Condensed Graphs:
Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing

John P. Morrison
Technische Universiteit Eindhoven
Department of Mathematics and Computing Science

Cover: M.C. Escher, WATERFALL.
©1996 M.C. Escher / Cordon Art — Baarn Holland. All rights reserved.
Acknowledgment

I would like to thank my two promoters, Prof. Martin Rem and Prof. P.G. O’Regan for their help in the realisation of this thesis. I am also very grateