditional Control Flow

As we have seen in section 6.1, conditional control flow is guided not by the evaluation of a boolean expression, but by the generation of a fail signal. The handling of such signals can by itself lead to hidden control flow jumps. Control flow and the handling of fail signals are therefore intricately interwoven. For the sake of clarity we start with treating conditional expressions as if they were controlled by simple boolean expressions.

BRANCH, MERGE AND LINK NODES

When conditional control flow is involved, the use-definition chaining becomes less straightforward than suggested in the previous section. Figure 6.4 shows the embedding of an if expression in its surrounding graph. The figure corresponds to an expression like

```

if condition
then a := a + 1 ; b := a + b
else a := a + 5
fi
    
```

Exposed uses are connected to previous definitions via MERGE nodes. Subsequent uses are linked to exposed definitions via BRANCH nodes. A BRANCH node always has an *outlink* arc for each branch. If the particular branch does not contain a definition for that variable, the arc will lead to the MERGE node. This is what we have called an induced use in the previous chapter.

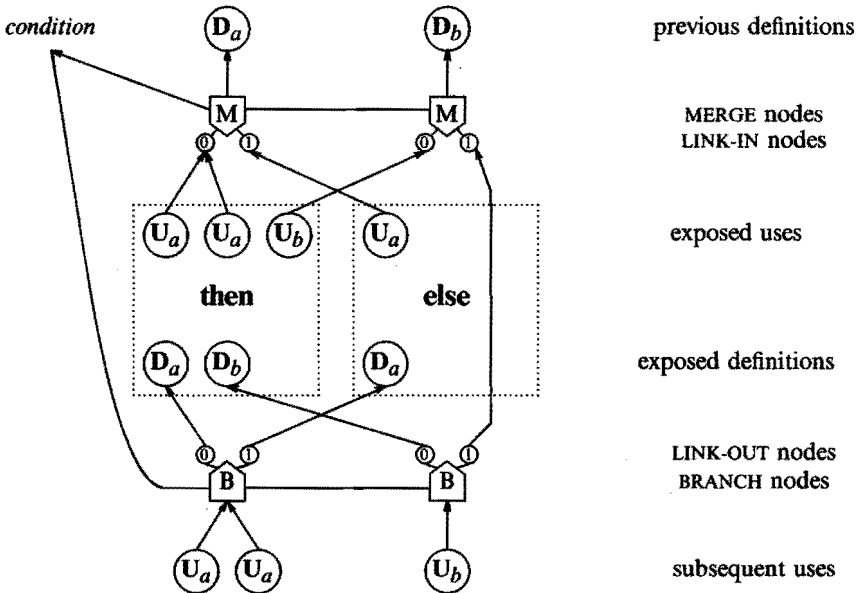


Figure 6.4. BRANCH, MERGE, and LINK nodes.

The branches of an if expression are completely surrounded by BRANCH and MERGE nodes. The LINK nodes are tagged with an integer *position* that corresponds to the value of the fail signal: 0 stands for success and 1 for failure. LINK-IN nodes are created when the first exposed use is encountered, while the other interface nodes are created when the cocoon is dissolved.

MERGE nodes are not strictly necessary, but surrounding each special expression by interface nodes facilitates demand propagation for many applications. This makes it easy to detect which nodes are part of the same expression. MERGE nodes may have one or two incoming arcs each passing through a LINK-IN node which can tag the demand signal with a *position* indication. In this way the MERGE node can distinguish demands coming from different directions. BRANCH nodes are provided with LINK-OUT nodes. Although included originally for reasons of symmetry they turned out to be

convenient for the implementation of the alias algorithm to be described in section 6.7. Many applications simply ignore LINK-IN, LINK-OUT, and MERGE nodes. LINK nodes and sometimes even MERGE nodes will be omitted from most illustrations.

CONDITIONAL COCOONS

BRANCH nodes are created by means of the cocoon mechanism. During the attachment of an IF node a CONDITIONAL-COCOON with two chainers is created and the two branches are attached each in its own chainer. A stack of chainers is maintained; its top is the *current chainer* to which calls of *use* and *def* are directed. The controlling expression, which is attached outside the cocoon, stores the controlling node in the chainer under the pseudo-name *Success*, as we will see later.

```
attach of IF
  attach condition
  create CONDITIONAL-COCOON
  link control of cocoon to use(Success)
  attach then-branch within then-chainer
  attach else-branch within else-chainer
  dissolve cocoon
```

Chainers have to be somewhat more elaborate than the ones described in the previous section, since they now have to handle exposed uses. These can easily be detected, since the first time a use is encountered for a variable for which the current chainer has not yet recorded a definition, the *deflist* is empty for that variable. The cocoon to which the chainer is connected is asked to create an appropriate *entry* node; a CONDITIONAL-COCOON will create a LINK-IN node. This entry node is recorded in a separate table *uselist* and all exposed uses of that variable are made to point to this entry node.

```
use(name) of CHAINER                                     (Simplified)
  if name in deflist
    return deflist[name]
  else if name not in uselist
    uselist[name] := cocoon.entry-node(position)
  return uselist[name]
```

The *deflist* always contains the last definition of a variable, so when a branch has been completely analyzed it contains all its exposed definitions. The *uselist* then contains the entry nodes to which all exposed uses have been connected. When both branches have been analyzed, these lists are sufficient to create the appropriate interface nodes.

Dissolving a CONDITIONAL-COCOON starts with the creation of the interface nodes. Since the creation of BRANCH nodes may induce new exposed uses, this has to occur before the creation of MERGE nodes. The interface nodes are then connected to the surrounding expressions by first popping the chainer stack, issuing a call of *use* for each MERGE node, and then a call of *def* for each BRANCH node.

CASE EXPRESSIONS

A case expression is treated very similar to an if expression, since the latter can be considered to be a case expression with only one non-default branch. BRANCH and MERGE nodes can in fact have an arbitrary number of outlinks or inlinks. If these nodes are used to interface a case expression they are connected to a CASE-SELECTOR node, which represents the comparison that determines during execution which branch is to be taken.

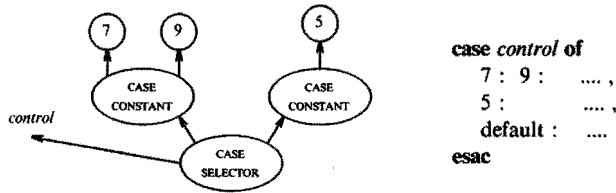


Figure 6.5. The control part of the demand graph for a case expression.

BRANCH and MERGE nodes can have an arbitrary number of link nodes corresponding to the alternatives in the case expression. The CASE-SELECTOR node represents the expression that determines which alternative is to be selected.

FAILURE MECHANISM

As described in section 6.1 an expression may deliver a fail signal instead of a value, causing a jump to the nearest surrounding expression that can catch the fail signal. This may be a convenient mechanism for the programmer, but it complicates the analysis considerably. It has two consequences for the demand graph. First, nodes that can generate signals have to be connected to expressions that can catch the signals. Secondly, the demand graph has to encode “hidden jumps”: if the head of an expression may fail, side-effects in its tail should be treated carefully.

Let us first ignore the second problem and concentrate on the separation of values from signals. Fortunately, with the exception of a few built-in classes and functions (*get*, *integer*, *real*), the nodes that create a value are distinct from the ones that generate a signal. We call these two sorts of nodes *value* and *signal* nodes. We can also separate arcs that carry values from the ones that carry signals.

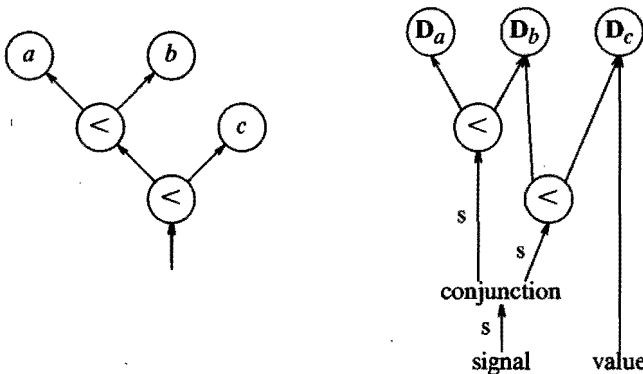


Figure 6.6. Separation of value and signal arcs.

Left the syntax tree and right the demand graph for the expression $a < b < c$. Signal arcs (marked with *s*) can only lead to signal nodes such as \ominus and value arcs to value nodes such as \oplus . The value of the expression is the value of *c*. The signal generated by the expression is a conjunction of two signals, which is encoded by a BRANCH node as we shall see below.

As illustrated in figure 6.6 this separation of value and signal nodes requires new arcs in certain expressions. These are made via the chainer by means of the two pseudo-names *Value* and *Success*. Just like value nodes announce themselves by issuing a call *def(Value, self)*, signal nodes issue a call *def(Success, self)*. New arcs to value nodes can

be made by calling on *use(Value)*, whereas signal nodes are accessible via *use(Success)*.

The second problem, that of encoding hidden jumps, can be solved by means of the same CONDITIONAL-COCOON that was used for the *if* expression. The expression

A & B

where A and B are expressions, is equivalent to

if A then B else fail fi

B is only evaluated when A succeeds. A similar transformation applies to all dyadic operators <dyop>:

A <dyop> B

becomes

if tmp := A then tmp <dyop> B else fail fi

To obtain the effect of this transformation a conditional cocoon could be created whenever a dyadic operator is encountered. The cocoon would introduce MERGE and BRANCH nodes for all values that would enter and leave the right operand, just as with an *if* expression.

This scheme provides a correct implementation of the evaluation mechanism, but would give an explosive growth of the number of nodes in the demand graph. The introduction of a new cocoon is therefore postponed until it has been determined to be necessary. A dyadic operator creates a cocoon, when it detects that its left operand may fail, signalled by a *def of Success*.

attach of DYOP	(Simplified)
attach left operand	
if Success in deflist	
install cocoon	
treat-right-operand within then-chainer	
dissolve cocoon	
else	
treat-right-operand	

Each type of dyadic operator has its own version of the procedure *treat-right-operand*.

AND AND OR NODES

Because of the general failure mechanism the '&' operator is the most rudimentary dyadic operator: it simply glues two expressions together without generating any values or signals. In fact in SUMMER programs one often encounters an '&' operator where in other languages a ';' would be found. AND and OR nodes are left out of the demand graph just as SEQUENCE nodes are: their function is fully interpreted during demand graph construction. Figure 6.7 illustrates the translation of an AND node into BRANCH and MERGE nodes for the pseudo-name *Success*.

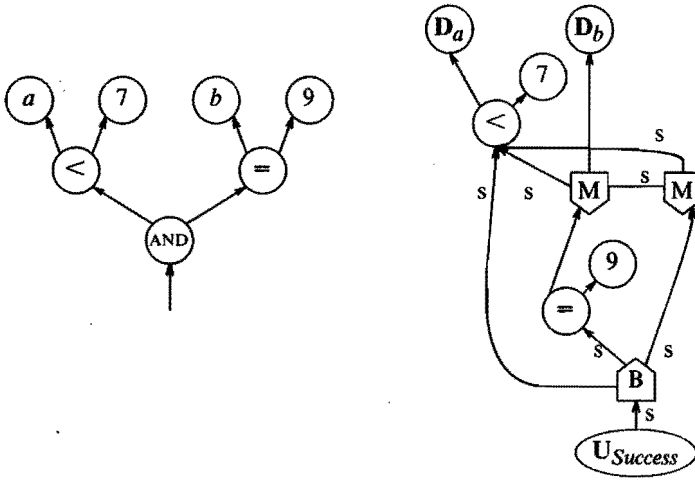


Figure 6.7. Translation of the AND node.

The AND node is translated into BRANCH and MERGE nodes. The left operand of the AND node may fail, so the right operand is attached within the *then* chainer of a CONDITIONAL-COCOON. This creates an exposed definition of *Success*, which causes an induced use and the creation of the rightmost MERGE node. This node is connected to the defining node of *Success* in the surrounding expression, which happens to be the same \ominus node that controls the cocoon. This connection of BRANCH and MERGE nodes for *Success* is equivalent to a conjunction, as will be shown in the next figure.

For the benefit of certain applications each CONDITIONAL-COCOON checks whether any branch contains side-effects, i.e. whether the *deflist* of its chainer contains anything beyond the pseudo-name *Success*. This information is recorded in each interface node created by the cocoon. Figure 6.8 summarizes the translation from AND and OR nodes into BRANCH and MERGE nodes.

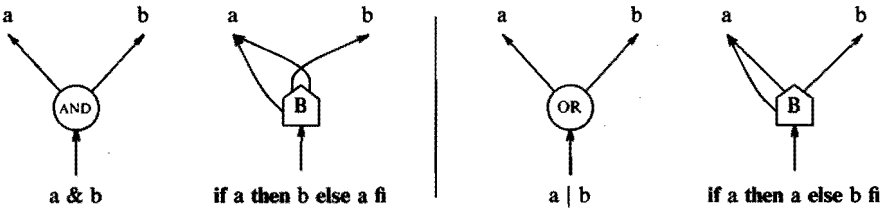


Figure 6.8. Translation of AND and OR nodes.

By connecting the three outgoing arcs of a BRANCH node to two operands it can serve as AND or as OR node. The LINK and MERGE nodes have been omitted from the figure.

CONDITIONAL EXPRESSIONS IN ADDRESS OR VALUE CONTEXT

The shuffling of the *Address* and *Value* pointers by the ASSIGN node, which may have appeared overly complicated in the previous section, provides, in combination with the CONDITIONAL-COCOON, exactly the right interface nodes when conditional expressions appear outside a *void* context. Figure 6.9 shows the two basic cases.

$a := \text{if test then } x \text{ else } y \text{ fi}$

$\text{if test then } x \text{ else } y \text{ fi} := a$

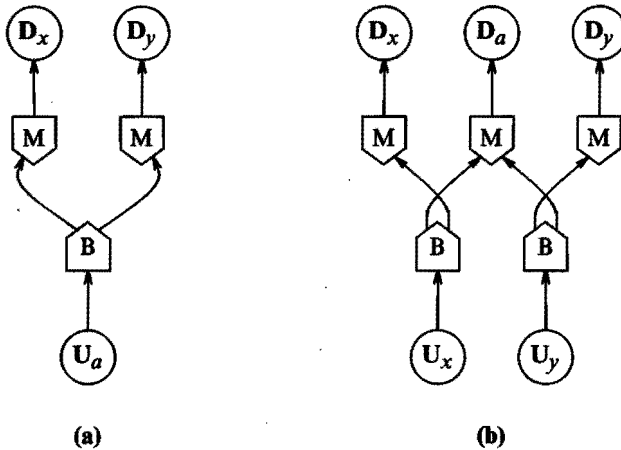


Figure 6.9. Conditional expression in value and address context.

(a) In a *value* context the VARIABLE nodes will both issue a call of *def* for *Value*, resulting in one BRANCH node. The calls of *use* issued by the VARIABLE nodes cause the creation of the MERGE nodes.

(b) In an *address* context a VARIABLE node issues both a call of *use* for *Address* and a call of *def* for the variable name. For the two definitions two BRANCH nodes are created, which in turn induce two uses. A total of three MERGE nodes are therefore needed. This graph is more complicated than the one in (a), because the expression has a stronger effect on the use-definition relationships.

ITERATION

The treatment of *while* and *if* expressions are remarkably alike. When a WHILE-LOOP node is encountered, a LOOP-COCOON with two chainers is created. Since both the *body* and the *test* expression may contain side-effects, each is attached within its own chainer. When the cocoon is dissolved the interface nodes that are created are connected in a way that may lead to cycles, reflecting the cyclic data dependencies that a loop may introduce.

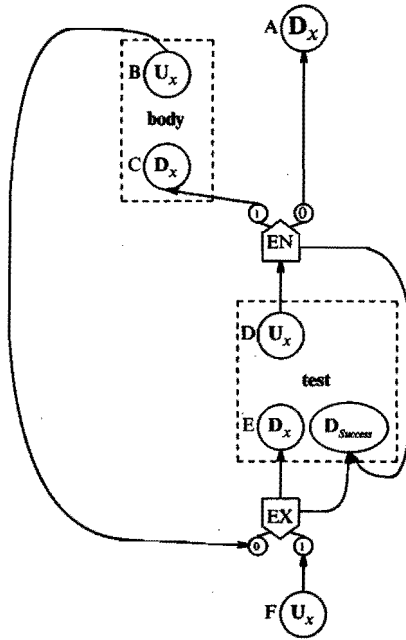


Figure 6.10. The interface of a *while* expression.

The demand path is illustrated for a variable x that is both used and defined in the test as well as the body. For the first iteration, the exposed use of the test (node D) should be connected to the previous definition in the surrounding expression (node A). For subsequent iterations node D should be connected to the exposed definition of the body (node C). This ambiguity is encoded by the ENTRY-LOOP node, which functions like a BRANCH node. The exposed use of the body (node B) and the subsequent use in the surrounding expression (node F) are connected to the exposed definition of the test via an EXIT-LOOP node, which functions like a MERGE node.

Figure 6.10 illustrates this process for the most general case: a variable that is defined and used in both branches. In other cases some arcs or nodes may be left out. In the usual case, where the test does not define variables, the EXIT-LOOP node is directly linked to the ENTRY-LOOP node. An EXIT-LOOP node is created for each variable that occurs in the loop. In the case when the loop does not define the variable a cycle is created consisting only of the two interface nodes and their link nodes. When the expression as a whole does not use the variable (i.e. it is always defined before it is used) the ENTRY-LOOP node is omitted. Figure 6.11 shows the complete demand graph of an example loop.

A for loop is a *while* loop with an empty *test* part and a special control node. The analysis of the two kinds of loops and the resulting demand graphs are almost identical. In the rest of the thesis we will ignore the differences.

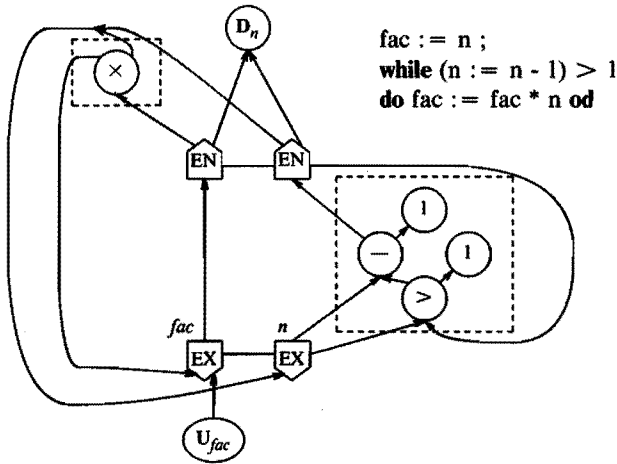


Figure 6.11. Demand graph for a loop that computes a factorial.

From this and subsequent illustrations link nodes have been omitted. The test expression uses and defines n and produces the signal that controls the interface node. The body uses both variables and defines fac . The use of fac in the body induces a use in the test, which in turn causes an ENTRY-LOOP node to be created.

6.5. Multiprocedural Graphs

The use of procedures may complicate the demand graph in several ways:

Global Variables

An expression may contain “hidden” uses or definitions due to global variables in any of the procedures that it calls.

Return Expressions

A subexpression may not be evaluated due to a return expression in a previous subexpression.

Recursion

The demand graph of an expression may have itself as a component.

Recursion will not be covered here; the previous chapter already explained how it is handled with the aid of the summaries of global uses and definitions collected during syntax tree construction (see also section 6.2). The other two issues are treated in this section.

GLOBAL VARIABLES

As already discussed in section 3.3, a procedure with global variables has a partially hidden interface: an exposed use of a global variable is a hidden parameter, a definition a hidden return value. Each program can be transformed into an equivalent one without global variables by replacing these hidden inputs and outputs by extra parameters and return values. During the construction of the demand graph a similar transformation takes place: hidden inputs and outputs are made explicit by interface nodes. Each input corresponds to a PARAMETER and each output to a RESULT node. When a PROC-CALL node is encountered, the interface nodes of the called procedure are connected to a corresponding set of local interface nodes at the calling site. These in turn are connected to the rest of the calling expression. Figure 6.12 shows the details of this interface.

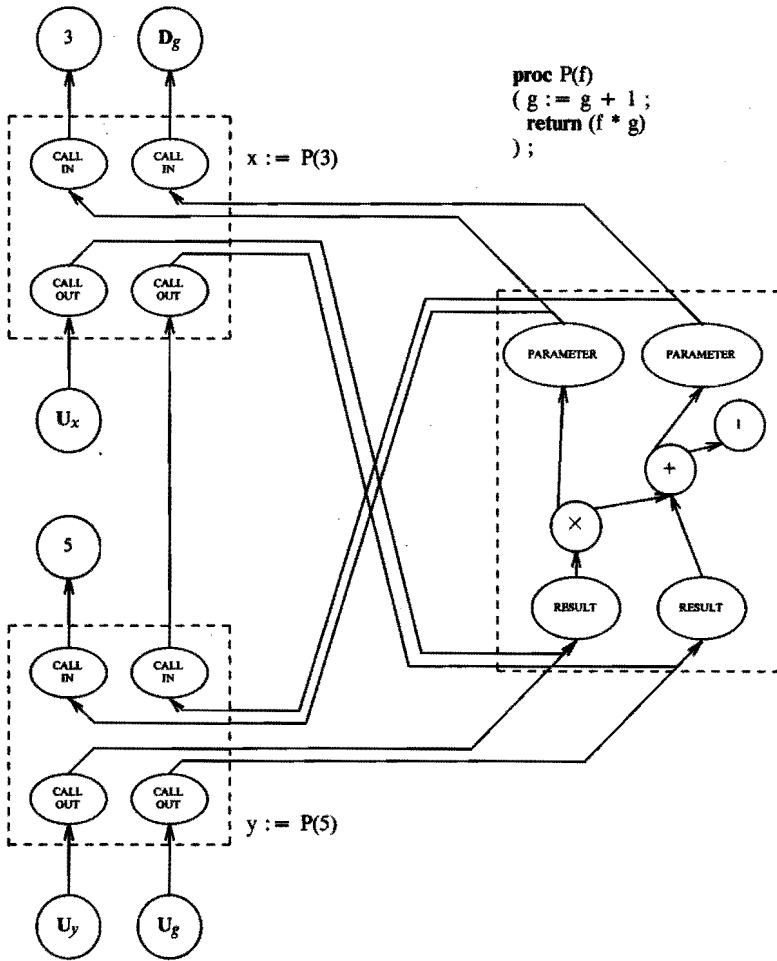


Figure 6.12. The interface of procedure calls.

On the right a procedure P that uses and defines the global variable g . On the left two calls of P without intervening definition of g . The input and output interface nodes of a procedure are called **PARAMETER** and **RESULT** nodes. These are connected to their local counterparts at the calling site (**CALL-IN** and **CALL-OUT** nodes), so each **PARAMETER** node has as many outgoing arcs as there are calls of the procedure. Note that the distinction between the exposed uses of a global variable and one of a parameter has disappeared. The same is true for a definition of a global variable and an explicit return value. It is also interesting to note that cycles may be created even without recursion.

The interface nodes of the called procedure are only available if its demand graph has already been constructed. If this is not the case the procedure is analyzed first before proceeding. The effect is that procedures are analyzed depth first with respect to the calling graph, so that a definition of a global variable in a deeply nested procedure becomes visible in all intermediate layers. In case of recursion the summaries compiled during syntax tree construction have a similar effect: if one member of a strongly connected component of procedures in the calling graph defines a global variable, it becomes visible in all members of the component.

RETURN EXPRESSIONS

The evaluation of a **return** expression has two effects:

- If there is an operand it is evaluated. If it fails the procedure fails, otherwise its value becomes the return value of the procedure.
- The evaluation of the current procedure is aborted and evaluation of the calling expression is resumed.

An **freturn** expression is equivalent to a **return** expression with failing operand. The *attach* procedures of RETURN and FRETURN have to simulate these two effects.

The first effect is simulated by means of two new pseudo-names: *Return-value* and *Return-signal*. When a RETURN node is attached these pseudo-names are made to point to the defining nodes for *Value* and *Success* entered by the operand. If there is no operand, *Return-signal* is made to point to a CONSTANT node *Always*, which encodes the boolean value *true*.

The second effect, that of the escape, is simulated by means of the pseudo-name *Returns* causing the appropriate cocoons to be generated. If an expression contains a **return** or **freturn** expression, its attachment causes a definition of *Returns*, just as an expression that may fail causes a definition of *Success*. This pseudo-name has the same function as the pseudo variable in the transformation illustrated in the previous chapter in figure 5.10(c). Its effect on the demand graph is the creation of exactly those BRANCH and MERGE nodes that encode the boolean expressions of figure 5.10(b).

```
attach of RETURN
  def>Returns, Always)
  if there is an operand
    attach operand
    def(Return-value, use(Value))
    def(Return-signal, use(Success))
  else
    def(Return-signal, Always)
```

The mechanism for handling failure escapes, described in the previous section, is extended to handle return escapes. A similar mechanism using the pseudo-name *Exits* handles ASSERT and STOP nodes, which represent escapes on program level rather than procedure level.

Procedures are attached within a separate chainer and a PROC-COCOON. When this cocoon is dissolved *Return-value* and *Return-signal* are converted back into *Value* and *Success*, and all inputs and outputs are connected to PARAMETER and RESULT nodes, which are stored in tables in the PROCEDURE node.

```
dissolve of PROC-COCOON
  for each [name,node] in deflist
    if name is global variable
      outputs[name] := RESULT(node)
    else if name is Return-value
      outputs[Value] := RESULT(node)
    else if name is Return-signal
      outputs[Success] := RESULT(node)
  for each [name,node] in uselist
    if name is global variable
      inglobals[name] := node
    else if formal parameter
      formals[position of formal] := node
```

When a PROC-CALL is attached, CALL-IN and CALL-OUT nodes are created to form the local interface. Default definitions for *Returns* and *Return-value* are provided by the *attach* procedure of PROCEDURE.

Figure 6.13 may clarify the process. In simple expressions, such as this one, superfluous nodes may be created: the BRANCH node pointing to *Always* and *Never* functions as an identity node. The body of the procedure is in effect transformed into

$$g := \text{if } 0 < g \text{ then } g \text{ else } 2 \text{ fi}$$

If the return had appeared in the else branch of the original if expression, the arcs would have been interchanged and the BRANCH node would function as a boolean NOT.

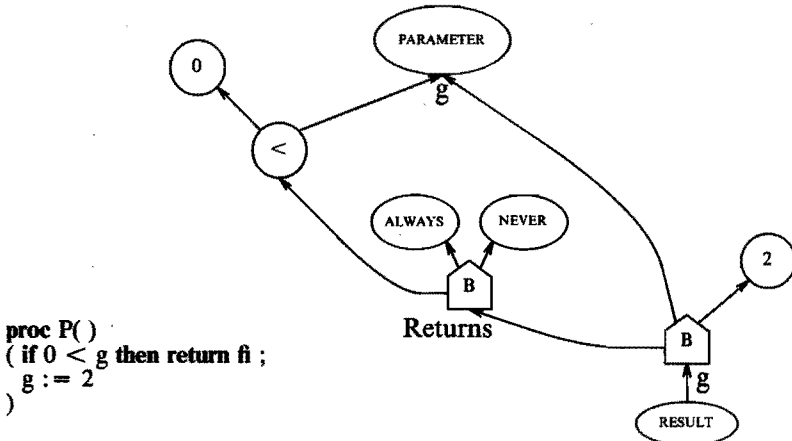


Figure 6.13. The effect of a return escape.

Procedure *P* may or may not assign a new value to global variable *g*. This ambiguity is encoded in the BRANCH node on the right, which is controlled by the boolean value produced by the left BRANCH node. The latter has an arc to *Always*, which is the definition of *Returns* issued by the RETURN node, and an arc to *Never*, which is the default definition for *Returns*. The return expression has no operand, so there is no path for *Return-value*. To avoid clutter the path for *Return-signal* has been omitted from the illustration.

6.6. Arrays

The objects we have encountered so far (strings, integers, reals), cannot be changed: the only way to give a variable another value is to assign a new object to it. An object of type *array*, however, can be modified by means of the *update* operation, which is written as an assignment. For instance, after

```
a := [ 0, 'abc', 3.5 ] ;
a[1] := 'xyz'
```

the element with index 1 has been updated and *a* now has the value [0,'xyz',3.5]. As a consequence of this selective update operation, objects may be partly redefined and cyclic data structures¹ may come into existence. Partial redefinitions complicate the demand graph, since aliasing can no longer be ignored: an update of an array is visible

1. A data structure is cyclic if the graph of pointers that implement the data structure contains a cycle.

through all its aliases. Cyclic data structures are possible because an arbitrary object can be assigned to an array element. After, for instance,

```
a := ['pqr'];
a[0] := a
```

the original value of the only element of *a* (i.e. 'pqr') is replaced by the value of *a* itself, i.e. *a* is an array with itself as only element. Cyclic data structures complicate the aliasing problem considerably; they have been omitted from the current implementation. The same holds for interprocedural aliasing.

In this section the handling of arrays and simple aliasing is treated. Handling conditional aliasing efficiently is a difficult problem and the next section will be devoted to the rather complicated algorithm that has been developed for this.

ARRAY AND ARRAY-ACCESS NODES

An assignment to a variable is a complete redefinition: none of the information of a previous assignment will be available any more. Several assignments to the same variable are therefore unrelated and do not have to be kept in order. In contrast, the update of an array element is a partial redefinition of the array: previous updates may still have an effect on subsequent retrieves. The order of several updates on the same object has to be maintained.¹ This is reflected in the demand graph by linking all updates of the same object into a chain. Since the order of several retrieves between two updates is irrelevant, each retrieve is linked to its previous update.

Chainers are used to maintain information about updates. Each update is stored in the *deflist* under a key that uniquely identifies the object. A variable name cannot be used for this purpose, since it represents just one name for an object for which there may be several aliases. Another object may be assigned to a variable, while the original object remains unchanged and still accessible through one of the aliases, as in the expression

```
a := [3];
b := a;
a[0] := 1;
a := 0;
b[0] := 2
```

where the two updates are to the same object and so have to be linked to each other, independent of the intervening reassignment to *a*. In straight-line code the defining node of a variable provides a unique identification of an object. When crossing cocoon boundaries, however, this way of identification is not sufficient. A new field *origin* is therefore defined for each node. For nodes that may represent an array object this field points to a node that uniquely identifies the object. For nodes that only represent simple objects *origin* has the value *Simple*.

The creation of a new object of type array is represented in the syntax tree by an ARRAY node. It has outgoing arcs to nodes that represent the initial values; these arcs are omitted from the illustrations. Since the ARRAY node represents the creation of a

1. Strictly speaking two updates of an array with different subscripts do not have to be kept in order. Detecting this would require a kind of analysis that is deferred to the application specific part, keeping with the principle that the form of the demand graph, is not influenced by values of constants (see the remarks made about loops in section 5.2.2).

completely new object, it is itself a unique identifier for that object, so its *origin* field is made to point to itself.

```
attach of ARRAY
  origin := self
  def(Value, self)
  def(origin, self)
(Simplified)
```

The expression 'a[0]' may be either a retrieve or an update depending on the context. Both type of accesses are represented by ARRAY-ACCESS nodes. A call of the procedure *find-context* (see section 6.2) marks these as either retrieve or update. Figure 6.14 gives an example.

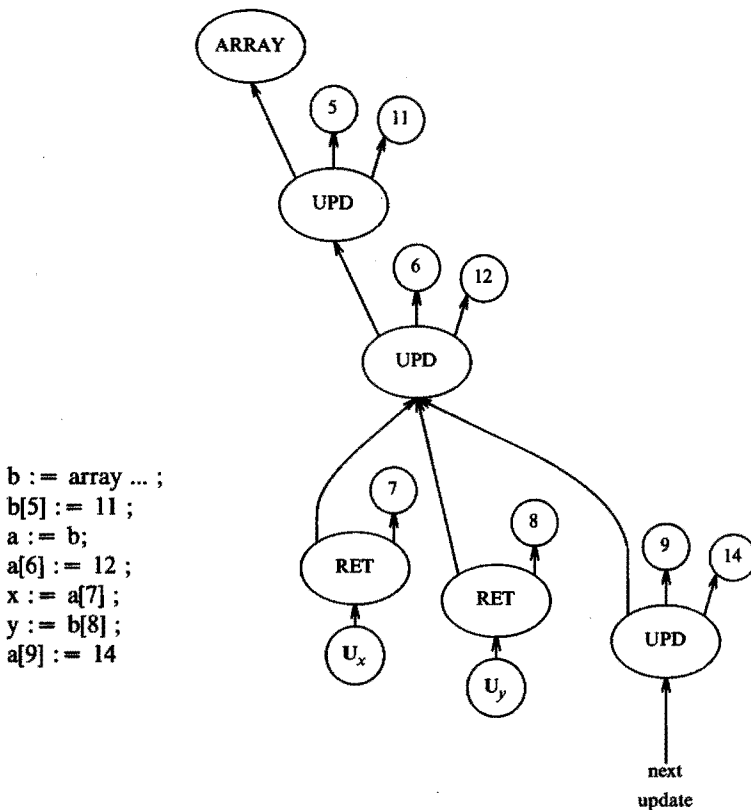


Figure 6.14. Unconditional aliases in straight-line code.

Variables *a* and *b* both point to the same array. All ARRAY-ACCESS nodes (in the figure marked with RET for a retrieve and with UPD for an update) are linked to each other through their *previous-update* field. It is as if each update node represents a new array that is the combination of the new element and the array represented by the previous update. Note that the relative ordering of several retrieves between two updates is lost.

The *attach* procedure of an ARRAY-ACCESS node determines the defining node of the object (*object-source*) through which the unique identification can be found to be used by procedure *connect-to-previous-update*. In addition an update defines itself as the currently last update.

```
attach of ARRAY-ACCESS (representing an update) (Simplified)
  source := use(Address)
  attach index
  attach object
  object-source := use(Value)
  connect-to-previous-update(object-source.origin)
  def(object-source.origin, self)
```

```
connect-to-previous-update(object-origin) of ARRAY-ACCESS (Simplified)
  previous-update := use(object-origin)
```

As long as no conditional aliasing is involved, procedure *connect-to-previous-update* can simply retrieve the previous update from the chainer. This treats unconditional aliases in straight-line code correctly, since they point to the same defining node and so accesses through two different aliases get the same *object-source*.

ACCESSES FROM WITHIN A CONDITIONAL

If two names *a* and *b* are made aliases of each other, in the rest of the same straight-line code *deflist['a']* will be equal to *deflist['b']*. If after this a conditional expression is encountered, the aliasing should also be reflected in the subgraphs constructed for its branches. This means that these subgraphs can no longer be constructed in isolation. To detect that names are aliases, the procedure *use* is extended such that, whenever an exposed use of an array is encountered, the *origin* information is imported from the environment. The environment is the current chainer of the surrounding cocoon. The effect is that, for each exposed use in a nested expression the stack of chainers is searched until a definition is encountered.¹ The *origin* of this node is then copied to the entry nodes created at all intermediate levels. So the fact that *a* and *b* are (unconditional) aliases is reflected in the equality of *deflist['a'].origin* and *deflist['b'].origin*. When the cocoon is dissolved, BRANCH nodes are created for each object for which the conditional expression contains an update. No adaptation is required of the cocoons as described in the previous sections.

1. This search will not extend beyond the current procedure, since interprocedural aliasing is not allowed.

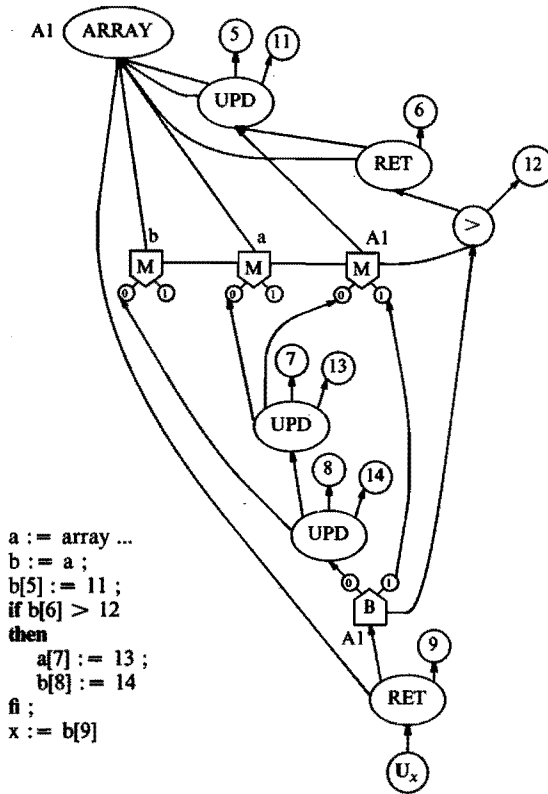


Figure 6.15. Updates within a conditional.

The left-most arc of each access is the *object-source* arc; the next one the *previous-update* arc. The then-branch contains two updates of the same object through the two aliases *a* and *b*. For the two names two separate LINK-IN nodes are created but both have the same origin *A1*. The second update therefore gets linked to the first update, which in turn is an exposed use of the origin node *A1*. When the cocoon is dissolved, a MERGE and a BRANCH node for the key *A1* are created. The MERGE node is linked to the last update of the object before the conditional expression. A subsequent access is linked to the new BRANCH node.

ACCESSES FROM WITHIN A LOOP

The two branches of a while expression are treated almost identical to the branches of an if expression. The only difference is that while in the latter case, the branches are alternatives, in the while expression the test is always executed before the body. The environment for the body is therefore the test; for the test it is the surrounding expression. Otherwise an array defined in the test would not be treated correctly. A typical loop that updates all elements of an array is shown in figure 6.16. This example is free of aliases. Since the variable *i* takes on a different value in each iteration, the updates in the body are, strictly speaking, independent and do not have to be linked to each other. The cycle could therefore be removed. This requires, however, a type of analysis that belongs to the application domain, since it involves taking the values of constants into account. We will come back to this issue in section 8.6.

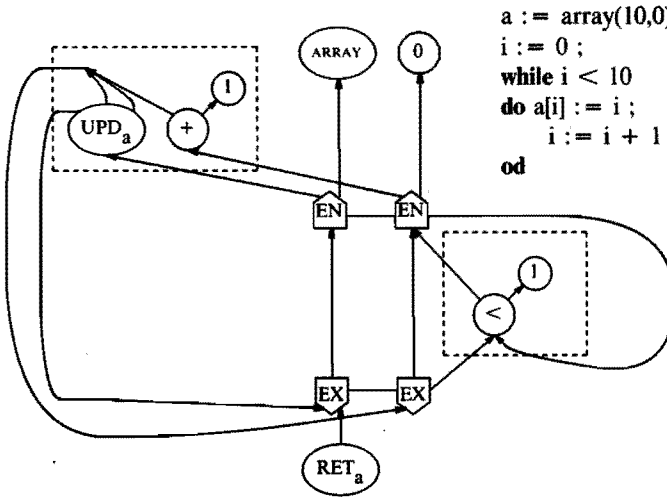


Figure 6.16. Updates within a loop.

The update in the body gets its *origin* information by searching the *test* expression and the surrounding expression for the previous definition of *a*. The *object-source* arcs and the corresponding nodes have been left out of the figure. The cycle that links the update to itself indicates that updates of subsequent iterations have to be kept in order.

6.7. Conditional Aliasing

If the aliasing between two variables is determined by a condition that is not evaluated statically, we call these variables *conditional aliases*. For instance, after

```

a := [0];
b := [1];
c := if test then a else b fi
    
```

it depends on the success of *test* whether *c* and *a* denote the same object or not. After this expression an access through *c* has to be linked to either the previous update through *a* in case *test* succeeds or to the previous update through *b* in case *test* fails. Since this ambiguity cannot be resolved statically, it has to be expressed in the demand graph. For this purpose we introduce ACCESS-BRANCH nodes that provide paths to the alternative updates and to the node that determines the proper path at run time. These nodes behave exactly like the BRANCH nodes we encountered before. Consider the example in figure 6.17. It appears at first that an ACCESS-BRANCH node has to be created for every access through *a*, *b*, or *c*. If the aliasing relation involves more than one condition, each access requires a graph of several ACCESS-BRANCH nodes. Such a subgraph linking an access to previous accesses of conditional aliases we call an *alias access graph*.

The alias access graphs grow with the complexity of the aliasing relation. This suggests that the number of nodes that are needed just to encode the access path ambiguity, is proportional to the product of the number of accesses and the average complexity of the aliasing relation. Since in any program conditional expressions are abundant and, in addition, in SUMMER programs aliasing is wide-spread, the average complexity of the aliasing relation is very high. Consequently, the size of the demand graph for an average SUMMER program would be dominated by the number of ACCESS-BRANCH nodes. This makes the direct encoding of the aliasing relation into the demand

graph impractical in the general case, but an efficient algorithm that constructs reasonably small alias access graphs for all programs in a restricted, but interesting, class would still be of great value.

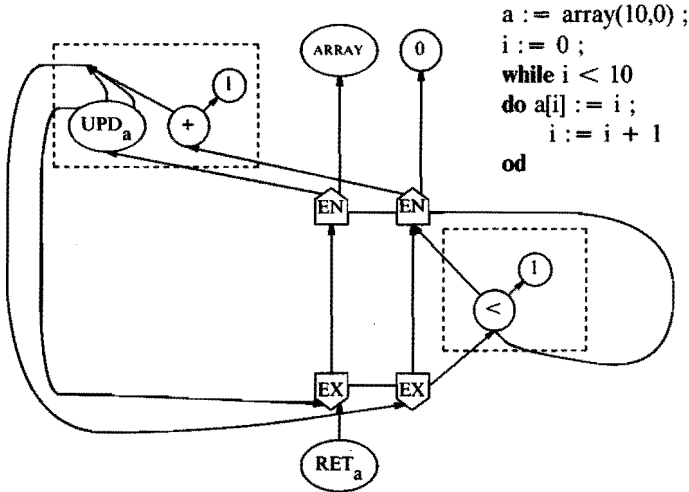


Figure 6.17. ACCESS-BRANCH nodes.

The conditional expression makes *a* and *c* as well as *b* and *c* conditional aliases of each other. An update of *c* (marked with UPD_{*c*}) is linked to the previous updates of both its conditional aliases via an ACCESS-BRANCH node (marked with AB). The latter node also points to the expression that controls the aliasing.

For many programs the number of ACCESS-BRANCH nodes can indeed be reduced substantially. Note for instance that, in the example above, two accesses through *c* without intervening access through either *a* or *b* can be connected without an ACCESS-BRANCH node. Moreover, accesses through *a* and *b* do not have to be connected to each other, since *a* and *b* are not aliases of each other, although they have a conditional alias in common. A reasonable assumption to make is that, in a typical program, complicated aliasing relationships may be created, but that locally the number of names through which an object is accessed is rather limited. To make the handling of conditional aliasing practical an algorithm is needed that exploits this locality to construct small alias access graphs in a reasonably short time.

THE LACAP ALGORITHM

We present an algorithm that constructs small alias access graphs in a time that is proportional to the number of ACCESS-BRANCH nodes. It has been called the LACAP algorithm after a set of pointers in the demand graph (the *last accessed conditional alias pointers*) whose selective maintenance is the key to its efficiency. It can handle all aliasing in programs without interprocedural aliasing or multi-dimensional arrays, but in this presentation we initially assume that all conditional aliases are due to if expressions. Including conditional aliasing caused by case or while expressions is straightforward as we shall see later.

The first problem is how to represent the aliasing information. Fortunately most of the aliasing information is already available in a convenient format. As we saw in the previous section the *deflists* together with the *origin* information form a mapping from variable names to nodes such that unconditional aliases are mapped to the same node.

For each conditional alias a **BRANCH** node with its **LINK-OUT** nodes is formed. If we let each **LINK-OUT** node copy its *origin* pointer from its successor, subgraphs are formed consisting of **BRANCH**, **LINK-OUT**, and **ARRAY** nodes. These graphs, which contain all necessary alias information, we call *alias graphs*. Figure 6.18 shows an example.

```
(a, b, d, and e are arrays)
c := if t1 then a else b fi ;
f := if t2 then d else c fi ;
g := if t3 then c else e fi ;
```

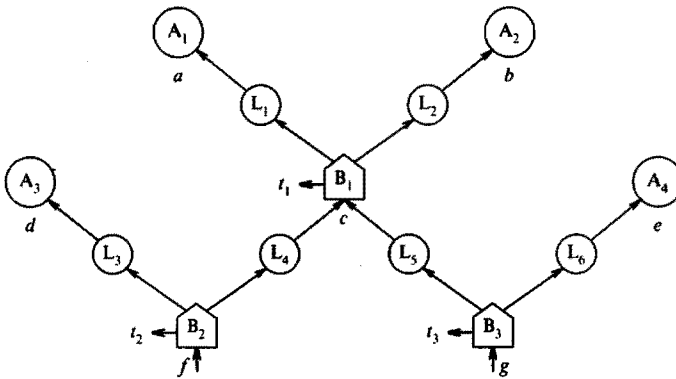


Figure 6.18. An Alias Graph.

A set of conditional expressions and the resulting alias graph. The outgoing arcs of each node point to previously created nodes. Alias graphs are consequently acyclic. They consist of **ARRAY** nodes as sinks and **BRANCH** and **LINK-OUT** nodes as internal nodes. Each **BRANCH** node has its corresponding **LINK-OUT** nodes as successors and each **LINK-OUT** node points with its *origin* field to either a **BRANCH** node or an **ARRAY** node. The variable names under **ARRAY** and **BRANCH** nodes indicate the mapping through *deflist* and *origin*.

Note how not all variables in this graph are conditional aliases of each other. *c* is a conditional alias of *a*, but *b* is not: for no value for any condition could *b* and *a* become aliases of each other. *g* and *f*, however, are aliases, since if *t₂* fails and *t₃* succeeds *g* and *f* will point to the same object (the one *c* is pointing to).

During demand graph construction alias graphs change frequently and can grow with sudden jumps: the analysis of a conditional assignment can cause the connection of two arbitrary large alias graphs. An algorithm that relies on information that has to be maintained globally per alias graph is therefore unfeasible. The LACAP algorithm stores information in the nodes of the alias graph that does not have to be updated each time the alias graph grows, but only during the construction of an alias access graph. We give a functional description of the algorithm before we describe its implementation

FUNCTIONAL DESCRIPTION

When, during analysis, an access is encountered the LACAP algorithm traverses part of the alias graph, starting in the node to which the variable being accessed is mapped (the *accessed node*). For each **BRANCH** node visited it creates a corresponding **ACCESS-BRANCH** node in the alias access graph. To construct a small alias access graph the part of the alias graph to be visited has to be limited. To indicate the paths that have to be followed a pointer, called *lacap*, is associated with each node of the alias graph. Each pointer has one of three values *Laca*, *Ancestor*, or *Descendant*. It has the value *Laca* for

a node that is a *Last Accessed Conditional Alias* (a LACA), i.e. a node through which an access has been made but no subsequent access through any of its conditional aliases. If the accessed node is a LACA, no alias access graph needs to be constructed (i.e. the alias access graph is empty). For a node that is not a LACA the *lacap* pointer indicates in which direction a LACA can be found: it has the value *Descendant* if a LACA may be found through one of the descendants and *Ancestor* otherwise.

The *lacap* values within one alias graph have to be consistent. To define this consistency we introduce two relations between nodes of an alias graph:

- A node *a* is *more recently accessed* than a node *b*, different from *a*, if there has been no access to an object through a name mapped to *b* after the last access through a name mapped to *a*.
- We say that *a* is *linked to b*, if in the alias graph one of two conditions hold
 - *a* is an ancestor of *b*
 - *a* is not a descendant of *b*, but they have a common descendant

So in figure 6.18 B_3 is linked to all nodes except L_3 , A_3 , and B_3 itself.

With each node *N* a *lacap* is associated that has one of the following values:

Ancestor (for ARRAY, LINK-OUT, and BRANCH nodes)

if a node *x* linked to *N* has been more recently accessed than both *N* and all nodes that *N* is linked to.

Descendant (for BRANCH nodes)

if *N* is linked to a node *x* that has been more recently accessed than both *N* and all nodes that are linked to *N*.

Laca (for ARRAY and BRANCH nodes)

otherwise

The definition of *lacap* implies that within one alias graph there may be several LACAS, but they are never linked to each other.

This state of the *lacaps* amounts to an invariant to be maintained by the algorithm. Two actions may affect this invariant: an access or a change of the alias graph. The latter is easy to deal with through proper initialization. Alias graphs can only change through the addition of a BRANCH and its LINK-OUT nodes. When a BRANCH node is created it has no ancestor and no access through it can yet have been made, so its *lacap* is initialized to *Descendant*; the same holds for the *lacap* of a LINK-OUT node. When an ARRAY node is created, it is the only node of an alias graph, so its *lacap* is initialized to *Laca*.

When an access is encountered, the *lacap* of the accessed node and its surrounding nodes may have to be updated to maintain the invariant. The algorithm traverses the graph from the accessed node towards the LACAS it is linked to. The *lacaps* of the nodes in the other direction already have the correct value. The *lacaps* guide the algorithm to avoid the paths along which no LACA is to be found, visiting only nodes where the *lacap* needs to be updated and their direct neighbors. Other nodes can be avoided because when a node is reached whose *lacap* already has the correct value, the invariant implies that all nodes that can only be reached through that node also have the correct values.

Since the algorithm creates an ACCESS-BRANCH node whenever it visits a BRANCH node, and the time spent per visit is bounded by a constant, the time complexity of the LACAP algorithm is proportional to the size of the alias access graph that it creates. This size is small, if the accesses through conditional aliases show locality in the sense that, on the average, two subsequent accesses through conditional aliases correspond to nodes in the alias graph that are close together.

EXAMPLE

We follow the algorithm for a series of subsequent updates. We restrict ourselves to the more simple case where the alias graph is a tree. Refer to figure 6.19 for the alias access graphs that are created and the *lacap* values after each update.

```
(a, b, and d are arrays)
c := if t1 then a else b fi ;
f := if t2 then d else c fi ;
a[i] := 0 ;
c[j] := 1 ;
f[k] := 2 ;
b[l] := 3 ;
c[m] := 4 ;
```

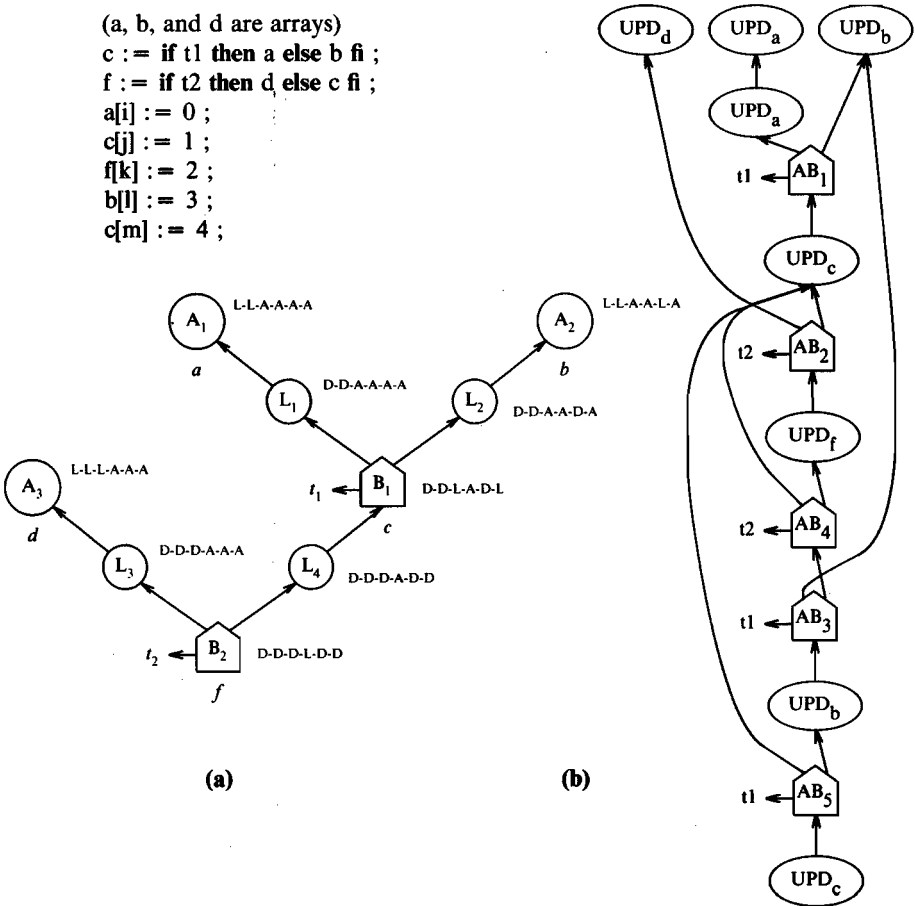


Figure 6.19. A series of updates in one alias graph.

(a) A subtree of the alias graph shown in figure 6.18. A string on the right of each node indicates the values of its *lacap* pointer at different times during analysis. The string encodes, from left to right, the initial value and the value after the analysis of each of 5 updates: 'A' stands for *Ascending*, 'D' for *Descending*, and 'L' for *Laca*.

(b) The alias access graphs created for 5 subsequent updates: of *a*, of *c*, of *f*, of *b*, and again of *c*.

Update of *a*

Initially only the ARRAY nodes are LACAS. An access through one of them can be connected directly to its previous update, since none of its conditional aliases are accessed yet.

Update of *c*

The algorithm starts in the accessed node B_1 , creates the ACCESS-BRANCH node AB_1 and descends the alias graph towards the LACAS A_1 and A_2 reversing the *lacaps* of L_1 and L_2 on the way. All descendants and ancestors of B_1 now have their *lacap* pointing towards it.

Update of *f*

ACCESS-BRANCH node AB_2 is created and the alias graph is descended, this time starting at B_2 , but the descent can stop at B_1 , since its *lacap* shows that no accesses through descendants of B_1 occurred after the previous update of *c*.

Update of *b*

The alias graph has to be traversed in the opposite direction. ACCESS-BRANCH nodes are created while climbing the graph.¹ The node AB_3 is created when B_1 is reached. Its left branch is linked to the previous update of *b*, since when t_1 succeeds *b* has no aliases. If t_1 fails *c* is an alias of *b* and maybe *f* too, depending on t_2 . This is encoded in AB_4 , which is created when the alias graph is climbed one stage further to B_2 .

Update of *c*

The *lacap* values are different from those at the time the analysis reached the previous update of *c*, since the ambiguity (represented by AB_5) now involves the previous update through *c* and *b*, but not the one through *a*.

The reader may convince himself that the alias access graphs that are created are sufficient by choosing a success/fail value for each condition and applying this set of values to both the original program and to the access graph. This partitions a set of conditional aliases into sets of direct aliases, as indicated in the following figure.

t_1	t_2	alias sets
succeeds	succeeds	{c,a} {f,d} {b}
succeeds	fails	{c,a,f} {d} {b}
fails	succeeds	{c,b} {f,d} {a}
fails	fails	{c,b,f} {d} {a}

Figure 6.20. A truth table for the two conditions in the previous figure.

If this procedure is followed, each alias access graph is reduced to a single arc. If the linking of the ARRAY-ACCESS nodes is correct for all sets of condition values, the alias access graphs are at least sufficient (if not necessarily minimal).

IMPLEMENTATION

We discuss the implementation of the algorithm, again restricting ourselves to the case where each alias graph is a tree. We saw in the previous section that an ARRAY-ACCESS node made a link to the previous update by the expression

connect-to-previous-update(object-origin) of ARRAY-ACCESS (Simplified)
 previous-update := use(object-origin)

This connection may now involve the creation of an alias access graph, so the accessed node of the alias graph is requested to provide the connection:

1. Each node in the alias graph has an extra arc to its predecessor to make climbing the graph (i.e. traversing towards the ancestors) possible. These extra arcs have been omitted from the illustrations.

```
connect-to-previous-update(object-origin) of ARRAY-ACCESS
  previous-update := object-origin.alias-access-graph
```

```
alias-access-graph of ARRAY and BRANCH
  return node returned by descend
  set lacap to Laca
```

The *descend* procedures of the alias graph nodes maintain the *lacap* invariant and create the appropriate ACCESS-BRANCH nodes.

```
descend of ARRAY and BRANCH (Simplified)
  case lacap of
    Laca:
      return node returned by use(self)
    Descendant:
      return new ACCESS-BRANCH node with each
        LINK-OUT node linked to descend of corresponding child
    Ancestor:
      return node returned by ascend(use(self)) of parent
  set lacap to Ancestor
```

Note that if the accessed node is a LACA, no ACCESS-BRANCH nodes are created and the call of *alias-access-graph* is equivalent to a call of *use*. If no conditional aliasing is involved, each ARRAY node is the single node of a (degenerate) alias graph and will always be a LACA.

When the accessed node is not a LACA, surrounding nodes need to be accessed to maintain the invariant and ACCESS-BRANCH nodes are created on the way. We distinguish two cases, the simpler of which is when a LACA can be found via a descendant. In that case *alias-access-graph* calls *descend*, which creates an ACCESS-BRANCH node and calls on *descend* of its two LINK-OUT nodes.

```
descend of LINK-OUT (Simplified)
  case lacap of
    Descendant:
      return node returned by descend of child
    Ancestor:
      return node returned by use(parent)
  set lacap to Ancestor
```

The first update of c in figure 6.19 illustrates this case. B_1 is the accessed node so *alias-access-graph* of B_1 calls *descend* of B_1 which creates the node AB_1 and calls on its children L_1 and L_2 to provide the appropriate connections. The LINK-OUT nodes transmit the *descend* signal to the ARRAY nodes, which return a link to their previous updates by calling *use(self)*. The new ACCESS-BRANCH node will be connected to these two ARRAY-ACCESS nodes and the *lacap* pointers are updated to reflect that the previous update through any of this set of aliases was through B_1 .

A LINK-OUT node with its *lacap* set to *Ancestor* prevents a series of *descend* calls to enter a path along which no LACA is to be found. See for instance the last update of c in figure 6.19, where the *lacap* of the LINK-OUT nodes have opposite values, due to the previous updates through c and b . The *descend* procedure of LINK-OUT takes care of this situation by returning the previous update of its parent rather than transmit *descend* to its child.

We now turn to the more complicated case where a LACA is to be found among the ancestors of the accessed node. In that case the *descend* procedure of the accessed node

calls procedure *ascend* of its parent including its previous update as a parameter. LINK-OUT nodes simply transmit the *ascend* signal to their parents adding an extra parameter to indicate the direction from which the *ascend* reaches the BRANCH node.

```
ascend(default-access) of LINK-OUT (Simplified)
  return node returned by ascend(default-access, self) of parent
  set lacap to Descendant
```

The update of *b* in figure 6.19 provides an example. This update should be linked to the updates of *c*, of *f*, and of *b*. First *descend* of the accessed node A_2 detects that the graph has to be climbed and calls *ascend* of its parent L_2 . This LINK-OUT node transmits the *ascend* signal to its parent B_1 , which creates a new ACCESS-BRANCH node AB_3 . The branch that does not correspond to aliasing with the accessed node A_2 (t_1 succeeds) is linked to the *default-access*. The other branch is linked to the node delivered by a recursive call of *ascend*. However, if no LACA is to be found among the ancestors, i.e. a LACA is reached or a descendant along another branch is a LACA, this branch is linked to the previous update of the current node.

```
ascend(default-access, requesting-node) of BRANCH (Simplified)
  return new ACCESS-BRANCH node with each
  LINK-OUT node linked to either:
    if branch corresponds to requesting-node
      if lacap = Ancestor
        ascend(use(self)) of parent
      else
        use(self)
    else
      default-access
```

ALIAS GRAPHS THAT ARE NOT TREES

When we drop the restriction that alias graphs should be trees, three complications arise.

- During an *ascend* all predecessors have to be accessed rather than just the one parent.
- Conditional aliasing may be due to two nodes having a common descendant in the alias graph. A descend of the graph that reaches a node with *lacap* set to *Ancestor* therefore has to be followed with an ascend along the other incoming paths. The descendants of the particular node do not have to be visited, since their contribution to the alias access graph has already been created during a descend for a former alias access graph and can be shared. To retrieve the proper ACCESS-BRANCH node all nodes created during a descend are stored in the *deflist*. Since these do not represent real definitions they are especially marked so as to be discarded when the cocoon is dissolved.
- During the construction of one alias access graph the same node may be reached along two different paths causing the creation of erroneous nodes. A marker uniquely identifying each request is therefore transmitted through all calls and remembered in each node.

The final versions of the procedures can be found in appendix II.

This mechanism is illustrated in figure 6.21. The update of *f* initiates a descend along B_2 and B_1 just as described before. The node AB_1 created by B_1 is stored in the *deflist* as the last update of *c*. For the update of *g* a descend is started in B_3 resulting in the creation of node AB_6 . The descend on the right branch along L_6 and A_4 is as described above and results in a link to the last update of *e*. Along the other path the

traversal changes direction in B_1 and calls on *ascend* of L_4 . This node transmits the signal to B_2 , which creates AB_7 .

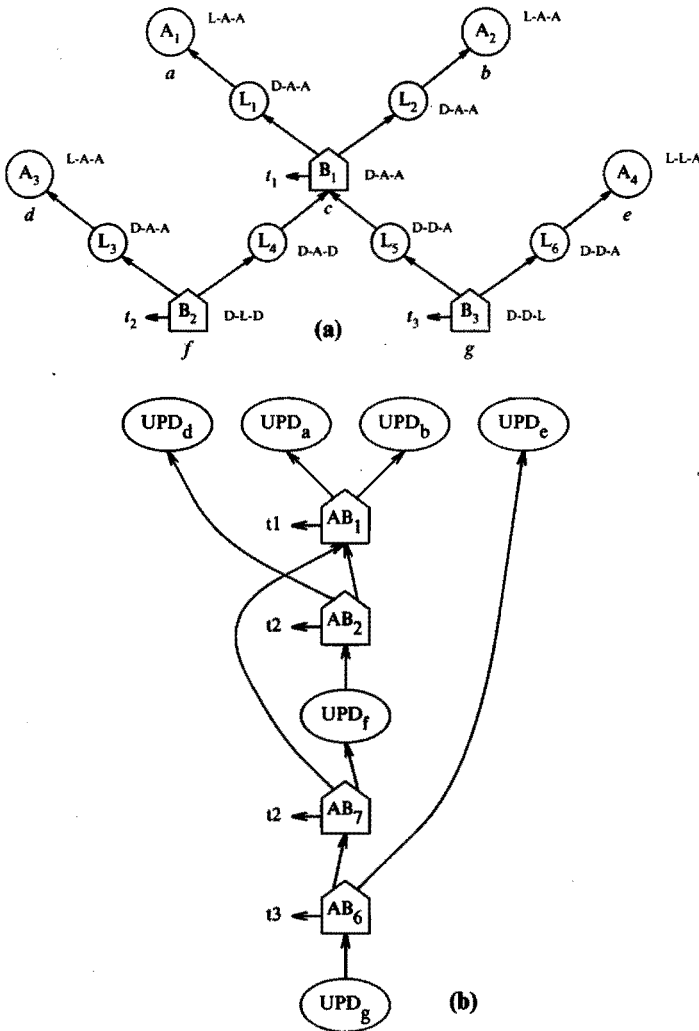


Figure 6.21. Aliasing due to a common descendant.

(a) The alias graph of figure 6.18. It contains the tree of figure 6.19(a) with the same initial *lacap* values. g and f are conditional aliases, because they are both ancestor of B_1 .

(b) The alias access graph for an update of f followed by an update of g . The latter has to be linked to the former in case t_3 succeeds and t_2 fails.

CROSSING COCOON BOUNDARIES

The *lacap* administration is local to a chainer: the different branches of a conditional can be analyzed in an arbitrary order and thus the *lacap* values should be the same at the start of each branch. Therefore, *lacap* values are stored in the current chainer. When the value of a *lacap* is requested but not available in the current chainer it is imported from the environment.

This leads to two problems when a cocoon is dissolved:

- The access history as expressed in the local *lacap* administration of the different branches has to be exported to the surrounding chainer. This is accomplished by simulating an access in the surrounding expression to each node that is a LACA in any branch.
- When a BRANCH node for an array object is created, a new alias graph is in effect created combining two smaller alias graphs. In some cases (nested conditionals or the creation of an array within a conditional) the *lacap* states of the constituent alias graphs have to be made consistent with each other by simulating accesses within the separate branches.

The details of the algorithm dealing with these special cases will not be presented.

CASE EXPRESSIONS, LOOPS, AND PROCEDURES

Case expressions lead to BRANCH nodes with more than two outgoing arcs. The algorithm as presented above is already formulated independent of the number of descendants of BRANCH nodes and is therefore capable of handling general conditional expressions.

The inclusion of conditional aliasing due to loop expressions is nearly as simple, since the demand graph created for a loop is similar to that created for an if expression, with ENTRY-LOOP nodes serving the role of BRANCH nodes. To extend the algorithm described above to include loops, all references to "BRANCH nodes" are simply replaced by references to "BRANCH and ENTRY-LOOP nodes."

The algorithm is extended to include arrays that cross procedure boundaries by treating PARAMETER and CALL-OUT nodes as if they were ARRAY nodes. Interprocedural aliasing, however, cannot be handled.

References

- Klin80. KLINT, P. (Jan 1980). An Overview of the SUMMER Programming Language, *Seventh Annual Symposium on Principles of Programming Languages*, 47-55, ACM.
- Klin82. KLINT, P. (1982). *From SPRING to SUMMER*, Mathematical Centre, Amsterdam.
- Tarj72. TARJAN, R.E. (1972). Depth-First Search and Linear Graph Algorithms, *SIAM Journal*, 1.2, 146-160.
- Veen80. VEEN, A. (Dec 1980). *Using Conventional Languages to Exploit Data Flow Machines*, Dissertation proposal, Internal memo.

Chapter 7

Demand Propagation

Once a demand graph has been constructed initial information can be deposited in the nodes and the information propagation can start. The design of the demand propagation part of an application is concerned with the following issues:

Assertion Space

A choice must be made as to what information is collected and how it is represented.

Assertion Lattice

A partial order and minimum element has to be defined to structure the assertion space into a semilattice.

Initial Assertions

Some nodes should be initialized with an assertion larger than the minimum one.

Propagation Rules

The set of propagation rules determines the direction of information flow: forward, backward, or mixed.

Propagation Control

A scheduler must be designed that determines the order in which nodes are processed. The processing order should be efficient and fair, i.e. each node whose neighboring assertions have changed will eventually be processed.

Termination is guaranteed under the following conditions:

- The assertion lattice is bounded, i.e. each of its chains is finite.
- Each propagation rule is order-preserving, i.e. it never replaces an assertion with a smaller one.
- A node that has been processed is only rescheduled for processing, if any of its neighboring assertions changes.

It is awkward to describe the application specific part without being able to refer to a specific application. The Value Approximation application, briefly discussed in chapter 4, is used as the principle example in this chapter. It is presented in the first section making two assumptions that greatly simplify the application: the demand graph is acyclic and forward propagation of information is sufficient. In the next sections these restrictions are removed. The second section considers complications due to cycles. In

the third section backward information flow is introduced with a simple application that has no forward component at all. The concluding section returns to the Value Approximation application and treats the interaction between forward and backward information flow.

7.1. Forward Propagation through an Acyclic Graph

The Value Approximation application includes traditionally separate applications such as constant folding, constant propagation, and static type analysis. During constant folding and propagation some of the computation is performed statically that would normally be done at run-time.

In a language like SUMMER in which the types of variables do not have to be declared and may even vary, type checking is usually postponed until run-time. Doing part of this at compile-time (static type analysis) has two advantages. Type conflicts, i.e. using an operator on a value for which it is not defined, is the most common run-time error. Static type analysis therefore makes programs more robust. Also, more efficient code can be generated, since some of the run-time type checking overhead can be avoided. The static messages about type conflicts are especially useful for programs with many user defined types. Unfortunately, the demand graph construction algorithm as currently implemented cannot handle user defined types. Most of the mechanisms for demand propagation described in this section would, however, remain the same if user defined types were included. An earlier version of this application has been implemented by van Dijk&Veldkamp [Dijk83].

ASSERTION SPACE

We already mentioned in chapter 4 that, in general, the range of values of each particular variable occurrence cannot be determined precisely, but has to be approximated. The choice of assertion space determines to a large extent the accuracy of the approximation. The choice made in this application is that each assertion describes the range of values of a data item by *one* of the *value-domains* of figure 7.1. Note that this set of assertions does not allow the recording of the disjunction of two constants: if it has been determined that a particular arc will either carry an integer 7 or an integer 9, this has to be summarized by the value-domain *Positive Integer*. This entails loss of information, but such loss is essential to refrain from a complete (symbolic) execution of programs in general.

constants		
all values from the set of integers		
all values from the set of reals		
all values from the set of strings		
Undefined	True	False
approximations		
Positive Integer	Integer	Numeric
Positive Real	Real	Boolean
String	Defined	

Figure 7.1. Value-domains for the Value Approximation application.

The number of value-domains is infinite, since it includes the sets of integers, reals, and strings.

In addition to the value-domain, each assertion contains two more components. One boolean component (*is-an-array*) records whether the arc will carry an array value; if so the value-domain encodes information about the elements of the array. This

component is sufficient, since only one-dimensional arrays (first-order pointers) are allowed. A second component (*message*) provides space for a possible type conflict message.

In a forward application the assertions belonging to all incoming arcs of a node are identical. Therefore, assertions are associated with the node rather than with its incoming arcs.

ASSERTION LATTICE

The ordering of assertions is implied by the ordering of their components. Only the ordering of the value-domain component is non-trivial. This ordering is depicted by the tree in figure 7.2. Constants represent the most precise information and consequently the greatest (or strongest) assertions; the other value-domains are smaller (or weaker) approximations. A bottom element *Unknown* is added to make a meet semilattice. The meet operation, taking the greatest lower bound of two value-domains, is needed whenever a path in the demand graph diverges as in **BRANCH** and **PARAMETER** nodes.

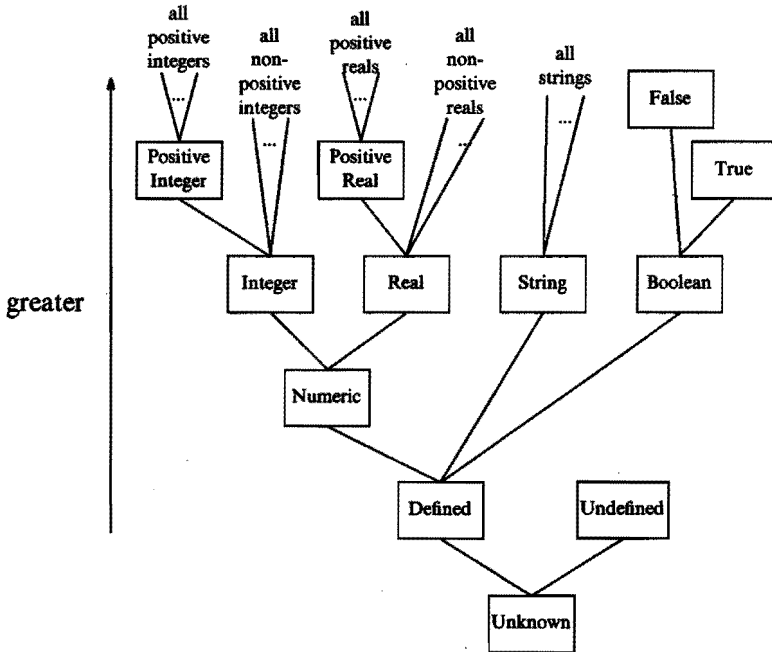


Figure 7.2. The semilattice of value-domains of the Value Approximation application.

Each box represents a possible value-domain. A phrase like "all positive reals" represents an unordered and infinite set of boxes. The semilattice is therefore infinite, but it is bounded since each of its chains is finite. The bottom of the lattice (*Unknown*) corresponds to the smallest assertion: it is consistent with all possible values. The *meet* of two value-domains is the greatest of their common ancestors in the tree.

INITIAL ASSERTIONS

CONSTANT nodes could simply be initialized with the most precise assertions and all other nodes with the bottom assertion. This would, however, complicate the reporting of type conflicts. Ideally each error should result in exactly one message. One way to prevent multiple messages originating from one error, is to selectively disable type checking in nodes that are dependent on a node that has detected a conflict. This could be implemented by adding a top element *Error*. This disabling may, however, prevent the detection of other errors. More errors can be detected, if a node that detects a type conflict leaves its assertion at its current value. This requires strong initial assertions as listed in figure 7.3.

node	initial value-domain	node	initial value-domain
CONSTANT	the particular constant	GET	String
CASE-CONSTANT	the particular constant	OVER	Integer
RELATIONAL-DYOP	Boolean	STRING	String
ARITHMETIC-DYOP	Numeric	REAL	Real
NEGATE	Numeric	INTEGER	Integer
CONCATENATE	String	TYPE	String
CASE-SELECTOR	Integer	ALWAYS	True
		NEVER	False

Figure 7.3. Value-domains of initial assertions.

Nodes not listed here receive the bottom assertion.

PROPAGATION RULES

If static type analysis is implemented as a forward application, it is simply a less precise form of constant propagation. Forward propagation of type information in an acyclic graph may provide exact type information except where the type of a value is dependent on conditional control flow. This occurs rather infrequently. In many languages an input expression may deliver an arbitrary type, but in SUMMER this is not a problem, since input is always a string.

Forward propagation rules are encoded in procedures *forward*, which are called by the propagation control subsection. Figure 7.4 gives a few representative examples.

ARITHMETIC-DYOP

```

new-assertion :=
    if both operands are constants
        folded constant
    else
        meet of assertions of two operands
if current-assertion <= new-assertion
    current-assertion := new-assertion
else
    set message "possible type conflict detected"

```

BRANCH

```

if control is constant
    current-assertion := assertion of particular branch
else
    current-assertion := meet of assertions of all value branches

```

PARAMETER

```

current-assertion := meet of assertions of all inputs

```

ARRAY

```

current-assertion := meet of assertions of initial-values
with is-an-array set to Yes

```

ARRAY-ACCESS

```

if this a retrieve
    current-assertion := assertion of previous-update
with is-an-array set to No
else
    current-assertion := meet of assertions of previous-update and source

```

Figure 7.4. Some forward propagation rules.

The test on a constant *control* operand in **BRANCH** nodes seems a bit excessive, but in combination with special propagation control for this node (see below) and with constant folding in **DYOP** it has the effect of providing conditional compilation without extending the language: in the expression

```
if compiler-switch = 1 then A fi
```

expression *A* is compiled conditionally. Van Dijk&Veldkamp have experimented with other features that make the language more convenient to use without changing its syntax or semantics. Subgraphs corresponding to expressions of the form

```
assert type(a) = 'integer'
```

are recognized and the information that *a* is of type *integer* is propagated to subsequent nodes. In this way the programmer could reap the benefits of strong typing in selected parts of his program.

PROPAGATION CONTROL

The demand graph is defined as all nodes reachable from the source-of-demands. Determining the demand graph requires a propagation of demands from the source-of-demands backwards. A node can receive forward flowing information only after it has been determined to be part of the demand graph.

Forward propagation in an acyclic graph can be implemented by a recursive descent traversal of a spanning tree of the demand graph. The initial demand is sent to the source-of-demands. The first time a node receives a demand, it addresses each operand in turn by sending it a demand and waiting for its reply. Replies contain the forward flowing information. After all replies have been received they are incorporated into the current assertion by procedure *forward*, and the node replies by sending its assertion to the node that issued the demand. The sink-of-demands has no operand so it can reply immediately thus initiating forward information propagation. If a node receives a second demand, it also replies immediately. The only exceptions are BRANCH nodes with a constant *control* operand; such a node propagates a demand only to the branch indicated by the *control* value.

7.2. Propagation in a Cyclic Graph

In a cyclic demand graph, some nodes receive a second demand before completing the processing of the previous demand. When the propagation control mechanism described in the previous section is used, a node that receives such a so called *cycling demand* simply replies with its current assertion. This guarantees termination, but the final assertions that it will produce in cycles are insufficient in strength. Figure 7.5 illustrates this point. The EXIT-LOOP node is the first node to receive a cycling demand. If it would simply reply with its current assertion (value-domain = *Unknown*) the application would not deduce that variable *a* is *Integer* and the PLUS node would report a possible type conflict.

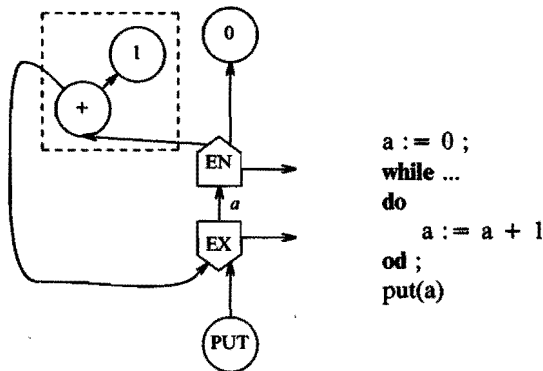


Figure 7.5. A cycle in the demand graph complicating type determination.

The type of variable *a* does not change in the loop. When demand propagation is started at the PUT node, the EXIT-LOOP node receives the first cycling demand. If it would reply with its current assertion the type of *a* would not be determined. Instead the EXIT-LOOP node propagates the cycling demand to the ENTRY-LOOP node, which replies with the hypothesis that the type of *a* remains *Integer* inside the loop. This hypothesis is propagated forward until it reaches the same ENTRY-LOOP node, where it is verified.

The application should make an effort to reach stronger assertions. The best approximation within the assertion lattice is not computable in general. Most practical cases are, however, quite simple: on most cycles the value of a variable changes, but its type is left intact. An application that produces the correct type information except for those variables that change type on a cycle, would therefore be sufficiently precise.

Whether the type of a variable is left intact on a cycle can be verified by induction on the number of iterations. For example, before the first iteration of the loop expression in figure 7.5 the type of variable *a* is *Integer*. If the type of *a* is *Integer* after *n* iterations, it will still be *Integer* after *n* + 1 iterations. The process used during demand propagation has a similar structure. A particular node on the cycle generates the proper hypothesis assertion. This assertion is propagated forward until it reaches the same node, which checks whether the propagated assertion corresponds to the hypothesis. The forward propagation of the hypothesis once around the cycle corresponds to the induction step. It is important to choose the proper hypothesis: a cycle usually leaves only those types intact that are acceptable to the operators on the cycle. The proper assertion can be derived from information outside the cycle, as we will show below.

To implement this strategy the propagation control must support the special handling of cycling demands without compromising termination. Most nodes react to a cycling demand as they do to their first demand: they propagate demands to their successors. The remaining nodes are *cycle breakers*; these are nodes that have outgoing arcs corresponding to alternative control flow: BRANCH, ENTRY-LOOP, and PARAMETER nodes. In programs without infinite recursion each cycle contains at least one cycle breaker.

ENTRY-LOOP nodes are cycle breakers for loops. When a cycling demand arrives at an ENTRY-LOOP node, it derives a hypothesis assertion from the assertion returned by the outgoing arc *entry* (i.e. the previous definition before the loop). A strong value-domain that is not expected to be preserved on the cycle (like *Small Integer* or a constant) is replaced by a weaker one. The hypothesis assertion is then replied to the demanding node. Eventually, the ENTRY-LOOP node receives a reply from its outgoing arc *last*. If the value-domain in this reply is still of the same type as the hypothesis, the cycle does not affect the type and the hypothesis is verified. If, however, the types differ, the current value-domain is set to *Unknown*. To ensure that the propagation rules preserve order, an extra component is added to each assertion to mark whether it is tentative because it is based on a hypothesis.

The scheme described so far provides the required type information for all cycles due to single loops free of conditional expressions. It works also for cycles that cover more than one iteration: the induction is not on the number of iterations but on the number of traversals through a cycle. It even works for some cycles that affect types. The expression in figure 7.6 illustrates both these properties.

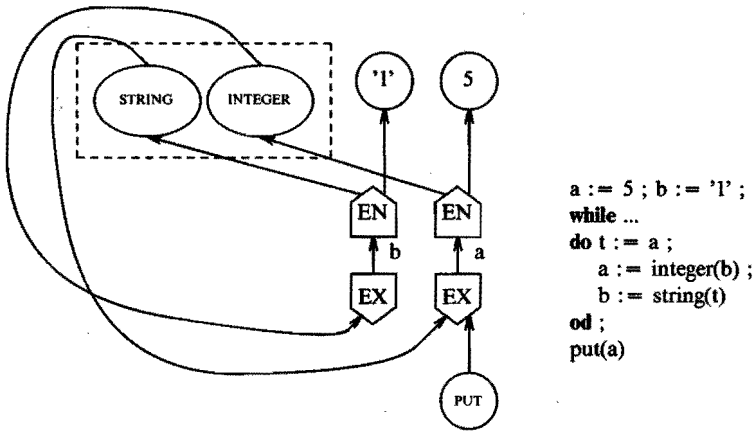


Figure 7.6. A cycle covering several iterations.

In this, somewhat contrived, expression the value of variable *a* inside the loop is dependent on its value two iterations earlier. The cycle therefore covers two ENTRY-LOOP/EXIT-LOOP pairs. A demand that enters at the rightmost EXIT-LOOP node propagates backward through the entire cycle and eventually reaches the rightmost ENTRY-LOOP node for the second time. This node returns the hypothesis assertion *Integer*, which is propagated forward through the cycle. The hypothesis has been transformed into *String* when it reaches the leftmost ENTRY-LOOP node and back into *Integer* when it reaches the rightmost ENTRY-LOOP node. The latter assertion confirms the hypothesis.

So far the first cycle breaker that is encountered (the ENTRY-LOOP node) is also a *cycle entry*, i.e. a node that has an outgoing arc to an initializing node outside the cycle. It is also clear which of the outgoing arcs leads to the initializing node so a demand can first be propagated along this arc and a reply received on which to base a hypothesis before a demand is propagated along the arc that is on the cycle.

Unfortunately, this scheme fails in case of cycles due to procedure calls. PARAMETER and BRANCH nodes, the cycle breakers for such cycles, cannot make any a priori assumption as to which of their outgoing arcs lead to initializing nodes and which may be on cycles. In fact, even for ENTRY-LOOP nodes the assumption that the *entry* arc is not on a cycle may be incorrect in case of nested loops.

A heuristic mechanism is implemented to identify cycle entries and their outgoing arcs that are not on a cycle. It enables a cycle breaker to delay the processing of a cycling demand, and consequently the construction of a hypothesis, until it has received information from outside the cycle. To achieve this, the issuing of demands and the processing of replies are separated, so that a node may propagate demands along all its outgoing arcs and then process replies in the order in which they are received. Demands and replies are handled by a central scheduler, which delays cycling demands to a cycle breaker until there are no more normal demands or replies left to be processed. The recursive program in figure 7.7 illustrates this mechanism.

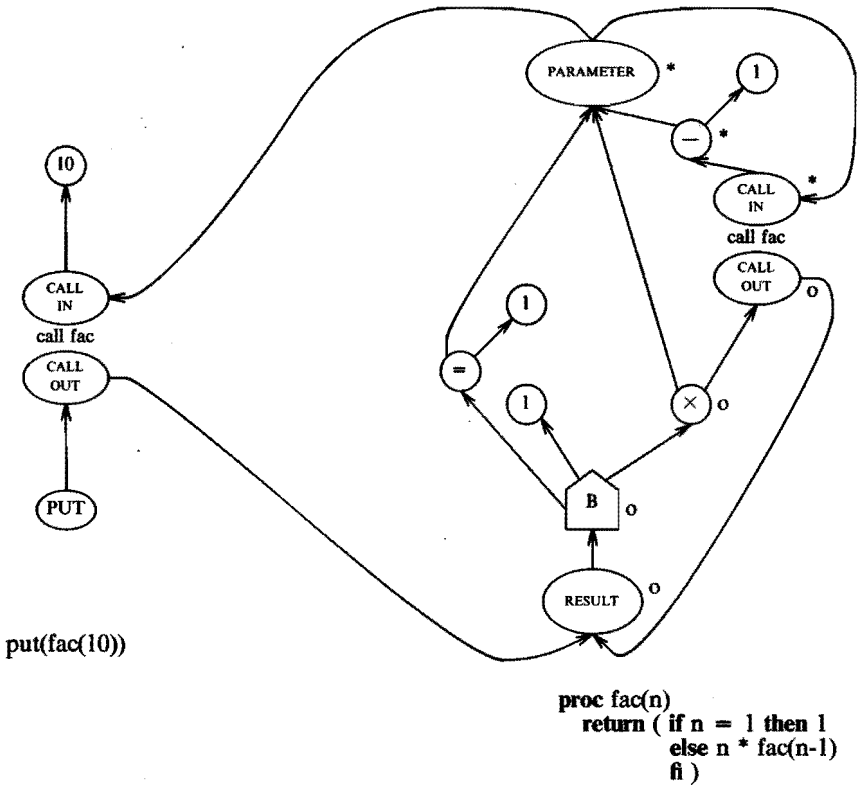


Figure 7.7. Propagation control in recursive cycles.

Two cycles are involved. One consists of the nodes marked with "*" and has the **PARAMETER** node as cycle breaker. The latter node delays the handling of a cycling demand until no normal demands or replies are pending. It will then have received information from the **CONSTANT** node 10 on which to base its hypothesis. The same holds for the cycle marked with "0" of which the **BRANCH** node is the cycle breaker.

Since cycles may contain more than one cycle breaker, not each of which is also a cycle entry, the scheduler only issues a delayed demand to a cycle breaker, if the arrival of forward information has confirmed that it is a cycle entry. If there is no such node among the cycle breakers for which demands have been delayed, no cycle entry has apparently been reached. In that case all delayed demands are propagated, in order to reach the next cycle breaker on the cycle. For programs without infinite recursion this process terminates, since in such programs each cycle has a cycle entry.

7.3. Backward Flowing Information

Before discussing the interaction between backward and forward information flow, we present a simple application, called *Static Allocation*, that needs only backward information flow. The purpose of this application is to attach to each arc either the assertion “The data item represented by this arc is accessible from outside the current procedure” or its negation. Since objects that are not accessible from outside the procedure can be allocated on the stack, the availability of such assertions can sharply reduce the allocation of objects on the heap and consequently garbage collection overhead.

An application like this consists mainly of use-definition analysis. Since this has already been performed during demand graph construction, the demand propagation phase is very simple. An object is accessible from outside the procedure, if the node that creates the object can be reached from a procedure interface along a path with only nodes that transmit objects (such as `BRANCH` nodes). Nodes that create a new object and `PUT` nodes use but do not transmit objects. The application therefore amounts to marking all nodes in the demand graph that can be reached from a procedure interface along a path without a `PUT` node or a node that creates an object.

Figure 7.8 summarizes the *Static Allocation* application. The assertion associated with each arc is stored in the node that is the tail of the arc; each node contains a number of outgoing assertions. An assertion consists of two boolean components: *Untouched/Touched* and *Local/Global*. The bottom assertion is (*Untouched, Local*). The propagation starts as usual by sending a demand to the source-of-demands. Each node processes a demand by calling procedure *backward*, which incorporates the backward flowing information accompanying the demand into its assertions. Demands are then propagated along all arcs of which the information has increased. Marking each assertion that is processed as *Touched* ensures that each node of the demand graph will be reached. Most nodes simply transmit incoming information along their outgoing arcs. Nodes that create an object, `PUT` nodes, and nodes at a procedure interface behave differently.

```

most nodes
  for each assertion
    set Touched
    if new information has Global set
      set Global

```

```

RESULT and PARAMETER
  set assertion(s) to (Touched, Global)

```

```

DYOP, NEGATE, STRING, INTEGER,
REAL, CONSTANT, TYPE, GET, and PUT
  set assertion(s) to (Touched, Local)

```

```

ARRAY-ACCESS
  set index assertion to (Touched, Local)

```

Figure 7.8. The backward procedures of the *Static Allocation* application.

The initial assertion for all nodes is the bottom assertion (*Untouched, Local*). Procedure interface nodes produce *Global* signals. Object creating nodes and `PUT` nodes set their outgoing assertion to *Local*. All other nodes propagate incoming *Global* signals to their descendants.

7.4. Bi-Directional Information Flow

The Value Approximation application can also benefit from backward flowing information: operators that accept only one or a few types can be used to derive information about an operand at those points where forward flowing information is insufficient. This is especially useful if the program contains many user defined types and operations on these types. Since the demand graph construction algorithm as currently implemented precludes user defined types we have to restrict ourselves to the standard operators. Figure 7.9 lists these requirements; they correspond directly to the initial backward assertions.

node	outgoing arc(s)	assertion	
		value-domain	is-an-array
NEGATE	<i>operand</i>	<i>Numeric</i>	
NOT	<i>operand</i>	<i>Boolean</i>	
ARITHMETIC-DYOP	both	<i>Numeric</i>	
OVER	both	<i>Integer</i>	
CONCATENATE	both	<i>String</i>	
BRANCH	<i>control</i>	<i>Boolean</i>	
MERGE	<i>control</i>	<i>Boolean</i>	
EXIT-LOOP	<i>control</i>	<i>Boolean</i>	
ENTRY-LOOP	<i>control</i>	<i>Boolean</i>	
ARRAY-ACCESS	<i>index</i>	<i>Integer</i>	
	<i>previous-update</i>		<i>Yes</i>

Figure 7.9. Backward flowing initial assertions.

Each backward assertion is associated with an operand arc. It specifies the type restrictions that the node places on its operands. All nodes or arcs not listed here receive the bottom assertion.

Backward flowing information is kept separate from forward flowing information: each node has an *operand* assertion for each of its outgoing arcs and one *value* assertion for all its incoming arcs combined. Each operand assertion contains an *Untouched/Touched* component to ensure that each node propagates demands at least once.

The interaction between the two directions of information flow may be complicated, since forward flow may induce backward flow and vice versa. The demand graph in figure 7.10 illustrates this. Let us assume that a demand arriving at the EQUAL node gets propagated to the CALL-OUT nodes, which, due to complicated conditions within procedure *f*, cannot determine the type of their results. A subsequent demand to the CONCATENATE node will propagate the assertion *String* backward to the CALL-OUT node, which is propagated forward to the EQUAL node. Since relational operators in SUMMER require their operands to be of the same type, it can be deduced that *b* should also be a *String*. This information can be propagated *backward* to the other CALL-OUT node where it produces a type conflict message.

```

a := f( ... );
b := f( ... );
x := 'abc' || a ;
if a = b then ...
y := b + 1

```

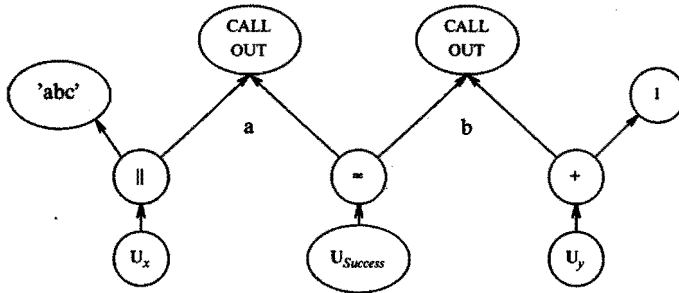


Figure 7.10. Interaction between backward and forward information flow.

Information flowing backward from the CONCATENATE node (marked with `||`) adds information at the CALL-OUT node. This gets propagated forward to the EQUAL node. Since both operands of a RELATIONAL-DYOP should be of the same type, information about *b* can be deduced. This is propagated backward to the second CALL-OUT node.

To implement the reversal from backward to forward flow (as in the CALL-OUT nodes in figure 7.10) each node maintains a list of predecessors from which it has already received a demand. Information is propagated to these predecessors when the value assertion increases. After replies have been received for all the demands that were propagated from one demand, they are incorporated into the current assertions by procedure *forward*. The value assertion is replied to the demanding node and, if the value assertion has increased, also to all other predecessors. If forward propagating information increases another operand assertion (as may happen in the EQUAL node in figure 7.10), a backward propagation along that arc is initiated.

References

Dijk83. DIJK, F. VAN AND A. VELDKAMP (May 1983). *Data Flow Analysis in SUMMER*, internal report, Centre for Mathematics and Computer Science, Amsterdam.

Chapter 8

Generating Dataflow Code

The major application of the demand graph method, and the one for which it was originally developed, is the generation of code for a dataflow machine. As already discussed in the introduction, the purpose of the translation is to test the hypothesis that an imperative language is a suitable programming language for a dataflow machine. The application described in this chapter translates SUMMER programs into graphs to be executed on the Manchester Dataflow Machine.

The dataflow graph to be generated is structurally similar to the demand graph: most of the demand graph nodes can be mapped onto one instruction in the dataflow graph¹ or to a small subgraph with the same number of input and output arcs. Most of the factors that make translating an imperative program into a dataflow graph problematic (jumps, aliasing, multiple assignment, global variables, see section 3.2) concern data-dependency analysis, and have already been dealt with during demand graph construction. A simple transformation from demand graph to dataflow graph is sufficient to obtain a correct translation. Moreover, the major part of this transformation, the mapping to the appropriate operation code and the generation of small subgraphs, can be relegated to the existing assembler by specifying an appropriate set of macros.

A suitable compiler, however, should produce code that is not only correct but also of high quality, at least comparable to that of code generated by compilers for other high level languages. High quality code is not only *efficient*, i.e. contains few overhead instructions, but also highly *parallel*.

1. To reduce the confusion between the two graphs we will use *instruction* rather than *node* when referring to the dataflow graph.

These quality requirements complicate the translation in several ways.

- SUMMER is dynamically typed, whereas the target language is strongly typed. Generating code that would perform dynamic type checking and conversion for every operator would produce an unacceptable overhead. Consequently, a static type analyzer, as described in the previous chapter, is necessary.
- The handling of arrays determines to a large extent the efficiency of the generated program. Since copying large arrays through interfaces is very costly, arrays are stored and pointers are circulated through the graph. Moreover, selective updates are made *in situ*: the array is not copied but the element is replaced in store. Care has been taken to reduce the serialization of accesses that this brings about. As an added benefit garbage detection is easily implemented.
- Parallelism can often be improved by an order of magnitude by implementing operations within loops in a parallel rather than a serial form. These loop optimizations require pattern recognition in the demand graph.
- Other subgraphs that need to be recognized are those that consist of several nodes but can be implemented by a single dataflow instruction.
- An efficient compiler should generate instructions with literals (a constant operand embedded in the operator). Fully exploiting this possibility sometimes requires constant propagation and bi-directional information exchange in the demand graph.
- The macro mechanism of the assembler is quite limited: it has no conditional construct and its parameter mechanism is restricted. Consequently, a subgraph of basic dataflow instructions often has to be produced directly by the code generator.

The first section of this chapter describes the target language. The next four sections treat language features roughly in the same order as followed in chapter 6. Section 8.6 treats loop optimizations.

8.1. The Target Language

The code generator produces assembly programs to be translated by the macro assembler provided by the Dataflow Research Group in Manchester. In this chapter we do not refer to this language directly, but use a graphical representation as illustrated in figure 8.1. An instruction is either *basic* or an application of a predefined *macro*. The functional behavior of an instruction is specified by the *operation code*, which determines the mapping from input to output values. The operation code determines also the number of input and output ports. Each output port may have an arbitrary number of output arcs.¹ An input port may be replaced by a constant input, called a *literal*, indicated as in figure 8.1(b). An output port without output arcs is illustrated as in figure 8.1(c).

1. In the machine language an instruction can have at most two output arcs, but this restriction is resolved by the assembler, which inserts extra DUP instructions.

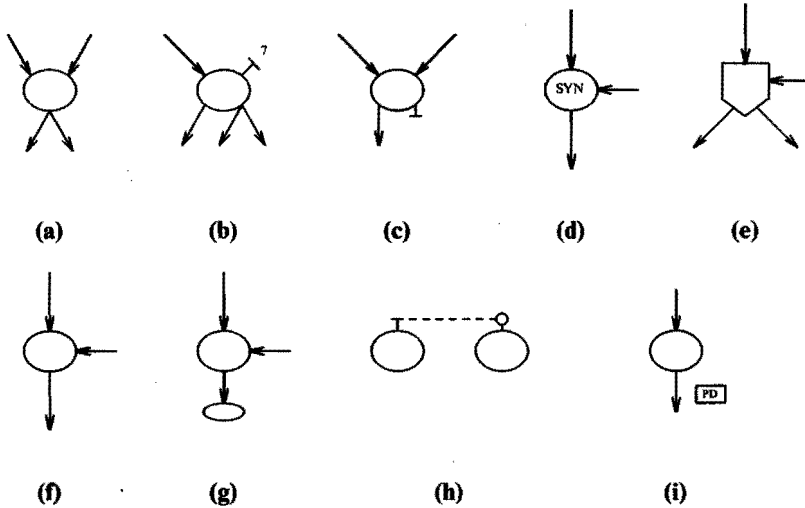


Figure 8.1. Notations used for instructions in figures.

- (a) Instruction with two input ports, one output port, and two output arcs.
 (b) Instruction with two distinct output ports and a literal as input.
 (c) Instruction with one unconnected output port. No token is produced on such an output port.
 (d) A use of the SYNCHRONIZE instruction where one input is used to trigger the release of the other input. In this special case the unconnected output port is not drawn.
 (e) Branching instruction; an instruction (possibly a macro instruction) whose output is sent either left or right. If the instruction has a control input it may be drawn on either side.
 (f) An instruction that accepts any type on its first input and a specific type on its second input. The specific input arc may be either drawn left or right.
 (g) Instruction with dynamic output arc.
 (h) On the left an instruction with a literal of type *destination*. The dashed line indicates that the literal refers to the input port of the instruction on the right.
 (i) An instruction producing tokens with the special matching function *Preserve-Defer*.

Each data item carries one of the type identifications listed in figure 8.2. The target language is strongly typed: many instructions are very particular about the type of their input tokens.

A	Activation Name	B	Boolean
C	Character	D	Destination
R	Real	G	Stream number
I	Integer	O	Ordinal
W	Context	X	Error

Figure 8.2. Some of the data types used in the Manchester Dataflow Machine.

A *destination* is a reference to an input port. A *context* is a combination of destination and (part of a) tag. A *stream number* identifies an input or output stream. An *ordinal* is an integer that is used as the value of an iteration level or index. A token of type *error* is created in all cases where the inputs fall outside the normal range.

A basic instruction has a three character operation code and its number of both input ports and output ports is limited to two. We divide the basic instructions into three groups: *operators*, *flow controllers*, and *tag manipulators*. The, sometimes highly specific, behavior for error tokens has been ignored in the following description.

The instruction set contains a great number of operators. Figure 8.3 lists all operators referred to in this chapter. The only two operators in this list that are not obvious are OST and RSR, which are used to convert between *Integer* and *Ordinal*. The RCK instruction is an example of a “micro-coded macro”: an instruction that is included for efficiency reasons to replace a simple subgraph of basic instructions.

Operators			
Op-code	range → domain	full name	description
ADI	$I \times I \rightarrow I$	ADD-INTEGERS	$i + j$
ADR	$R \times R \rightarrow R$	ADD-REALS	$x + y$
AND	$B \times B \rightarrow B$	AND-BOOLEANS	a and b
CEI	$I \times I \rightarrow B$	COMPARE-EQUAL-INTEGERS	$i = j$
CLI	$I \times I \rightarrow B$	COMPARE-LESS-OR-EQUAL	$i \leq j$
DRM	$I \times I \rightarrow I \times I$	DIVIDE-REMAINDER	$[i / j, i \bmod j]$
FLR	$R \rightarrow I$	FLOOR	convert real to integer
FLT	$I \rightarrow R$	FLOAT	convert integer to real
MLI	$I \times I \rightarrow I$	MULTIPLY-INTEGERS	$i \times j$
MLR	$R \times R \rightarrow R$	MULTIPLY-REALS	$x \times y$
NOT	$B \rightarrow B$	NOT-BOOLEAN	not a
ORB	$B \times B \rightarrow B$	OR-BOOLEANS	a or b
OST	$I \times I \rightarrow O$	OFFSET	integer subtraction and conversion to ordinal
RCK	$I \times I \rightarrow I$	RANGE-CHECK	error if left input is negative or greater than right input
RSR	$O \times I \rightarrow I$	RESTORE-ORDINAL	ordinal/integer addition and conversion to integer
SBI	$I \times I \rightarrow I$	SUBTRACT-INTEGERS	$i - j$

Figure 8.3. Some of the operator instructions provided by the Manchester Dataflow Machine.

The letters in *range* and *domain* indicate types as listed in figure 8.2. Note the two distinct output ports of the DRM instruction.

Figure 8.4 lists all flow control instructions referred to in this chapter. Most of these are not type specific. The BRW and BRR instructions are the basic branch instructions used in conditionals and loops; they differ only in the handling of error tokens. The SDS and SCD instructions are the main instructions with dynamic output arcs, used for procedure interfaces and storage of data structures. The BRT, SEP, SPL, TEX, and YZX instructions are useful for the handling of streams. The USE instruction is specially designed for efficient garbage collection.

Flow Controllers			
Op-code	range → domain	full name	description
BRT	$\forall \times \forall \rightarrow \forall \forall$	BRANCH-ON-TYPE	send left if inputs are of equal type else right
BRW	$\forall \times B \rightarrow \forall \forall$	BRANCH-WHILE	send left input to left or right
BRR	$\forall \times B \rightarrow \forall \forall$	BRANCH-REPEAT	send left input to left or right
DUP	$\forall \rightarrow \forall$	DUPLICATE	
IPT	$D \times G \rightarrow \forall$	INPUT	collect input from host stream and send to designated destination
OPT	$\forall \times G \rightarrow \forall$	OUTPUT	send left input to designated output host stream
SCD	$\forall \times W \rightarrow \forall$	SET-CONTEXT-DESTINATION	copy left input to destination with tag as specified by right input
SDS	$\forall \times D \rightarrow \forall$	SEND-TO-DESTINATION	copy left input to designated destination
SEP	$\forall \rightarrow \forall \forall$	SEPARATE-STREAM	decrement index of input and send left if index = 1 else right
SPL	$\forall \rightarrow \forall \forall$	SPLIT-STREAM-AND-HALVE-INDEX	divide index by 2 and send left or right depending on odd or even index
SYN	$\forall \times \forall \rightarrow \forall \times \forall$	SYNCHRONIZE	left input is copied to left and right input to right output
TEX	$\forall \times \forall \rightarrow \forall$	TEST-END-OF-STREAM-AND-INCREMENT-INDEX	if end-of-stream no output else copy input with index incremented
USE	$O \times A \rightarrow A -$	USE-COUNT	yield activation name if left input = 1
YZX	$\forall \rightarrow O -$	YIELD-INDEX-OF-EOS	yield index if input is end-of-stream and clear index field

Figure 8.4. Some of the instructions for flow control provided by the Manchester Dataflow Machine.

A "v" indicates any type. A "-" means no output. Branching instructions can be recognized by a "|" in the domain: output is sent to only one of the two output ports.

The function of most of the tag manipulators listed in figure 8.5 is obvious. The GAN instruction produces a unique activation name. No arithmetic is permitted on activation names. The use of the PRO and ENM instructions can give substantial efficiency improvement, since they can produce a whole series of tokens at once. The behavior of the PRP instruction is complicated; it is used for access to stored data structures.

Tag Manipulators			
Op-code	range → domain	full name	description
ADL	$\forall \times \bar{I} \rightarrow \forall$	ADD-TO-ITERATION-LEVEL	add integer to iteration level add integer to index produce series of copies of left input with increasing iteration level reserve new tag area produce series of copies of left input with increasing index combine destination with tag into context and set activation name transfer iteration level to index transfer index to iteration level swap activation name in value with that in tag
ADX	$\forall \times I \rightarrow \forall$	ADD-TO-INDEX	
ENM	$\forall \times O \rightarrow \forall$	ENUMERATE	
GAN	$\forall \rightarrow A$	GENERATE-ACTIVATION-NAME	
PRO	$\forall \times O \rightarrow \forall$	PROLIFERATE	
PRP	$A \times D \rightarrow W$	PREPARE-ACCESS	
SAN	$\forall \times A \rightarrow \forall$	SET-ACTIVATION-NAME	
SIL	$\forall \times O \rightarrow \forall$	SET-ITERATION-LEVEL	
SIX	$\forall \times O \rightarrow \forall$	SET-INDEX	
STL	$\forall \rightarrow \forall$		
STX	$\forall \rightarrow \forall$		
SWA	$A \rightarrow A$	SWAP-ACTIVATION-NAME	
YAN	$\forall \rightarrow A$	YIELD-ACTIVATION-NAME	
YIL	$\forall \rightarrow O$	YIELD-ITERATION-LEVEL	

Figure 8.5. Some of the instructions for manipulation of tags provided by the Manchester Dataflow Machine.

Frequently occurring subgraphs can be specified by means of macros. Each occurrence of the subgraph can then be replaced by a macro instruction. A macro application looks similar to a basic instruction, except that a macro can have an arbitrary number of input and output ports. We use macro names with more than three characters to distinguish them from basic instructions. Figure 8.6 shows an example.

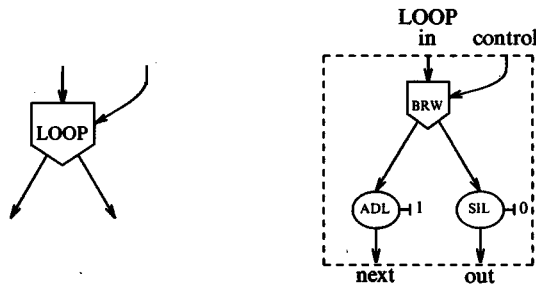


Figure 8.6. An example of a macro.

On the left an application and on the right the specification of a macro, called LOOP, to be used for simple loop interfacing (see also figure 2.10). The macro sends the token entering at arc *in* to the output arc *next* with its iteration level incremented as long as *control* indicates that the iteration should continue. Otherwise the token is sent to *out* with its iteration level cleared.

The translation from assembler to machine language is straightforward: after macro expansion each instruction is translated into one machine instruction, symbolic names are replaced by absolute addresses, and DUPLICATE instructions are inserted whenever needed to satisfy the packet constraint.

8.2. General Mechanisms

Except for loop optimizations, demand propagation uses only mechanisms described in the previous chapter. In this section we briefly review these basic mechanisms. Since the product of the translation is more interesting than its implementation, this chapter is less concerned with algorithms than the previous two chapters.

In the following description we occasionally refer to *cocooned expressions*. These are subgraphs that are completely surrounded by interface nodes all created by the same cocoon during demand graph construction.

ASSERTIONS

Each assertion has several components. The components for type analysis are as described in the previous chapter. Other components are for literal support, *in situ* update support, and loop optimizations. These are described in the appropriate sections.

PROPAGATION RULES

The application involves both backward and forward information propagation, but there is almost no interaction between the two. Type analysis is restricted to forward propagation. *In situ* update support uses backward flowing information. Only for literal support in dyadic operators forward flowing information may need to be directed backward again, as we shall see below.

Just as described in section 7.4, each node contains a value assertion for all incoming arcs combined and an operand assertion for each outgoing arc. The *get-demand* and *get-reply* procedures can be simpler than the ones presented in section 7.4, since they do not need to support the interaction between backward and forward flow. Only the *get-reply* procedure of dyadic operators needs to support a renewed backward propagation along one of the operand arcs, if warranted by information received from the operands. The type specific actions during demand propagation that we will encounter in the rest of this chapter are implemented by type specific versions of the procedures *backward* and *forward*. Since the algorithms are usually straightforward we do not treat these in detail.

PROPAGATION CONTROL

To accommodate type analysis in cycles a central scheduler handles demands and replies. The mechanism described in section 7.2 is used to delay cycling demands to cycle breakers.

EXTRACTION

During extraction nodes can be visited in any order, since the dataflow program is not order sensitive. For each node one or more lines of code may be generated, each of which describes one instruction in the dataflow graph. Only nodes that have received a demand generate code. At the end of the extraction phase code is generated that, when execution starts, will enter a single trigger token into the program. This token will be directed to the instruction generated by the sink-of-demands, which is the root of the dataflow program. Descendants of this instruction form the *trigger subgraph*, which will distribute trigger tokens to all instructions that need to be triggered: PROC-CALL, WHILE-LOOP, ARRAY, and CONSTANT instructions. A constant that is encoded as a literal needs no triggering. Care has been taken to minimize the trigger subgraph to avoid the generation of unnecessary trigger tokens.

8.3. Simple Operations

The translation of a straight-line segment is mostly straightforward: each operator corresponds to one node in the demand graph and to one instruction in the dataflow graph program. Complications are due to type mixing and strings. Taking advantage of efficiency improvements offered by literals requires extra analysis. The proper sequencing of I/O without reducing parallelism is also interesting. All four issues are treated in this section.

TYPE HANDLING

For a description of the static type analyzer see the previous chapter. Only forward propagation is employed. Cycles are handled by generating hypothesis assertions in cycle entries (see section 7.2). During extraction each operator checks whether the types of its operands are acceptable. If there exists a dataflow operator for which the operands have the required type, this instruction is generated. Otherwise conversion operators may be employed to coerce the operands into the required type. If this is not possible, an error message is issued. Figure 8.7 shows a few examples.

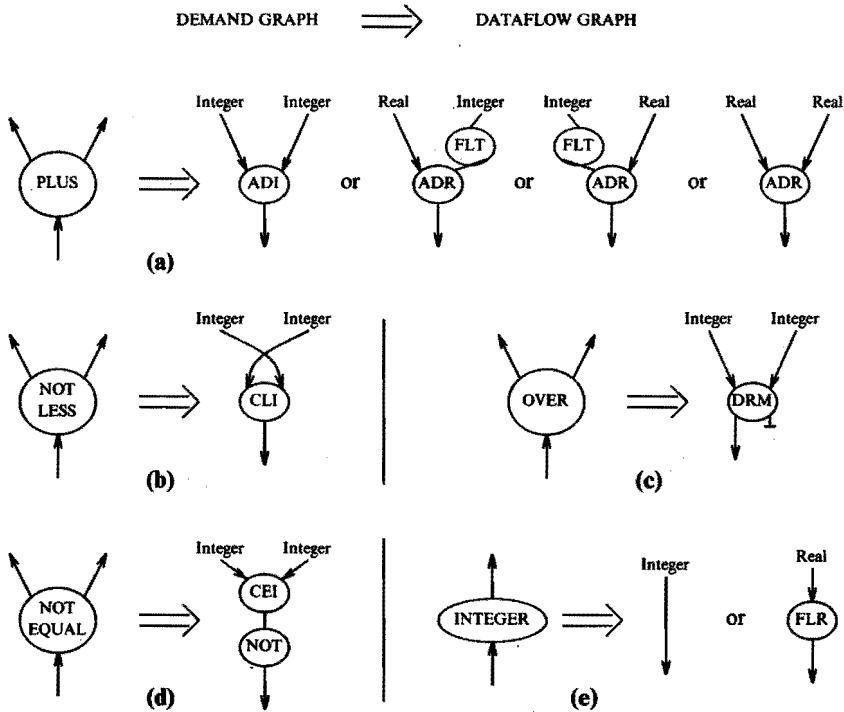


Figure 8.7. A few examples of type sensitive monadic or dyadic operators.

(a) An integer and a real addition are available in the target language. If the operands have different types a conversion instruction is inserted. Similar subgraphs are generated for MINUS, MULTIPLY, DIVIDE, and GREATER.

(b) Since no equality test for reals is defined, NOT-LESS accepts only integers. The `CEI` instruction is used with the operands interchanged.

(c) The integer division is implemented with a `DRM` operation without its *remainder* output.

(d) The implementation of NOT-EQUAL (only defined on integers) needs two instructions.

(e) Explicit conversion nodes do not generate code if the operand already has the required type.

STRINGS

Strings play a prominent role in most SUMMER programs. They can be of arbitrary length and cannot be encoded in a single token. They have to be represented as streams of tokens each carrying one character value. A stream is a series of tokens distinguished by consecutive values for the *index* field of the tag and terminated by an end-of-stream token. As with all streams, the implementor is faced with the difficult choice between *storing* and *copying*, as already discussed in section 2.3. When streams are stored in the matching unit (see figure 2.20), pointers can be passed around the graph. When copying is chosen, all tokens of the stream have to be copied whenever an interface is passed. Both approaches have their merits. Copying is simpler and, as long as streams are small (less than ten elements) and do not pass through many interfaces, more efficient. For most programs, however, copying would give a tremendous overhead. The advantages of storage will be even more pronounced when the structure store currently under construction has been installed.

Storing has been chosen for the implementation of arrays (see section 8.5) and copying for strings. Strings are copied, because the compiler is not expected to translate programs with much string processing. There are two reasons for this: the target machine is not very suited for string processing and the powerful pattern matching operations on strings that SUMMER provides are only defined for data structures for which demand graph construction has not been implemented.

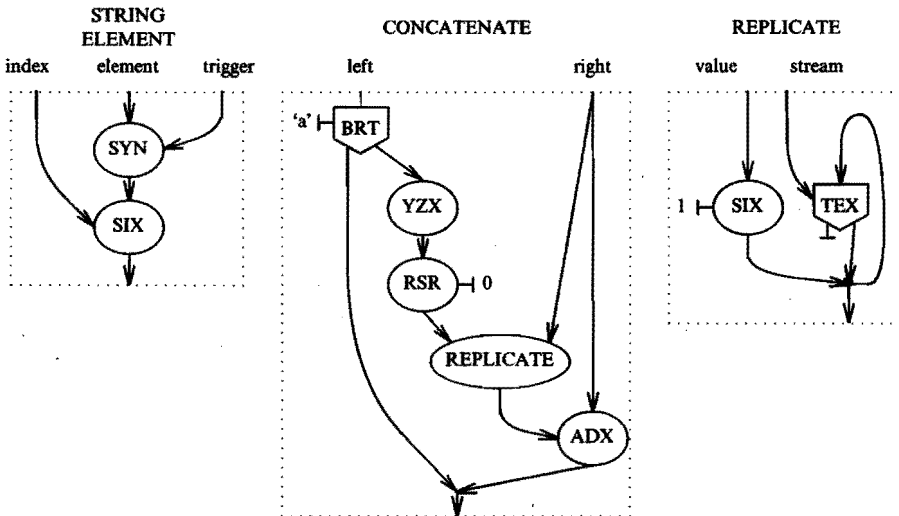


Figure 8.8. Macros for string handling.

(a) **STRING-ELEMENT** places a character token with the appropriate index in a stream. Since its first two inputs are literals it needs an input from the trigger subgraph to initiate its firing.

(b) Two strings are concatenated by merging the left stream without its end-of-stream token with the right stream with its index incremented. The **BRT** instruction sends the character tokens of the left stream to the left and its end-of-stream token to the right. The increment for the index is the size of the left stream and is deduced from the end-of-stream token. The **REPLICATE** macro makes a stream of increment tokens.

(c) The **REPLICATE** macro makes a stream of tokens with value equal to its left input. The stream is of the same size as the right input stream.

A `CONSTANT` node with a string value produces a `STRING-ELEMENT` instruction for each character of the string, plus one for the end-of-stream token. Each of these instructions produces one token with the proper index (see figure 8.8). Since they constitute one stream they all have the same target. The `STRING-ELEMENT` instruction needs a trigger input to initiate its firing. The outgoing arc of the `CONSTANT` node provides the appropriate connection with the trigger subgraph. A demand sent along this arc ensures that all nodes on the path to the sink-of-demands are marked as being demanded and will consequently generate the appropriate section of the trigger subgraph.

Operating on streams rather than on single tokens complicates the dataflow graph. The `CONCATENATE` macro in figure 8.8 illustrates this. The right stream passes through an `ADX` instruction to increment the index field. Just one increment token is not sufficient since it needs to be matched with every token of the stream. An application of the `REPLICATE` macro (copied from [Bowe81]) is therefore needed. This macro is used whenever a single value needs to be matched with a stream, such as when a string passes through an interface. The macros for comparing two strings have been omitted, because of their complexity; they each require a dozen basic instructions.

LITERALS

A constant can be represented by a `SYN` instruction that has the value of the constant as a literal and an arc from the trigger subgraph as input. In many cases a constant real or integer can be more efficiently represented by a literal embedded in its successor instruction. This does not only save the `SYN` instruction, but may also save part of the trigger subgraph: the interface instructions that distribute the trigger token to this cocooned expression can be omitted, if it does not contain any other instructions that need a trigger input. This optimization is especially effective in loops, since it may avoid the circulation of trigger tokens through all iterations.

Due to two restrictions of the target language an instruction may not be able to incorporate a literal. The first restriction is fundamental: no instruction can have only literal operands, since such an instruction would never become enabled. The second restriction is due to a peculiar limitation of the instruction memory: an instruction with two separate output ports has no space to store a literal. Unfortunately, the current assembler is not able to handle this low level detail.

Since it depends on its predecessor in the demand graph whether a `CONSTANT` node needs to generate a `SYN` instruction and send a demand along the outgoing arc, backward information propagation is required. Each node that does not accept literals indicates this to its operands by setting a particular component in its backward propagating assertions. A `CONSTANT` node representing a real or an integer does not propagate a demand into the trigger subgraph: until it receives such a demand. Otherwise it will communicate its value as a literal to the demanding node.

Dyadic operators are special since they accept a literal on either input arc (a frequently occurring case) but not on both. Their *backward* procedure initially communicates to both operands that literals are acceptable. If both operands return a literal a new demand is sent to one of the operands, this time specifying that a literal is not acceptable. A dyadic operator with two constant operands could of course be evaluated at compile time and folded into a new constant. This would, however, require a local restructuring of the graph. This situation was considered to occur too rarely to be worth the effort.

INPUT AND OUTPUT

Most of the analysis that is needed to support I/O has already been performed during demand graph construction. As explained in section 6.2, PUT and GET nodes are linked into one IO-subgraph with a STANDARD-IO node as sink. This IO-subgraph is translated into an equivalent subgraph of the generated program. The tokens flowing through this subgraph communicate to the input and output instructions sequence numbers to be used by the host processor to order the I/O items. This linking of I/O instructions does not limit parallelism, since the actual I/O and the calculation of the sequence numbers are performed asynchronously. The order in which the I/O actions are executed is therefore in general unpredictable.

The output and input primitives provided by the target machine are somewhat primitive; type handling of the IPT instruction is still to be defined. In the I/O macros in figure 8.9 it is assumed that only reals can be input, but that output can be of any type. In the SUMMER implementation all I/O is considered to be interactive (i.e. input and output are interrelated) and consequently to address the same stream. The STANDARD-IO instruction provides the initial value of the sequence number which is incremented in every GET or PUT instruction that is executed.

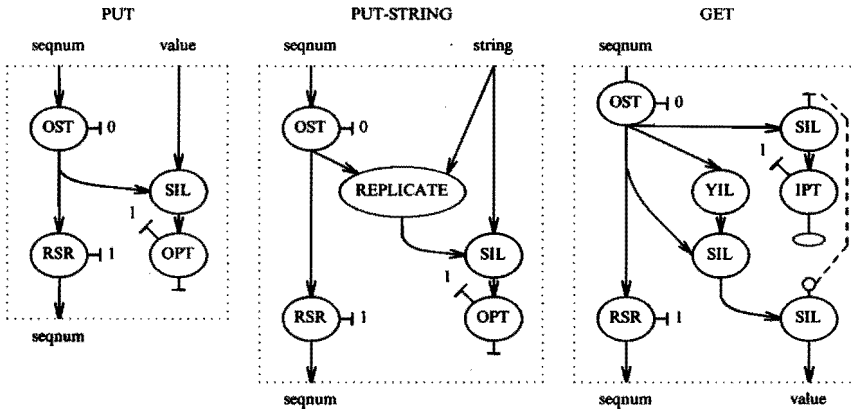


Figure 8.9. Macros for input and output.

(a) The sequence number is transmitted to the `opt` instruction through the *iteration level*, since the *index* field is already in use to distinguish characters in a string. Incrementing the sequence number requires two instructions due to the distinction between ordinals on which no arithmetic can be performed and integers which cannot be used to set tags. The new sequence number can be released before the output value has arrived.

(b) A string is output as a stream and counts as one item.

(c) The `ipt` instruction has a dynamic output arc; it requires an address token specifying to which input port the I/O token is to be sent. The dashed line indicates which input port is specified. For the correct handling of input instructions within a loop, the iteration level needs to be restored before the input value is sent to the rest of the graph.

8.4. Control Flow

Due to the analysis performed during demand graph construction, the handling of most control flow operators is quite simple. However, producing efficient rather than merely correct code requires a careful checking of special cases to detect opportunities for optimization. Fortunately, the basic instructions that are needed in the interface macros are not type-specific, so that type analysis is only needed when strings and arrays are involved. When a string passes through an interface, REPLICATE instructions

need to be inserted wherever a scalar interacts with the stream of character tokens (as shown in figure 8.9). Arrays are not treated in this section.

CONDITIONAL CONSTRUCTS

The translation of *if* and *case* expression, '&' and '|' operators, failure and other escapes amounts to generating the correct code for BRANCH, MERGE, and LINK-IN nodes. LINK-OUT nodes are transparent: they transmit every assertion unchanged and do not generate any code. We first treat the most general case, where the interface nodes are due to a *case* expression.

The translation of a *case* expression is illustrated in figure 8.10. Each input for each branch passes through a *gate*: a BRR instruction that either transmits the incoming token into the branch or discards it depending on its *control* input. These gates are generated by the LINK-IN nodes belonging to the MERGE nodes.

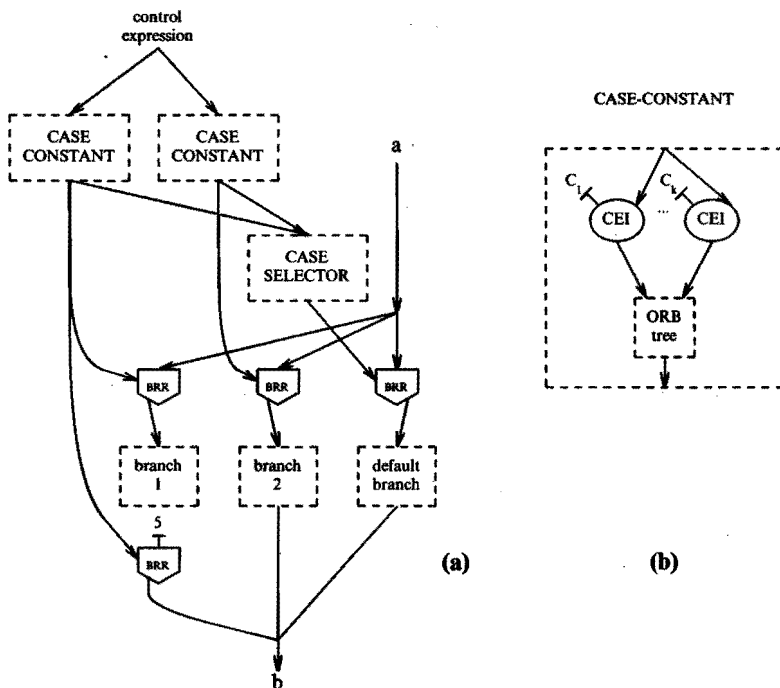


Figure 8.10. The translation of a case expression.

(a) The code generated for a *case* expression with 3 branches. The CASE-SELECTOR subgraph monitors the condition for the default branch; each CASE-CONSTANT subgraph monitors the condition for the other branches. Variable *a*, used in each branch, is passed through gates, each consisting of one BRR instruction. Variable *b* is defined in all branches; in the first branch it is assigned to a constant 5.

(b) The CASE-CONSTANT subgraph is not a macro, since the number of instructions it contains is variable. It contains a tree of ORB instructions which calculates the disjunction of all comparison signals. If there is only one constant the tree will be empty. The CASE-SELECTOR subgraph is a tree of ORB instructions that yields *false* if the default branch is to be executed.

The boolean tokens that control the gates are produced by a CASE-SELECTOR subgraph for the default branch and a series of CASE-CONSTANT subgraphs for the other branches. A CASE-CONSTANT subgraph compares the value of the control expression

with its case-constants and yields a boolean token indicating whether any of the comparisons succeeded. The CASE-SELECTOR subgraph yields the disjunction of the values produced by the CASE-CONSTANT subgraphs.

BRANCH nodes generate the output interface; usually a MERGE pseudo-instruction, which causes the assembler to direct tokens from the different branches to the same successor instruction. For branches that produce a literal, however, a gate is generated with the literal as value input. We shall see below that a BRANCH node may recognize special cases for which it can generate more efficient code.

For an if expression the controlling expression generates the appropriate boolean value directly, so the CASE-CONSTANT and CASE-SELECTOR subgraphs can be omitted. For if-then-else expressions a simple optimization often applies: if a MERGE node has exactly two LINK-IN nodes and both have been demanded, the MERGE node generates one combined BRR instruction instead of the two generated by the LINK-IN nodes.

OPTIMIZATIONS RECOGNIZED BY BRANCH NODES

Several circumstances may lead to BRANCH nodes in the demand graph. Sometimes better code can be generated if the situation that gave rise to the BRANCH node is recognized. This is the case for *compound comparative expressions*, i.e. a series of sub-expressions without side-effects and connected by '&' and '|' operators. Figure 6.7 in chapter 6 provided a simple example of this. In the code normally generated for this expression the evaluation of the sub-expressions would be serialized, because the standard implementation of a conditional is *lazy*: tokens enter one of the branches *after* the test has been evaluated.

The SUMMER code generator implements compound comparative expressions *eagerly*: the sub-expressions are evaluated in parallel. To support this eager implementation, each CONDITIONAL-COCOON records in each of its interface nodes whether any of its chainers has encountered a side-effect. If there have been no side-effects, the BRANCH node generates a boolean operator instruction instead of the code generated by the LINK-IN or MERGE nodes. Transforming AND and OR nodes into BRANCH and MERGE nodes during demand graph construction (see figure 6.8) and then back again into boolean instructions is a somewhat roundabout way to generate code. The advantage is that the same optimization applies to programs that are equivalent but are formulated with different operators, such as if expressions.

Another opportunity for optimization is provided by the way return escapes are handled during demand graph construction. This may lead to BRANCH nodes with constant boolean inputs, as for instance in figure 6.13. During extraction a BRANCH node therefore checks its value inputs; if both are boolean constants it generates code to either reproduce its *control* operand or produce its negation.

PROCEDURE INTERFACING

For procedure interfaces the standard solutions are adopted as illustrated in figure 8.11 (see e.g. [Gurd81]). A new activation name is generated, as soon as a trigger token enters a cocooned expression that contains a procedure call, making it certain that the call will be executed. This activation name is used in each CALL-OUT instruction to send an address to the corresponding RESULT instruction to specify where the result value should be sent. As soon as an actual parameter becomes available, the CALL-IN instruction passes it through the corresponding PARAMETER instruction to the procedure body. When a result value is produced, the RESULT instruction sends it through the appropriate CALL-OUT instruction to the calling environment.

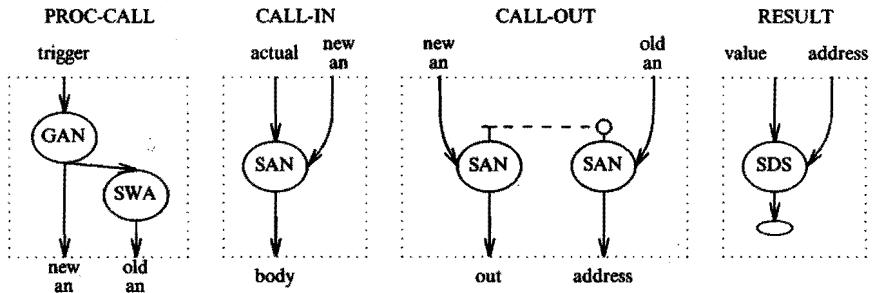


Figure 8.11. Macros for procedure interfacing.

Per procedure call there is one PROC-CALL instruction and a number of CALL-IN and CALL-OUT instructions and per procedure a number of RESULT instructions. PARAMETER nodes generate no code. The CALL-IN instruction sends an actual parameter with the new activation name into the procedure body. The CALL-OUT and RESULT instructions cooperate to send a result value back to the calling environment with the original activation name restored.

Note that the passing of parameters and result values are both fully asynchronous: a procedure body can start execution before all parameters are available. This is essential to attain high parallelism. A procedure containing a PUT instruction, for instance, could calculate its effect on the I/O sequence number independently of the calculation of the I/O values.

ITERATION

Figure 8.12 shows the macros involved in the translation of a while expression. The two targets of the EXIT-LOOP instruction are provided by the two LINK-IN nodes. If either of the two has not received a demand, the corresponding portion of the macro is omitted.

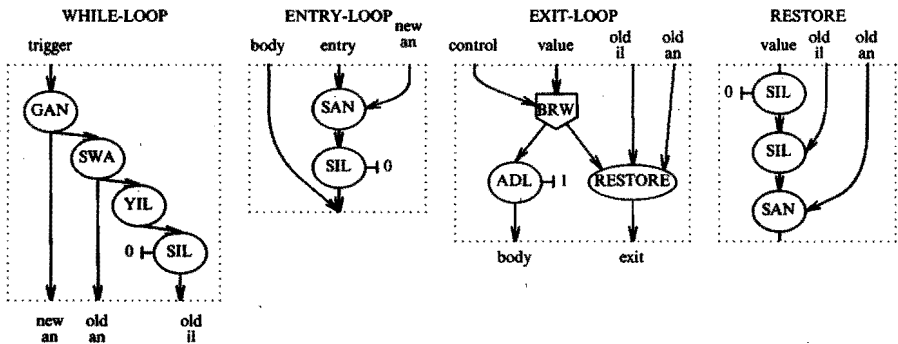


Figure 8.12. Macros for interfacing a while expression.

There is one WHILE-LOOP instruction per while expression, which generates a new activation name to create a new environment. This is necessary for a safe implementation of nested loops. In the ENTRY-LOOP macros this new activation name is attached to the incoming token and the iteration level initialized. The EXIT-LOOP instruction increments the iteration level and sends the token back into the body. When the controlling expression fails, the RESTORE instruction sends the result token to the successor expression, with the original iteration level and activation name restored.

Because this is a tagged machine, the passing of values through the interface instructions is asynchronous, just as with procedure calls. Each part of each iteration can therefore proceed concurrently with each part of any other iteration on which it is not data-dependent.

8.5. Arrays

The SUMMER code generator stores all arrays in the matching unit using special matching functions. The mechanism has been explained in section 2.5. The elements of an array are sent to one input port of a *storage* instruction, i.e. an instruction with a dynamic output arc. To retrieve an element a token is sent to the other input port of the storage instruction indicating to which instruction the element should be sent. By specifying a Preserve-Defer matching function a copy of the element is made and the element itself stays in the matching unit (see also figure 2.20). When all accesses to the array have completed, it needs to be removed from the matching unit by a garbage collection procedure.

The matching unit has no facilities for queuing requests; a request for an element that is not yet available is deferred by circulating the request tokens through the whole processing element. Extensive deferring causes a considerable waste. A retrieve of an element with the Preserve-Defer matching function should therefore not be attempted unless it is reasonably certain that the element is already present in the matching unit.

If it has been decided that arrays are stored, there is still a choice to be made regarding selective updates. In applicative languages a selective update operation, i.e. replacing one element in an array by a new value, is considered to produce a new array that is a copy of the old array with one element replaced. The old array remains accessible for retrieves. The corresponding implementation is the *copy update*, which creates a new array by copying the tokens of an array to a new storage instruction replacing one element by a new value. In an imperative language a selective update makes the old array inaccessible, so the obvious implementation is *in situ update*: the element is replaced within the stored array without making a copy.

These alternative implementations can be used for both types of language with similar trade-offs. Each method has its obvious drawbacks: copy-update gives considerable copying overhead, while *in situ* update limits parallelism. For programs with large arrays *in situ* update is the most attractive option, since reducing overhead is more of a problem than attaining parallelism for this kind of programs. It is the method adopted in the SUMMER code generator.

An *in situ* update changes the value of an array. Each value of an array is called an *instance*. An *in situ* update should only be executed after all accesses to the old instance have completed. A mechanism is therefore needed for *completion detection*. During analysis the number of retrieves that will occur for each instance is counted. Code is generated that, during execution, sends a signal to the next update when all retrieves of the previous instance have completed. If there is no next update the array has become garbage. So detection of garbage is a convenient by-product of completion detection.

Figure 8.13 illustrates this point. The ARRAY instruction stores the tokens in the matching unit and distributes a pointer to the RETRIEVE instructions and to a COUNT instruction. The latter has a literal input 2 indicating that it should wait for 2 completion signals from RETRIEVE instructions before passing the pointer on to the next update. The next instance has 3 retrieves. When they have all completed the COUNT instruction passes the pointer on to a GARBAGE instruction, which removes the stored tokens from the matching unit.

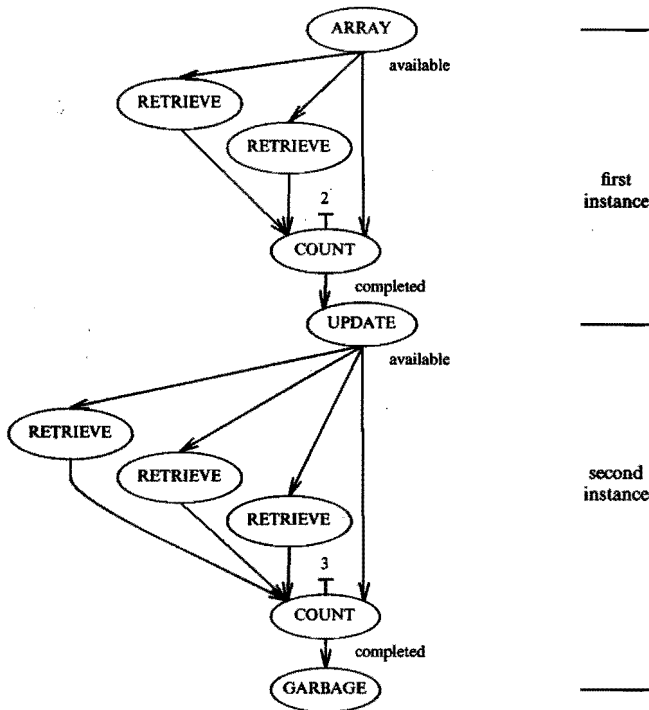


Figure 8.13. Serializing *in situ* updates.

Part of the code generated for a program with an array to which the only accesses are two retrieves, one update, and three retrieves, in this order. All instructions and arcs not involved with completion detection are omitted. The array is created by the `ARRAY` instruction and, when all accesses to it have completed, destroyed by the `GARBAGE` instruction. To prevent an update from replacing an element that still needs to be retrieved the accesses are serialized by completion signals issued by `RETRIEVE` instructions and collected by `COUNT` instructions. `ARRAY` and `UPDATE` instructions create new array instances; with each instance two signals are associated: *available* and *completed*.

Of course, when conditional control flow is involved the number of retrieves cannot be determined statically. Fortunately, each control flow decision is associated with a cocooned expression. A cocooned expression that contains retrieves can be counted as one retrieve, provided that the code within such an expression ensures that always exactly one completion signal is propagated to the surrounding expression. All retrieves for one array instance are independent and can be executed concurrently. By managing the completion signals within cocooned expressions carefully, this potential parallelism can be preserved.

MACROS

The macros for array handling are shown in figure 8.14. As originally proposed by Glauert, all arrays are stored on one shared storage instruction; the arrays are distinguished by activation name and the elements within each array are separated by the index field (see also [Sarg85]).

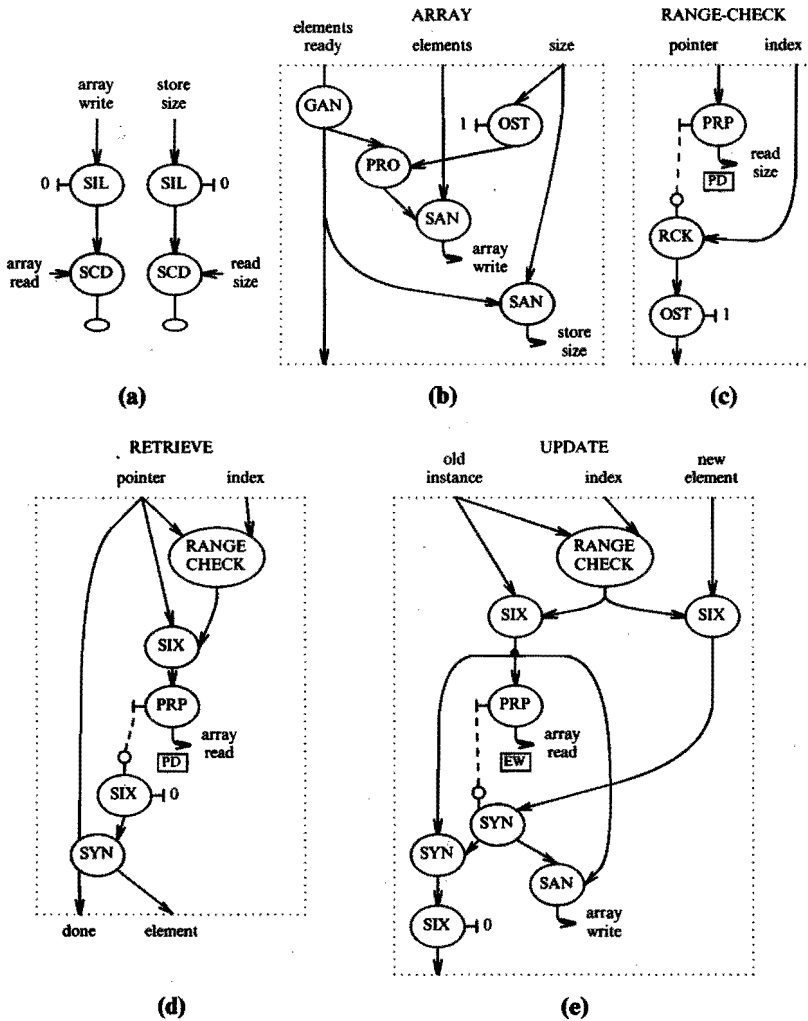


Figure 8.14. Macros for array storage and retrieval.

- (a) There are two storage instructions that are shared by the whole program. One is for the array elements; the other one for their sizes. An address token arriving at the *array-read* port or the *read-size* port causes the *scd* instruction to send the token to the address specified.
- (b) When the *ARRAY* instruction receives the signal that all elements are available, it generates a new activation name and sends the size with the new activation name to the shared storage instruction for sizes. The activation name is proliferated to match with all the elements of the new array, which are then sent to the other shared storage instruction.
- (c) Before an array is accessed by a *RETRIEVE* or *UPDATE* instruction the validity of the index is checked by comparing it with the stored size.
- (d) A *RETRIEVE* instruction reads the element from the storage instruction with the Preserve-Defer matching function so the array is not affected. When the element has been fetched the pointer is released to serve as completion signal.
- (e) An *in situ* update consists of a destructive read of the element followed by storage of the new element. The two actions are serialized (by means of a *syn* instruction) to avoid token clash. The pointer is not released until both the old instance and the new element are available.

When a new array is to be created, its elements are sent to the *elements* input port of an ARRAY instruction. SUMMER provides two ways of initializing an array. In the homogeneous case all elements have the same value and are produced by a trivial instruction not shown here. In the heterogeneous case the value of each element is produced by a separate expression. A COLLECT instruction (see e.g. [Bowe81]) produces a signal when all elements have arrived. The ARRAY instruction will then generate a new activation name, which serves as pointer to the array. This pointer is used in RETRIEVE instructions to fetch the proper element. In an UPDATE instruction the old element is removed from the storage instruction after which the new element is stored.

COMPLETION DETECTION

Most of the analysis that is needed to divide the lifetime of an array into instances has already been done during demand graph construction. Each array creation or update signals a new instance. We call nodes that represent such actions *instance headers*. In fact, we consider each node that is the destination of a *previous-update* arc an instance header: ARRAY nodes, ARRAY-ACCESS nodes that represent an update, and ENTRY-LOOP, LINK-IN, and BRANCH nodes that represent an array. Each instance needs a COUNT instruction, and the last instance of an array also needs a GARBAGE instruction (see figure 8.15). These are generated by the instance header. Two signals are associated with each instance, namely the *available* and the *completed* signal. All signals have the pointer to the array as their value. Each COUNT instruction has a *required* input indicating how many completion signals to collect for the instance.

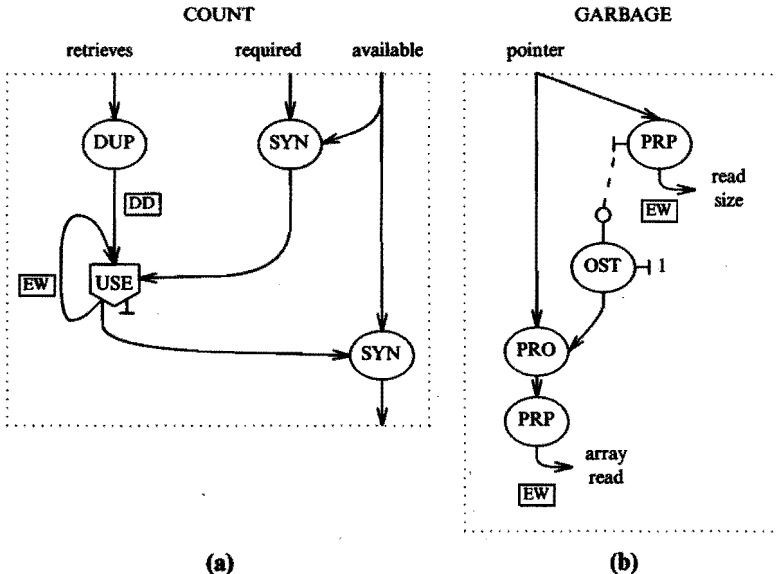


Figure 8.15. Macros involved in completion detection.

(a) The pointer arriving at the *available* input puts the initial *required* count into place. This indicates how many signals should arrive at the *retrieves* input before the *completed* signal is released. Counting is implemented by means of the Decrement-Defer matching function. The use instruction produces no output until the counter value has reached 0. The token is then released as the completion signal, as well as being circulated back into the use instruction with a normal Extract-Wait matching function to remove the stored token.

(b) Garbage collection is a destructive read using the PROLIFERATE instruction.

COUNT instructions with a *required* input value of less than 2 are omitted, since one of the incoming signals can be transmitted directly: if there are no retrieves, the *available* signal is used, and if there is only one retrieve, its completion signal is used. The value of the *required* input of the COUNT instruction is determined during demand propagation. Each assertion contains two boolean components, *Retrieve* and *Update*. ARRAY-ACCESS nodes are the source of the information: they set the two components according to whether they are a retrieve or an update. The two components are simply transmitted by most nodes, but interface nodes treat them differently.

Instance headers tally *retrieve* and *update parents*, i.e. parent nodes from which an assertion is received with either component set. During extraction the COUNT instruction is generated with the number of retrieve parents as *required* value. If no update parent has been encountered, the GARBAGE instruction is also generated.

Counting the number of retrieve parents ensures that all retrieves from within one cocooned expression are counted as only one retrieve in the surrounding expression, since each such retrieve assertion passes through the same interface node. Therefore, the instance header sees only one retrieve parent and consequently only one completion signal is expected from the cocooned expression.

A mechanism inside the cocooned expression has to collect and combine local completion signals. Figure 8.16 shows the code generated for a conditional without updates. Each branch has its own COUNT instruction that collects the completion signals within that branch. Each branch may also produce one completion signal, which counts as one retrieve in the surrounding expression. To prevent garbage collection from occurring more than once, garbage detection is done by the instance header in the outermost expression.

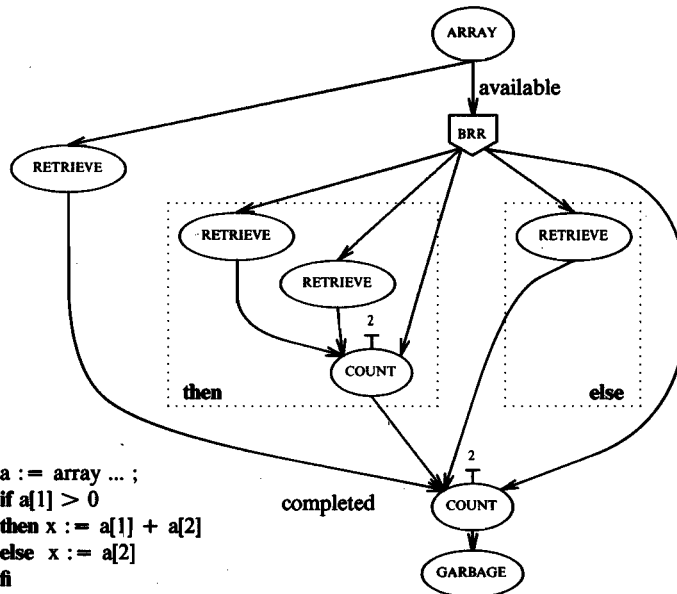


Figure 8.16. Conditional expression with retrieves but no update.

The RETRIEVE instructions in each branch receive the *available* signal through the BRR node. Note how all retrieves, inside and outside the conditional expression, may proceed concurrently.

LOOPS

Loops with updates are treated differently from those with only retrieves. In the latter it is important to treat the *available* and *completed* signals separately. If they are combined, a retrieve in one iteration cannot start until all retrieves of the previous iteration have completed. In the code depicted in 8.17 such serialization does not occur. The final signal that all retrieves have completed is sent to the surrounding expression.

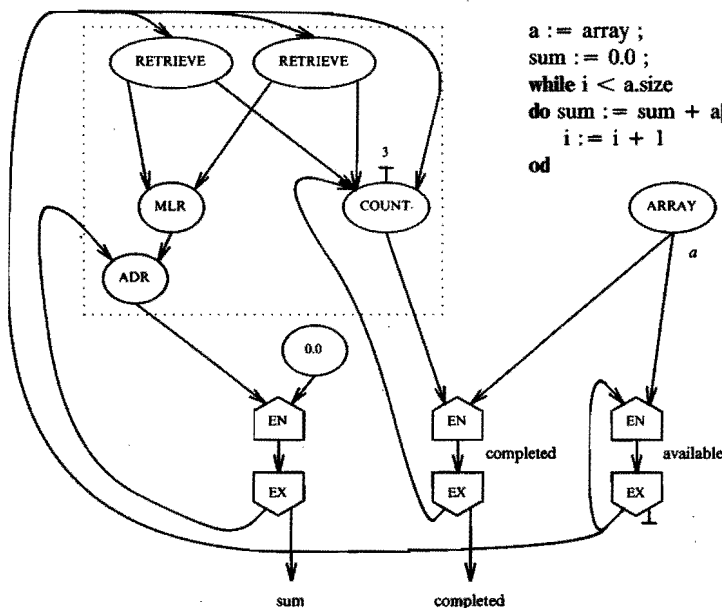


Figure 8.17. A loop with only retrieves.

A loop that computes the inner product of a vector with itself. The pair of interface instructions on the right provides the *available* signals. Because the two instructions form a cycle, the signals for subsequent iterations can be generated rapidly, without having to wait for completion of previous iterations. The retrieves of all iterations can therefore proceed concurrently. The cycle involving the pair of interface instructions in the center collect the completion signals of the retrieves and sends a signal to the surrounding expression, when they have all completed. The third cycle constitutes a *reduction operation*: it sums all values produced by the multiply instruction *MLR*. Nodes and arcs involved with the induction variable *i* have been omitted.

For a loop with updates (see figure 8.18) the iterations are already serialized. In this case efficiency has been preferred to parallelism and consequently the two signals are combined. This has consequences for the *Update* and *Retrieve* components transmitted by loop interface nodes. Fortunately, an interface node that receives a *Retrieve* assertion can easily check whether the loop contains updates for that array: if it does not, the *ENTRY-LOOP* and the *EXIT-LOOP* nodes form a tight cycle, i.e. they are only separated by a *LINK-OUT* and a *LINK-IN* node.

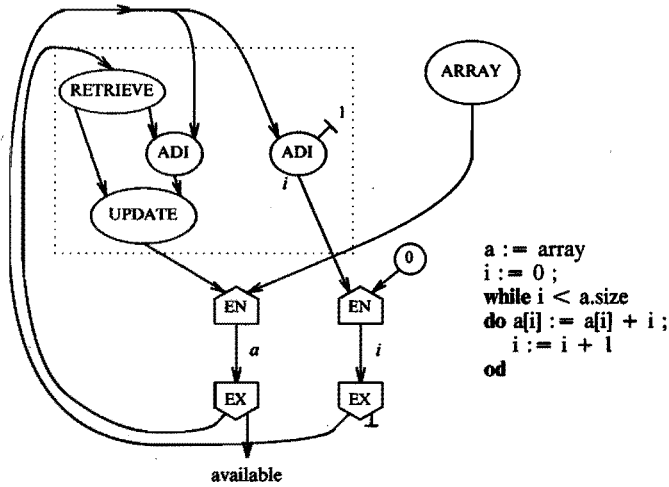


Figure 8.18. Loop with updates.

A loop with a complete redefinition of an array: all elements of the array receive a new value. The pair of interface instructions on the left carry the combined *Available* and *Completed* signal. Since there is only one retrieve per update, the COUNT instruction is omitted.

CONDITIONAL ALIASING

When the aliasing between two variables is unconditional, it has already been resolved during demand graph construction (see section 6.6). If the aliasing is conditional, however, code has to be generated to resolve the aliasing at execution time.

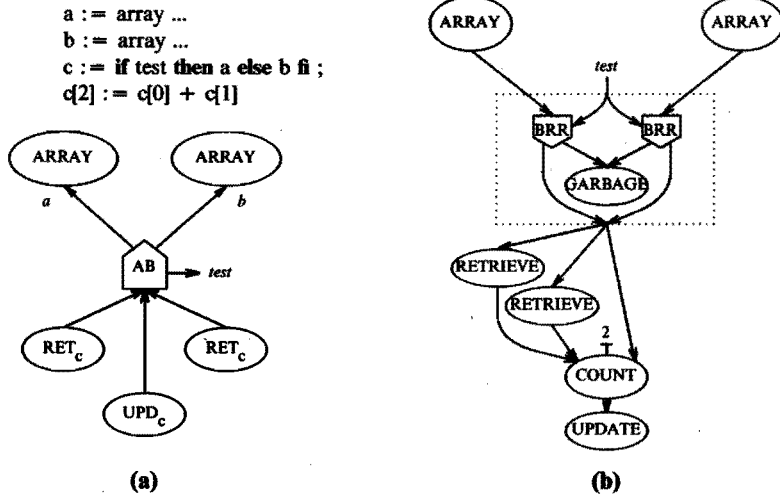


Figure 8.19. Simple conditional aliasing.

(a) A portion of the demand graph with an ACCESS-BRANCH node.

(b) The corresponding portion of the generated code. The pair of BRR instructions passes the appropriate pointer on to subsequent accesses. The pointer that is not selected belongs to an array that has become garbage.

The LACAP algorithm (see section 6.7) produces an ACCESS-BRANCH node for each ambiguity due to conditional aliasing. As far as code generation is concerned, an ACCESS-BRANCH node combines the functions of a BRANCH node and its corresponding MERGE nodes. The subsequent access is provided with the appropriate pointer by passing the alternative pointers through two complementary gates. Figure 8.19 illustrates this.

Garbage detection is concentrated in the code generated by the ACCESS-BRANCH node. In the example above, the array whose pointer is not sent to the subsequent retrieves has become garbage. However, sending a not selected pointer to a GARBAGE instruction is inappropriate if there are more ACCESS-BRANCH nodes pointing to the same instance header. Figure 8.20 gives an example of such a *shared instance header*.

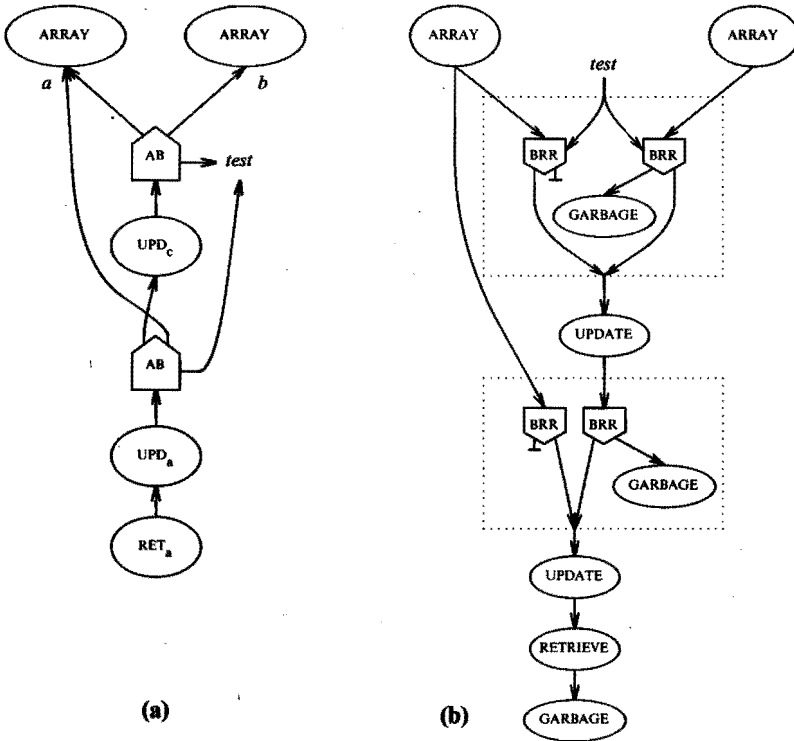


Figure 8.20. Complex conditional aliasing.

(a) Demand graph for a program segment with accesses through different aliases. The left-most $ARRAY$ node is a shared instance header: it has incoming arcs from two ACCESS-BRANCH nodes.

(b) The corresponding portion of the generated code. The BRR nodes corresponding to the shared instance header are complementary and consequently do not send the pointer to a garbage collector.

The LACAP algorithm only creates a shared instance header if it passes through a node in the alias graph twice. This occurs only if a *descend* in the algorithm is later followed by an *ascend*. The two ACCESS-BRANCH nodes that are created in these two phases are complementary with respect to the shared instance header. The code generated for one of these two nodes will pass the pointer on to subsequent instructions, where garbage detection will take place. Therefore, ACCESS-BRANCH nodes check the number of update

parents of their alternatives to see if they are shared instance headers. If one of them is, the corresponding BRR instruction does not send a token to a GARBAGE instruction.

8.6. Loop Optimizations

As the evaluation in the next chapter will show, the code generated for most language features is of reasonable quality. Improvements can, however, be made by recognizing special cases. The challenge is to identify those optimizations that are both easy to implement and yield substantial quality improvements. To stay within the scope of the current project only those optimizations that concern language features that differ significantly in SUMMER and SISAL have been explored, namely array handling and parallel loops. The two optimizations that yielded major efficiency improvements were those for loop constants and for a series of updates constituting a complete array update. Recognizing reduction operators may improve parallelism, but at the cost of somewhat lower efficiency.

8.6.1. PARALLEL DISTRIBUTION OF LOOP CONSTANTS

A loop constant is a value imported from outside a loop that is used in the loop but not redefined. It is represented by a series of tokens with identical values but consecutive iteration levels. The non-optimized code passes the loop constant repeatedly through an EXIT-LOOP instruction to increment its iteration level. Under certain conditions the same effect can be achieved by using an ENM instruction. Most instructions produce one or two tokens, but instructions like PRO and ENM can produce a great number of tokens in one burst, and may thus reduce execution time substantially. The situation is illustrated in figure 8.21. A loop constant can be recognized by the presence of a *tight cycle*, i.e. a cycle consisting of an EXIT-LOOP node and an ENTRY-LOOP node.

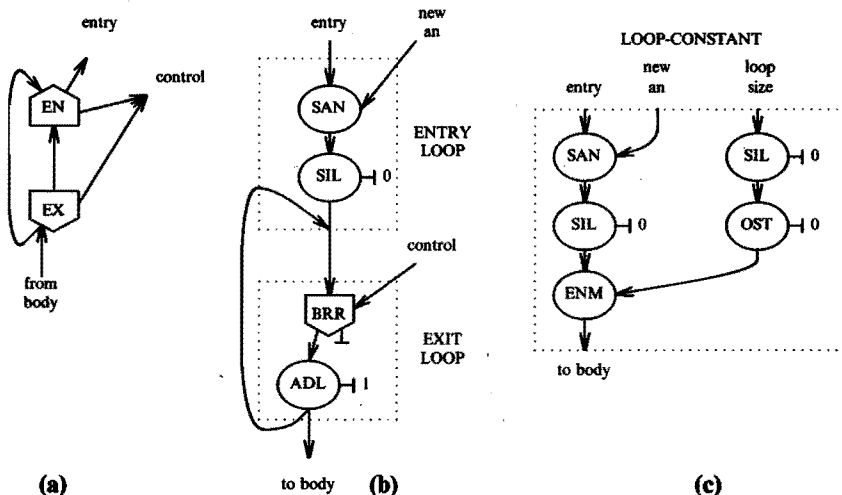


Figure 8.21. Parallel code for loop constants.

- A tight cycle indicates that a loop constant is imported into the loop.
- The code normally generated contains a cycle to increment the iteration level in a serial fashion.
- If the loop size is predetermined, the LOOP-CONSTANT instruction can be used, which generates the sequence of tokens in parallel by means of the ENM instruction.

The *loop-size* input to the LOOP-CONSTANT instruction indicates the number of iterations to be executed and determines how many tokens are produced. Obviously, this loop size cannot depend on any value computed within the loop. The LOOP-CONSTANT instruction can therefore be used only if the number of iterations of the loop is known, not necessarily at compile-time, but when execution of the loop is initiated. An obvious case is a **while** loop that is controlled by a RELATIONAL-DYOP node comparing a sequence of consecutive integers with a loop constant.¹ A predetermined loop size is recognized if this sequence of integers starts at 0 and is produced by a PLUS node that is on a *reduction cycle*, i.e. a tight cycle with one extra node.

The LOOP-CONSTANT instruction does not need input from the code generated by the control node. If that code has no other target, it is superfluous and should be suppressed: in a correct program each instruction has a target. The control node should not have been demanded, but on the other hand the optimization cannot be done before demands have propagated, since the recognition of the special cases depends on the propagated information. Because this problem is encountered in most optimizations, a general mechanism has been implemented to cancel a previously issued demand. Cancelling a demand causes the demanded node to remove the demanding node from its list of predecessors. If this makes the list empty, the node does not generate code, and in turn cancels any demands it has already issued to its operands.

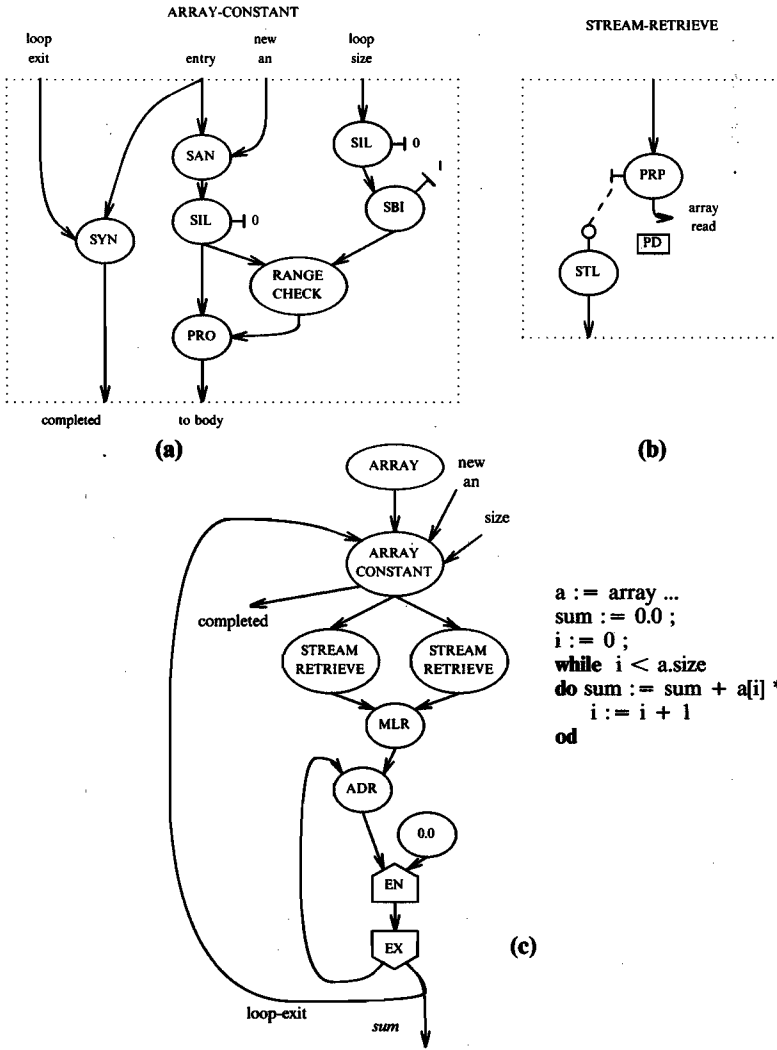
Loop Constant Arrays.

If a loop contains retrieves but no updates for a particular array, the array pointer is a loop constant. Figure 8.17 in the previous section provides an example. For a loop with a predetermined loop size this pointer is also distributed to all iterations in parallel by a LOOP-CONSTANT instruction. Further optimizations are possible if the index of the retrieve ranges from 0 to the loop size:

- The index does not have to be distributed separately, but can be derived from the iteration level.
- The range check and completion detection can be moved out of the loop.

The completion detection can be optimized, because in this case its only function is to generate the completion signal to the surrounding expression. Each (non-superfluous) retrieve within a loop contributes to the calculation of some output value from the loop. This value cannot be produced until all its contributing retrieves have completed. All completion detection code within the loop can therefore be replaced by a signal from the *loop-exit node*, i.e. the node that produces the particular output value. The ARRAY-CONSTANT and STREAM-RETRIEVE instructions, illustrated in figure 8.22, implement this optimization.

1. The optimizations have in fact also been implemented for the most common for loops. These somewhat simpler cases are ignored in this presentation.



```

a := array ...
sum := 0.0;
i := 0;
while i < a.size
do sum := sum + a[i] * a[i];
   i := i + 1
od
    
```

Figure 8.22. Macros for loop-constant arrays.

(a) For a loop-constant array the range check can be done just once. The completion detection is derived from output values of the loops that are dependent on the retrieves (one for each retrieve).

(b) A **STREAM-RETRIEVE** instruction is a **RETRIEVE** instruction from which the range check and completion part have been omitted. The **STL** instruction transfers the index field of the retrieved element to the iteration level.

(c) The inner product program of figure 8.17 optimized for loop-constant arrays. The elements are multiplied in parallel, but the summation is still performed in a sequential cycle. The final sum released from the loop is the loop-exit signal for the **ARRAY-CONSTANT** instruction, which triggers the completion signal for the array.

8.6.2. COMPLETE ARRAY UPDATE

As in most imperative languages, the only way to modify an array in SUMMER is by means of a selective update. *In situ* update is the most appropriate implementation for this, since for all but very small arrays its overhead is much less than that of copy update. The balance of this trade-off shifts, however, when a series of selective updates constitutes a *complete array update*, i.e. an operation that modifies each element of the array. In such a case the serialization overhead can be avoided by interpreting the series of updates as defining a new array that bears no relation to the old one. This optimization has been implemented for the case where the ARRAY-ACCESS node is on a reduction cycle, its index ranges over the whole array, and the loop and the array correspond in size. Figure 8.23 shows the COMPLETE-UPDATE macro and its effect on the generated program.

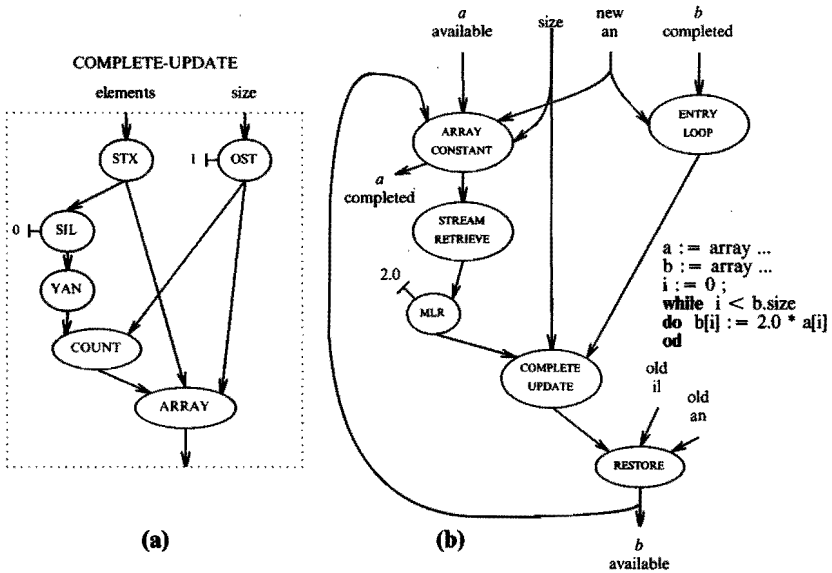


Figure 8.23. Complete update.

(a) A complete update is implemented by means of an **ARRAY** instruction. The values produced by the iterations are sent to the *elements* input port of the **ARRAY** instruction after their iteration level has been transferred to their index field. A **COUNT** instruction checks whether all elements are available. The old array can then be destroyed and the new array created.

(b) An application of the **COMPLETE-UPDATE** instruction. The pointer to the new array is the loop-exit signal for the **ARRAY-CONSTANT** instruction.

8.6.3. REDUCTION CYCLES

A dyadic operator on a reduction cycle produces a series of values each of which is based on the previous value. Two cases are recognized for optimization. In the first case the dyadic operator is a **PLUS** node and one of its operands is a constant integer 1 (e.g. as for variable *i* in figure 8.18). In this case a sequence of consecutive integers is to be produced and a macro instruction is generated that produces the series of tokens in parallel using the **ENM** instruction. This macro will not be shown.

The second case is when only the last value of the series is needed. The reduction cycle then amounts to a reduction operator, which often provides an opportunity for parallel code. This has been implemented for the equivalents of the **SISAL** primitives

sum and product. Bowen [Bowe81] devised a clever macro that performs reduction on streams of values in logarithmic time (provided there are an unlimited number of processors). The macro can best be understood by comparing it with the tree in figure 8.24(a). Note that the leaves of this tree have a token with odd sequence number as left input and the subsequent token as right input. If the nodes are numbered as in figure 8.24(b), node number k is connected to node $2k$ and $2k + 1$. The SPL instruction in figure 8.24(c) achieves the equivalent of these connections by manipulating the iteration level. The SEP instruction sends the output of the reduction operator back into the cycle except for the final result. To get the proper numbering, the first ADX instruction increases the iteration level of each incoming token by the loop size. Unfortunately, this requires a literal since it has to match with every incoming token. This macro can therefore only be used if the loop size can be determined at compile-time.

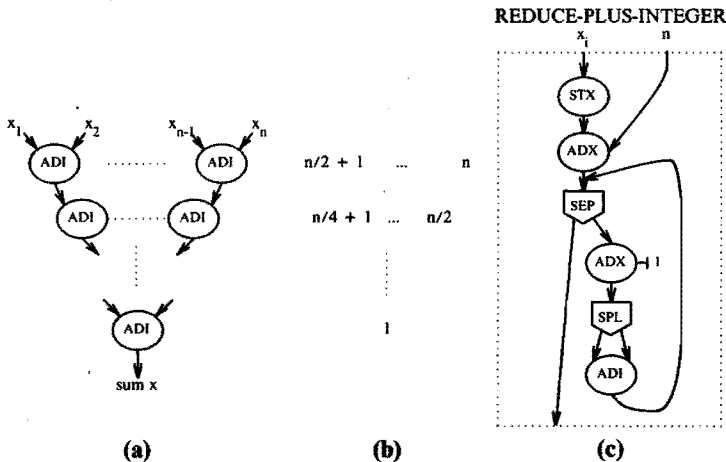


Figure 8.24. Reduction in logarithmic time.

- (a) n integers x_1, \dots, x_n can be summed in logarithmic time by a binary tree of ADI operators.
 (b) A numbering of the n operators.
 (c) All the ADI operators in (a) can be replaced by a single one, provided that the tokens originally belonging to different operators are distinguished by separate tags. The SPL instruction makes the equivalents of the connections in the tree by proper manipulation of the tags: it divides the index field by 2 and sends the token left or right depending on whether the original index was odd or even. The SEP instruction sends the final sum out of the cycle.

References

- Bowe81. BOWEN, D.L. (Apr 1981). *Implementation of Data Structures on a Data Flow Computer*, Ph.D. Thesis, Dept. of Computer Science - Victoria University of Manchester.
 Gurd81. GURD, J., J. GLAUERT, AND C.C. KIRKHAM (Jun 1981). Generation of Dataflow Graphical Object Code for the Lapse Programming language, *CONPAR81, Conference on Analysing Problem Classes and Programming for Parallel Computing*, 155-168.
 Sarg85. SARGEANT, J. (Apr 1985). *Efficient Stored Data Structures for Dataflow Computing*, Ph.D. Thesis, Dept. of Computer Science - Victoria University of Manchester.

Chapter 9

Evaluation

It is time to consider what the work reported in this thesis has taught us about dataflow machines and program analysis, and in particular about the suitability of imperative languages for the programming of dataflow machines. Recall from the introduction that the original goal of the compiler implementation was to verify the following two hypotheses:

- A translator from an imperative language into dataflow machine code produces code similar in *quality* to that generated from a dataflow language.
- Such a translator is similar in *complexity* to a conventional optimizing compiler.

The first two sections of this chapter present a quantitative appraisal of quality and complexity. The SUMMER language has not been completely implemented; we discuss the omissions and further extensions in the third section. The last section draws the conclusions.

9.1. Quality of the Generated Dataflow Code

Execution time is the obvious measure for the quality of a language implementation, i.e. the combination of a compiler with its target machine. However, an experimental machine such as the Manchester Dataflow Machine is frequently reconfigured and tuned and thus tends to be somewhat of a moving target. A better impression of the quality of the compiler itself is therefore gained by considering figures that express the consumption of key resources, i.e. those that are deemed to be crucial in any configuration.

Currently, the crucial resource of the machine is the matching unit: both its storage and its processing capacity form potential bottlenecks (see section 2.6). Two important metrics for quality are therefore the number of tokens stored in the matching unit at any one time (matching unit occupancy) and the total number of matching actions. As long as deferment is avoided, the number of matching actions per executed instruction does not vary much. The second metric can therefore be replaced by the total number of executed instructions. This is also a good indicator for the consumption of communication and processing resources.

The number of executed instructions becomes more informative if it is related to the computational complexity of the algorithm, independent of language, compiler, and machine. This gives an impression as to how much of the executed instructions are overhead, due to compiler or machine inefficiency. Unfortunately, an objective measure of the complexity of an algorithm (which we call its *algorithmic weight*) is hard to come by: which operations to count as an inherent part of the algorithm and which as necessary coding overhead is to some extent an arbitrary choice. For a certain class of numerical programs the number of floating point operations is a good choice for the algorithmic weight and consequently the fraction of the executed instructions that are floating point operations provides a useful measure.¹ We quote a variation of this measure, the *algorithmic content*, which is obtained by extending the algorithmic weight to include operations on data types other than reals.

The algorithmic content gives an indication of the *efficiency* of the generated program. As a measure of its *parallelism* we quote the ratio of the number of executed instructions and the length of the critical path, i.e. the longest chain of data dependent instructions. The Manchester dataflow group has determined that this ratio, called the *average parallelism* of a program, is a good predictor of how well a program exploits the parallelism of the machine. Since instruction storage is not a bottleneck, compactness of the generated code is not a primary quality metric.

Of course, these metrics are highly machine dependent and they should only be used to compare compilers for the same machine. The only other high level language compiler currently operative is the one for the SISAL language. This compiler is currently under revision to improve the quality of its code. This revised version, however, does not store arrays in the matching unit, but assumes that a structure store has been installed. It also uses an instruction set that is somewhat optimized for that compiler. It consequently generates code for a different machine and does not provide a fair comparison.

GATHERING THE STATISTICS

Six simple algorithms have been coded both in SISAL and in SUMMER: three programs to compare iteration, tail recursion, and double recursion, one program for simple string handling, and two programs that operate on arrays. The significance of the following comparison is limited by the simplicity of these programs. A more thorough evaluation requires the coding of large benchmarks in both languages.

The SISAL programs were translated with the most recent version of the compiler that stores arrays in the matching unit (called SISAL-MU) and the revised version that uses the structure store (called SISAL-SS). The SUMMER programs were translated with and without the optimizations described in section 8.6. We indicate the various optimizations with the following symbols: LC for loop-constants, CU for complete array update plus loop-constants, and RD for reduction operators plus the previous two optimizations. Optimizations that are not listed had no effect for the particular program.

Since the Manchester Dataflow Machine is not equipped to gather the required statistics, all programs were executed on a simulator, which simulates an idealized processing element with an unlimited number of functional elements and without communication delays. This has the advantage that the order in which enabled instructions are executed is fully determined: they are all executed in parallel. It

1. The Manchester dataflow group calls the inverse of this figure the "MIPS/MFLOPS" ratio.

further assumes that the execution times of all instructions are the same. It records the number of executed instructions, the length of the critical path, and the maximum occupancy of the matching memory. A figure for the algorithmic weight of the programs was chosen based on simple inspection. In the tables we list the following figures:

C = algorithmic content

Algorithmic weight divided by the number of executed instructions.

P = average parallelism

Number of executed instructions divided by the length of the critical path.

M = memory requirement

Maximum matching unit occupancy divided by the average parallelism.

C gives an impression of efficiency and P of the number of concurrent activities in the program. The product of these two figures gives an impression of the "real parallelism", i.e. parallelism without overhead computation. M indicates how much storage space has to be available to sustain one concurrent line of execution.

ITERATION AND RECURSION

The first programs sum the first hundred integers using iteration, single recursion, and double recursion.

iterative	recursive	double recursive
<pre>while i < 101 do sum := sum + i ; i := i + 1 od ;</pre>	<pre>proc sum(n) if n = 1 then 1 else sum(n-1) + n</pre>	<pre>proc sum(lower, upper) if lower = upper then lower else sum(lower,middle) + sum(middle+ 1,upper)</pre>

We take 100 as the algorithmic weight of all three programs.

The following figures were obtained:

	iterative			recursive			double recursive		
	C	P	M	C	P	M	C	P	M
SUMMER	0.10	1	4	0.05	2	168	0.02	40	14
SUMMER-RD	0.09	22	1						
SISAL-MU	0.06	6	54	0.06	2	112	0.02	49	3
SISAL-SS	0.44	1	193	0.07	2	134	0.02	47	7

If we look vertically we see that the SUMMER code does not compare badly with the code from the unrevised SISAL compiler. For loops it is somewhat better and for recursion slightly worse. The revision of the SISAL compiler has improved the efficiency of its code for loops enormously.

The unoptimized SUMMER code for the iterative program is completely sequential. The single recursive program has better parallelism, but this is due to extra overhead instructions. The lower efficiency indicates that procedure interfaces are twice as costly as iteration interfaces. This is because a new activation name has to be generated for each call and attached to parameter and result. The double recursive version gives considerable parallelism but at the cost of cutting efficiency (i.e. algorithmic content) in half. The optimization for reduction operators produces code for the iterative program that is better in all respects than the double recursive code.

STRING HANDLING

The next program produces the first hundred roman numbers separated by commas. It contains a procedure call within a loop, with each call producing one roman number. Each number requires several string concatenations and two procedure calls with string parameters.

	<i>P</i>	<i>M</i>
SUMMER	51	22
SISAL-MU	107	37
SISAL-SS	103	34

The algorithmic content has not been listed due to lack of an objective measure for the algorithmic weight. The number of executed instructions for the three generated programs is similar and quite high. A detailed inspection of the executed instructions for the SUMMER compiler reveals that almost half of them are DUP instructions and another 12 % due to the REPLICATE macro that is needed whenever a string crosses an interface. It is pleasant to see that, although the SUMMER program contains numerous sequential output statements, its parallelism is not affected. The SISAL compilers produce code that is twice as parallel.

ARRAY HANDLING

The following two programs operate on arrays: the first one computes the inner product of two arrays of 100 elements, while the second one multiplies two matrices of 10×10 elements. Matrices are stored as one-dimensional arrays, since the SUMMER compiler cannot handle more dimensions.

We take the number of floating point operations as the algorithmic weight. For the first program this is 200 and for the second one 2000.

	inner product			matrix multiply		
	<i>C</i>	<i>P</i>	<i>M</i>	<i>C</i>	<i>P</i>	<i>M</i>
SUMMER	0.03	9	27	0.02	58	13
SUMMER-LC	0.09	6	65	0.03	47	353
SUMMER-CU				0.03	104	157
SUMMER-RD	0.07	40	8	0.03	105	119
SISAL-MU	0.04	14	51	0.03	541	26
SISAL-SS	0.35	3	164	0.14	139	26

For the first program the optimizations for loop-constants improve efficiency considerably, due to the parallel distribution of the pointers for the array retrieves. The sequential addition, however, limits parallelism. The reduction optimizations remove this restraint. For the second program the effect of the loop-constant optimization is much less pronounced, since the necessary conditions for loop-constant array optimization are not fulfilled. The efficiency of the SUMMER and the SISAL-MU code is similar, whereas the revised SISAL compiler, which uses the structure store, produces much more efficient code. In these figures the accesses to the structure store have been ignored.

9.2. Complexity

An important property of a compiler is its computational complexity: the relation between compile time and the size of input programs. Determining this relation in general is too complicated, so we will follow usual practice and limit ourselves to the order of the asymptotic complexity, i.e. the limit of the computational complexity when program size goes to infinity.

The translation process consists of syntactic analysis, demand graph construction, and code generation. It is easy to see that syntactic analysis is of order n , where n is the size of the input program in some reasonable metric. Determining the complexity of the other two phases is more involved. We will estimate the computational complexity of an average case.

We claim without proof that the construction time of a demand graph node is bounded by a constant. The complexity of demand graph construction is then determined by the number of nodes. We distinguish three classes of nodes: interface nodes (`BRANCH`, `RESULT`, etc.), aliasing nodes (`ACCESS-BRANCH`), and the remaining nodes. The number of the latter nodes is of order n . We will show that the number of interface nodes is of order $n\sqrt{n}$ and the number of aliasing nodes of order $n\sqrt{n}$ or less.

The number of interface nodes depends on the distribution of cocooned expressions and of references to variables. For each cocooned expression there is one interface and each has a number of nodes equal to the number of exposed uses and definitions that occur in the expression or in any of the cocooned expressions that it contains, either directly or indirectly. Since we do not have statistics on the distribution of cocooned expressions, we have to make a rather broad assumption. We assume that the relative frequency of language constructs and their distribution over the program is independent of its size. It follows that the size of the average cocooned expression is constant. Consequently, the total number of cocooned expressions and hence the number of interfaces is of order n . Since the average number of variable references per cocooned expression is constant, the size of the average interface is proportional to the average (static) nesting depth of cocooned expressions. The latter corresponds to the average depth of a spanning tree of a graph, which according to [Flaj81], is of order \sqrt{n} , where n is the number of nodes. It follows that the total number of interface nodes is of order $n\sqrt{n}$.

The number of aliasing nodes is equal to the number of array accesses times the average number of nodes in the alias graph that the LACAP algorithm visits for one array access. The number of array accesses is of order n . The second factor depends on locality properties of the program. The LACAP algorithm has been developed under the assumption that, due to locality, the number of visited nodes per access is small and independent of program size. If this is the case the number of aliasing nodes is of order n . If the assumed locality does not materialize the average path covered by the LACAP algorithm is proportional to the depth of alias graphs, which is of order \sqrt{n} . So the number of aliasing nodes is of order $n\sqrt{n}$ or less.

Code generation requires a constant amount of time per generated instruction, since on the average one instruction is generated for each demand. The only exception is found in the handling of cycling demands for type analysis in cycles. The number of cycling demands per node is however constant due to the bounded number of input arcs of cycle headers. So code generation is proportional to the size of the generated program, which in turn is proportional to the number of nodes that are demanded during code generation. Many interface nodes are not demanded because they are not on a use-definition path. The relation between the size of the generated program and n

is determined by the average number of interfaces between a use and its corresponding definition. This in turn is proportional to the average length of a use-definition chain. If we call this L , it follows that the size of the generated program is of order $n \times L$. If references to variables are uniformly distributed, i.e. there is no locality of reference, L is proportional to nesting depth, i.e. of order \sqrt{n} . The size of the generated programs, and consequently the complexity of code generation, would then be of order $n\sqrt{n}$. Locality patterns may very well make L independent of n . The matter deserves further investigation, since it touches upon a central issue in the debate about applicative versus imperative languages.

In summary, the average case computational complexity of the complete translation is of order $n\sqrt{n}$.

The complexity of a program has another aspect related to the difficulty of writing, designing, and understanding the program. A rough indication is given by the length of the program, although this is sensitive to programming style and language. Three comparisons all indicate that the compiler is not excessively complicated:

- Considering the total translator for SUMMER to dataflow machine code, the program that constructs the demand graph and generates the code is smaller than the parser and the assembler combined.
- This total translator is about 50 % larger than the conventional SUMMER implementation, if the latter is restricted to the implemented subset.
- The SUMMER to dataflow translator is smaller than the SISAL implementation.

Design and implementation time of a program provide another, but even less precise, indicator of its complexity. The compiler took about 2 man-years to construct. More than half of this was spent on the design and implementation of the demand graph constructor. The translation to dataflow code is not significantly more complex than an ambitious optimizer such as one that performs static type analysis. The issues that were most time-consuming were the handling of conditional aliases and the correct interaction of escape signals with all other language elements. Other language elements that are often suspected of complicating the generation of parallel code, such as multiple assignments, global variables, and data structures, did not create any problems.

9.3. Extensions

There are two obvious ways in which the compiler could be extended: implementing the language features that have hitherto been omitted and improving the quality of the code by further optimizations. In the following subsection we discuss the omissions and suggest how they could be implemented. Suggestions for further optimizations conclude this section.

9.3.1. OMISSIONS

When the language features that have been omitted from the implementation are implemented three serious complications will arise: cyclic data structures, interprocedural aliasing, and overloading. We first discuss the implementation of the omitted language features ignoring these complications, and then present suggestions on how to attack the remaining complications.

Multi-dimensional arrays create conditional aliases of a new type: after the update "ar1[j] := ar2", "ar1[i]" and "ar2" may be aliases depending on the equality of i and j . Update nodes may thus become part of the alias graphs. The LACAP algorithm can be extended to handle these nodes in a fashion similar as that used for BRANCH nodes. The code generator will store a multi-dimensional array as an array of pointers. When

such an array is garbage collected, each sub-array has to be garbage collected as well, unless other pointers to it still exist.

When no overloading is involved, each user-defined data structure can be implemented as an array equal in size to the number of data fields. A selection of a data field amounts to a retrieve or update with the field name interpreted as a constant index. A procedural field selection amounts to a normal procedure call with the object itself (*self*) as extra input and output.

The remaining language features do not create problems. The *scan* and the *try* construct are straightforward. So are the *table* data type and the string operations, although an efficient implementation of these requires considerable effort in assembly programming.

Cyclic Data Structures.

Cyclic data structures complicate the aliasing problem, since they create cyclic alias graphs. The LACAP algorithm could possibly be extended to handle these, but cycles are notoriously difficult. A more fruitful approach may be to use a procedure, similar to that employed for recursion, to detect strongly connected components in the alias graph, and execute the original LACAP algorithm on the acyclic condensation of the alias graph. The problem is to restrict the search for strongly connected components sensibly, so as to avoid excessive analysis time. Cyclic data structures also complicate garbage collection, since completion detection as implemented amounts to static reference counting, which is not suitable when cycles are present. Reference counting on the acyclic condensation is, however, feasible.

Interprocedural Aliasing.

When interprocedural aliasing is allowed, the demand graph constructor has to assume that all data structure inputs to a procedure may be aliases of each other. The corresponding PARAMETER nodes become interior nodes of the alias graphs with all other PARAMETER nodes of the same procedure as descendants. During the analysis of the procedure body the LACAP algorithm may insert ambiguity nodes in the alias access graph for each PARAMETER node in the alias graph. When the demand graph constructor subsequently encounters a call of the procedure, the aliasing condition of each pair of actual parameters is available and is connected to the PARAMETER nodes. An attempt to resolve the ambiguity due to aliasing of parameters can be made during code generation. Most parameters will never be aliases; any ambiguity nodes that may have been created for these parameters can be ignored. Code is generated to resolve any remaining ambiguity at run-time.

Overloading.

When a field name is overloaded, i.e. it may refer to fields of different types, it is ambiguous, during demand graph construction, which data field is accessed or which procedure is being called. This ambiguity is encoded in a FIELD-SELECTION node that has the object as one of its descendants. As far as demand graph construction is concerned (global variables, failure, etc.) the FIELD-SELECTION node for a procedural field has the effect of the alternative procedures combined. Through static type determination this ambiguity may be resolved during demand propagation. For any unresolved ambiguity, code can be generated that selects the appropriate field during execution by means of the *types-fields matrix* generated by the parser.

9.3.2. FURTHER OPTIMIZATIONS

The loop optimizations described in section 8.6 should be generalized; in the current implementation the conditions under which the optimizations take effect are far too specific. The complete update optimization should be employed even if a few elements of the array remain unchanged. BRANCH nodes should take part in the optimizations to make efficient merging and concatenation of arrays possible.

The code generated for a loop interface is about twice as efficient as that for an equivalent tail-recursion. It is relatively easy to recognize tail-recursive calls in the demand graph and generate a more efficient interface that manipulates iteration level rather than activation name. This could be generalized to all directly recursive calls by using as increment to the iteration level the number of recursive calls in the procedure body (rather than 1). In retrospect, it would have been more consistent to create the same demand graph for loops as is created for recursion and to recognize during code generation the recursive calls that can be implemented iteratively. The loop optimizations would then have been equally effective for recursion.

Handling of data structures is a fruitful area for optimizations, but experiments with these are better postponed until after the code generator has been adjusted to take advantage of the structure store now being installed. Reports from Manchester indicate that the structure store may improve efficiency by 40 %. Even with the structure store, *in situ* update would be useful but leaves much room for improvement. Two updates of the same array are always serialized, but this is not necessary if they access two different elements. This may sometimes be determined at compile time by a closer inspection of the indices. Analysis of this sort could increase parallelism, although it would not directly improve efficiency. However, if this analysis is performed in (iterative or recursive) cycles, access patterns may be recognized that can be more efficiently implemented than with *in situ* update. Kuck and his colleagues have done much work in this area [Kuck81], some of which may be applicable. Their focus has been on generating parallel code for vector machines, for which an exact recognition is much more pressing, since such machines do not have asynchronous mechanisms to absorb the delays of minor maladjustments. Major improvements are, however, not expected from this type of analysis: preliminary investigations indicate that the extra instructions needed in the parallel macros often cancel any gain in efficiency.

9.4. Conclusions

PROGRAM ANALYSIS

A method has been described that transforms a program to be analyzed into a demand graph, a representation in which all control flow constructs have been replaced by data flow operators. Constructing a demand graph amounts to an extensive use-definition analysis that is different from the usual approach in that it retains all information that influences data-dependencies. Each branch in a data-dependency path represents a static ambiguity. These ambiguities are encapsulated in ambiguity nodes, an approach that has been very convenient, since the ambiguity can be ignored in most of the subsequent analysis without any loss of information. The only exceptions occur when the ambiguity concerns aliasing, which is the most characteristic feature of the imperative style. A naive solution of the aliasing problem would require an excessive number of nodes. A heuristic algorithm has been developed that exploits locality properties to reduce the complexity to manageable proportions for all but exceptional cases. This is an encouraging result, since it indicates that a worst case complexity argument does not need to deter one from searching for a heuristic solution that is

good in most cases.

The ambiguity nodes are created by the cocoon mechanism, which, although trivial in its original form, has proven to provide exactly the right abstraction. A cocoon is created wherever a conditional branch in the control flow is encountered during analysis. For each alternative control path a chainer is created to mimic memory. Since the cocoons not only register variables of the program, but also conditions and pointers used by the abstract evaluation function, they support the analysis of complicating features, such as escapes and aliasing, effectively.

Applications that require an extensive use-definition analysis benefit most from using the demand graph; some become trivial, as for instance the Static Allocation application described in section 7.2. A great advantage of the demand graph representation is that an operation is only connected to those operations that are relevant to it. Two sets of operations that are not dependent on each other are analyzed separately. One advantage of this is that it is often sufficient to propagate *local* assertions, which contain only information that is relevant locally. Most other analysis methods require *global* assertions, which cannot be both precise and of manageable size, since they contain information on the total state of the program. Another advantage of the separation is that a simple optimization technique is more often successful, since the exceptional cases for which the technique fails affects only the analysis of a small part of the program. In this respect the demand graph is similar to the Extended Data Flow Graphs proposed by Ferranti&Ottenstein [Ferr83]. The main difference is that the latter representation does not contain ambiguity nodes, but labels each operation with the predicate that most directly controls its execution. Ferranti&Ottenstein avoid cycles in the graph by marking loop controlling predicates differently and by excluding interprocedural analysis. In contrast to their representation, each non-trivial demand graph contains cycles, which may complicate the analysis considerably. The solutions found so far are complicated and *ad hoc* (see section 7.3); a more general and more elegant way of dealing with cycles is needed.

DATAFLOW PROGRAMMING

Dataflow computing is one of the most promising approaches to create a general purpose parallel computer that performs well on a wide variety of tasks, including those that do not exhibit regular and predictable parallelism. Dataflow machines are mainly programmed in so-called dataflow languages, which belong to the family of declarative languages. These languages have been developed because none of the existing languages was considered to be appropriate. Especially, it has been claimed that the imperative nature of these languages makes it difficult or even impossible to generate dataflow code with sufficient parallelism. Two arguments are usually put forward in support of this conclusion.

The first argument is that imperative programs obscure their parallelism by constructs that are based on sequential execution and that removing superfluous sequencing constraints is not a practical option. The work reported in this thesis shows that this is not a valid argument. Chapter 8 described the code generator of a compiler that translates a subset of the imperative language SUMMER into dataflow code. The first two sections of this chapter show that this compiler produces code of similar quality as a compiler for a dataflow language and that it is of similar complexity as a conventional SUMMER implementation with static type checking. These two results lend strong support to the hypotheses mentioned in the introduction. The translation of other imperative languages into dataflow graphs is not expected to uncover fundamentally new issues. Languages that rely strongly on pointer operations require

an efficient handling of aliases, which can be modelled on the algorithm developed for the SUMMER implementation. Unrestricted jumps should be distinguished by direction: a forward jump can be treated as an escape and a backward jump as a loop.

The second argument, in favor of using declarative languages for dataflow machines, is that they lead the programmer to avoid algorithms that are hard to execute in parallel. Unfortunately no evidence for this argument has been offered so far. If this influence on programming style can indeed be demonstrated, it remains an interesting question to which of the differences between imperative and declarative languages it should be attributed. Most of the constructs in a dataflow language are easily coded in an equivalent imperative form that can be recognized by the compiler (see e.g. the optimizations described in section 8.6). Therefore, the interesting differences amount to constructs that are absent in declarative languages. The comparison of declarative and imperative languages in chapter 3 lead to the conclusion that the main advantage of using declarative languages for parallel processing is that they require the programmer to specify the interface of each expression (its input and outputs) explicitly. The discussion on asymptotic complexity on the previous pages indicates that, without locality, these interfaces grow with the square root of the program size. This soon becomes bothersome and the programmer will tend to avoid large interfaces by writing programs with more locality. On a parallel machine locality is often conducive to efficient execution. If, however, data structures can be manipulated as easily as scalars, the specification of a large interface can be abbreviated to one reference to a data structure. Some data structure operations may therefore remove the incentive for locality and consequently jeopardize the advantage of declarative languages for parallel processing. So it may not be so much the declarative nature of a programming language that makes it attractive for parallel processing, but the absence of certain operations on data structures.

A FUNCTIONAL PERSPECTIVE ON IMPERATIVE PROGRAMS

The interpretation of an imperative program usually relies on a model that manipulates a computational state. Such a computational model is easily mapped onto traditional uni-processors, but it is only one of the models on which program interpretation can be based. The computational state is not a convenient concept for either parallel processing or for reasoning about program analysis. For these purposes a functional interpretation, i.e. a specification of the relation between input and output, is more convenient. In this interpretation program fragments specify what is to be accomplished rather than how: the term 'a + b' specifies a sum rather than an addition. A functional interpretation of a program fragment includes a specification of its input and output. In imperative programs these are not always explicit and may have to be uncovered. The lesson learned from this project is that this does not require a complicated analysis, except when a significant amount of aliasing is involved. The functional interpretation initially requires a change of perspective, but is eventually just as natural as the interpretation by means of the computational state. The sequential nature of an imperative program is in the eye of the beholder. The semicolon symbol, often construed as specifying the sequential execution of two expressions, may just as well be interpreted as specifying functional composition. There is no need for parallel processing to exclude the misconstrued semicolon.

References

- Ferr83. FERRANTE, J. AND K.J. OTTENSTEIN (Jan 1983). A Program Form Based on Data Dependency in Predicate Regions, *Tenth Annual Symposium on Principles of Programming Languages*, 217-236.
- Flaj81. FLAJOLET, P. AND A. ODLYZKO (Feb 1981). *The Average Height of Binary Trees and Other Simple Trees*, *Rapports de Recherche* 56, INRIA - Rocquencourt.
- Kuck81. KUCK, D.J., R.H. KUHN, D.A. PADUA, B. LEASURE, AND M. WOLFE (Jan 1981). Dependence Graphs and Compiler Optimizations, *Eighth Annual Symposium on Principles of Programming Languages*, 207-218.

Appendix I

From Program to Parse Tree

This appendix specifies the grammar of the subset of SUMMER that is accepted by the demand graph constructor and indicates the mapping from program to nodes in the parse tree.

The grammar is given in the BNF-like notation used in [Klin82]. The symbol '[' indicates alternatives, '*' zero or more repetitions and '+' one or more repetitions. Optional grammar symbols are enclosed between '[' and ']'. The sequence '{ a b }+' is equivalent to 'a (b a)*'. Each reserved word in SUMMER is printed **bold**; literal symbols are within quotes.

The \rightarrow symbol indicates the translation to parse tree. The name of a node type is given in CAPITALS. If a series of nodes of the same type may be generated, the node name is followed by the symbol '*' or '+'. A node name may be followed by the names of interesting output arcs: each output arc corresponds to a non-terminal in the preceding production rule. An arc name followed by a '*' indicates a list of output arcs.

```
<summer-program> ::=
  ( <global-variable-declaration> | <procedure-declaration> )*
```

```
<global-variable-declaration> ::=
  var { <identifier> ';' } + ';' 
```

```
<procedure-declaration> ::=
  (proc | program) <identifier> <formals> [ <expression> ] ';'
   $\rightarrow$  PROC-DECL(name, formals, body)
```

```
<formals> ::= '( { <identifier> ';' } * )'
   $\rightarrow$  PARAMETER*
```

```
<expression> ::=
  <monadic-expression> | <dyadic-expression> | <primary>
```

`<monadic-expression> ::=`
`<monadic-operator> <expression>`
`| <monadic-function> '(' <expression> ')'`
`→ MONOP(operand)`

`<monadic-operator> ::= '-' | '~' | assert`
`→ NEGATE | NOT | ASSERT`

`<monadic-function> ::= return | type | stop | string | integer | real`
`→ RETURN | TYPE | STOP | STRING | INTEGER | REAL`

`<dyadic-expression> ::= <expression> <dyadic-operator> <expression>`
`→ DYOP(left-expression, right-expression)`

`<dyadic-operator> ::=`
`'&' | '|' | ':' | '=' | <arithmetic-operator> | <relational-operator>`
`→ AND | OR | ASSIGN | ARITHMETIC-DYOP | RELATIONAL-DYOP`

`<arithmetic-operator> ::= '+' | '-' | '*' | '/' | '%' | '|' | '|'`
`→ PLUS | MINUS | TIMES | DIVIDE | OVER | CONCATENATE`

`<relational-operator> ::= '<' | '<=' | '=' | '>' | '>=' | '="'`
`→ LESS | NOT-GREATER | EQUAL | GREATER | NOT-LESS | NOT-EQUAL`

`<primary> ::= <unit> <subscript>*`

`<subscript> ::= '[' <expression> ']'`
`→ ARRAY-ACCESS`

`<unit> ::=`
`<constant> | <variable-or-call> | <call> | <fail-return>`
`| <if-expression> | <case-expression> | <while-expression>`
`| <for-expression> | <parenthesized-expression> | <array-expression>`

`<constant> ::=`
`<string-constant> | <integer-constant> | <real-constant> | undefined`
`→ CONSTANT(value)`

`<variable-or-call> ::= <identifier>`
`→ VARIABLE(name) | PROC-CALL(name, undefined)`

`<call> ::= <identifier> '(' <actuals> ')'`
`→ PROC-CALL(name, actuals)`

`<actuals> ::= { <expression> ',' }*`
`→ CALL-IN(source)*`

`<fail-return> ::= freturn`
`→ FRETURN`

`<if-expression> ::= if <expression> then <block> [else <block>] fi`
`→ IF(control, then-branch, else-branch)`

<case-expression> ::=
 case <expression> **of** ((<case-constants> | (**default** ':') <block>)* **esac**
 →CASE(CASE-SELECTOR(control, case-constants*), alternatives*)

<case-constants> ::= { <expression> ':' } +
 →CASE-CONSTANT(value+)

<while-expression> ::= **while** <test> **do** <block> **od**
 →WHILE-LOOP(test, body)

<for-expression> ::= **for** <identifier> **in** <expression> **do** <block> **od**
 →FOR-LOOP(FOR-CONTROL(counter, distributor), body)

<parenthesized-expression> ::= '(' <block> ')'

<block> ::= <local-variable-declaration>* { [<expression>] ';' } *
 →SEQUENCE(operands*)

<local-variable-declaration> ::= **var** { <identifier> ';' } + ';'

<array-expression> ::=
 array (<size-definition> [**init** <initial-values>] | <initial-values>)
 →ARRAY(size, initial-values)

<size-definition> ::= '(' <expression> ';' <expression> ')'

<initial-values> ::= '[' { <expression> ';' } + ']'

References

Klin82. KLINT, P. (1982). *From SPRING to SUMMER*, Mathematical Centre, Amsterdam.

Appendix II

Algorithm for Demand Graph Construction

This appendix contains most of the algorithm described in chapter 6. Only the final versions of each procedure are shown. Each procedure refers to the section where its explanation can be found.

BASIC OPERATIONS

<pre> use(key) of CHAINER if key in deflist return deflist[key] else if key not in uselist E := cocoon.entry-node(position) uselist[key] := E if key is not a node E.origin := environment.use(key).origin return uselist[key] </pre>	section 6.4
<pre> def(key, node) of CHAINER deflist[key] := node </pre>	section 6.3
<pre> attach of SEQUENCE attach all children in order </pre>	section 6.3
<pre> attach of CONSTANT source := use(Sink) def(Value, self) </pre>	section 6.3
<pre> attach of VARIABLE if is-a-use def(Value, use(name)) else def(name, use(Address)) </pre>	section 6.3

attach of ASSIGN	section 6.3
attach right-hand side	
save definition of Address	
def(Address, use(Value))	
attach left-hand side	
def(Value, use(Address))	
restore definition of Address	
attach of PUT	section 6.3
left-source := use(Standard-IO)	
attach actual parameter	
def(Standard-IO, self)	
attach of GET	section 6.3
source := use(Standard-IO)	
def(Value, link-node(0, self))	
def(Standard-IO, link-node(1, self))	

OPERATORS

attach of DYOP	section 6.4
attach left operand	
if Success in deflist	
install cocoon	
treat-right-operand within then-chainer	
dissolve cocoon	
else	
treat-right-operand	
treat-right-operand of ARITHMETIC-DYOP	section 6.3
left-source := use(Value)	
attach right operand	
right-source := use(Value)	
def(Value, self)	
treat-right-operand of RELATIONAL-DYOP	section 6.3
left-source := use(Value)	
attach right operand	
right-source := use(Value)	
def(Success, self)	
treat-right-operand of AND	section 6.3
attach right operand	
attach of OR	section 6.4
attach left operand	
install cocoon	
attach right operand within else-chainer	
dissolve cocoon	

CONDITIONALS

- attach of IF section 6.4
 attach condition
 create CONDITIONAL-COCOON
 link control of cocoon to use(Success)
 attach then-branch within then-chainer
 attach else-branch within else-chainer
 dissolve cocoon
- dissolve of CONDITIONAL-COCOON section 6.4
 create-branch-nodes
 create-merge-nodes
 export-definitions
- create-branch-nodes of CONDITIONAL-COCOON section 6.4
for each name that occurs in some deflist
 create BRANCH node and enter into export-list
 link each outlink of BRANCH node
 to use(name) in appropriate chainer
- create-merge-nodes of CONDITIONAL-COCOON section 6.4
for each name that occurs in some uselist
 create MERGE node and link to use(name)
 link LINK-IN nodes in uselists to MERGE node
- export-definitions of CONDITIONAL-COCOON section 6.4
for each [name,node] in export-list
 def(name,node)

LOOPS

- attach of WHILE section 6.4
 create LOOP-COCOON
 attach test-branch within test-chainer
 set control of cocoon to use(Success)
 attach body-branch within body-chainer
 dissolve cocoon
- dissolve of LOOP-COCOON section 6.4
for each name in some deflist or uselist
 create EXIT-LOOP node X
 link X to use(name) in test-chainer
if name in uselist of test-chainer
 let E be the ENTRY-LOOP node uselist[name]
 link E.entry to use(name)
 link E.last to use(name) in body-chainer
if name in uselist of body-chainer
 link uselist[name] to X.last
if name in some deflist
 def(name,X)

PROCEDURES

<pre> attach of PROCEDURE if not yet done create PROC-COCOON push new chainer def>Returns, Never def'Return-value, Void attach body pop chainer dissolve cocoon </pre>	section 6.5
<pre> attach of PROC-CALL attach called procedure treat-operands(actual parameters) for each <name,node> in inglobals link node to CALL-OUT(use(name)) for each <name,node> in outputs def(name, CALL-IN(node)) </pre>	section 6.5
<pre> treat-operands(list-of-operands) treat-operand(first of list-of-operands) if rest of list-of-operands is not empty if Exits in deflist create CONDITIONAL-COCOON and push else-chainer if Returns in deflist create CONDITIONAL-COCOON and push else-chainer if Success in deflist create CONDITIONAL-COCOON and push then-chainer treat-operands(rest of list-of-operands) pop chainers and dissolve cocoons </pre>	section 6.4
<pre> treat-operand(actual) of PROC-CALL attach actual link formal PARAMETER node to CALL-OUT(use(Value)) </pre>	section 6.5
<pre> attach of RETURN def>Returns, Always if there is an operand attach operand def'Return-value, use(Value) def'Return-signal, use(Success) else def'Return-signal, Always </pre>	section 6.5
<pre> attach of FRETURN def>Returns, Always def'Return-signal, Never </pre>	section 6.5

dissolve of PROC-COCOON section 6.5
 for each [name,node] in deflist
 if name is global variable
 outputs[name] := RESULT(node)
 else if name is Return-value
 outputs[Value] := RESULT(node)
 else if name is Return-signal
 outputs[Success] := RESULT(node)
 for each [name,node] in uselist
 if name is global variable
 inglobals[name] := node
 else if formal parameter
 formals[position of formal] := node

ARRAYS

attach of ARRAY section 6.6
 attach each initializing value and
 if any of their origin fields is not Simple
 error 'more dimensional array'
 origin := self
 def(Value, self)
 def(origin, self)

attach of ARRAY-ACCESS section 6.6
 if this is an update
 source := use(Address)
 if source.origin is not Simple
 error 'more dimensional array'
 attach index
 attach object
 object-source := use(Value)
 connect-to-previous-update(object-source.origin)
 if this is an update
 def(object-source.origin, self)
 else
 def(Value, self)

connect-to-previous-update(object-origin) of ARRAY-ACCESS section 6.7
 previous-update := object-origin.alias-access-graph

CONDITIONAL ALIASING

- alias-access-graph of ARRAY and BRANCH section 6.7
 return node returned by descend
 set lacap to Laca
- descend of LINK-OUT section 6.7
 case lacap of
 Descendant:
 return node returned by descend of child
 Ancestor:
 return node returned by use(parent)
 set lacap to Ancestor
- descend of ARRAY and BRANCH section 6.7
 if request already treated
 transmit to children
 else
 case lacap of
 Laca:
 return node returned by use(self)
 Descendant:
 return new ACCESS-BRANCH node with each
 LINK-OUT node linked to descend of corresponding child
 Ancestor:
 return node returned by treat-predecessor(first predecessor)
 transient-def(self, node to be returned)
 set lacap to Ancestor
- ascend(default-access, requesting-node) of BRANCH section 6.7
 return new ACCESS-BRANCH node with each
 LINK-OUT node linked to either
 if branch corresponds to requesting-node
 treat-predecessor(first predecessor)
 else
 default-access
- treat-predecessor(current-predecessor) of ARRAY and BRANCH section 6.7
 if request already treated or no more predecessors or lacap = Ancestor
 return use(self)
 else
 return ascend(treat-predecessor(next predecessor)) of current-predecessor
- ascend(default-access) of LINK-OUT section 6.7
 case lacap of
 Ancestor:
 return node returned by ascend(default-access, self) of parent
 Descendant:
 node to be returned is default-access
 set lacap to Descendant

STELLINGEN

behorende bij het proefschrift

The Misconstrued Semicolon Reconciling Imperative Languages and Dataflow Machines

van

Arthur Veen

1. Een gestructureerde imperatieve programmeertaal is geschikt voor het efficiënt programmeren van een dataflow machine.
2. De efficiënte verwezenlijking van een fijnkorrelige parallele computer vereist een uiterst snel synchronisatiemechanisme. Zo'n mechanisme is equivalent aan de representatie van een grote, schaars bezette, ruimte met een zeer snelle toegangstijd en een bescheiden geheugengebruik.

Paragraaf 2.3 van dit proefschrift.

3. Veel van de operaties die programma-analyse voor imperatieve talen bemoeilijken zijn bij applicatieve talen verborgen in de operaties op datastructuren.

Paragraaf 3.3 van dit proefschrift.

4. De efficiëntie van een parallele computer wordt voor een groot deel bepaald door de mate waarin programmalocaliteit op machinelocaliteit is afgebeeld.
5. Sommige computerarchitecten kiezen een hoog niveau machinetaal teneinde de semantische afstand tussen programmeertaal en machinearchitectuur te verkleinen. Dit komt de efficiëntie vaak niet ten goede, omdat het de ruimte beperkt waarin een compiler kan optimaliseren.

6. Bij de keuze van een programmeertaal is het belang van de intrinsieke eigenschappen van de taal omgekeerd evenredig met de omvang van het programmeerproject.
7. Bladeren aan plantenstengels zijn vaak in spiraalvormige patronen geplaatst. Deze patronen kunnen worden verklaard door een eenvoudig diffusiemodel, als wordt aangenomen dat het groeien van een nieuw blad wordt geïnduceerd op het moment dat lokaal de concentratie van een voor de groei noodzakelijke stof een drempelwaarde overschrijdt.

A. VEEN & A. LINDENMAYER (1977).

Diffusion Mechanism for Phyllotaxis, *Plant Physiology*

8. De prominente rol die binnen de Nederlandse informatica aan wiskunde is toebedeeld leidt tot een verwaarlozing van gebieden die zich vooralsnog moeilijk laten formaliseren.
9. Zolang informatica-onderzoek in Nederland beperkt wordt door gebrek aan geschikt personeel, en niet door gebrek aan onderzoeksgelden, kan van het verruimen van het onderzoeksbudget weinig effect worden verwacht.
10. Omdat bij vrouwen dominantie van de linker hersenhemisfeer vaker voorkomt dan bij mannen, dient de wiskunde van zijn mannelijk imago te worden ontdaan.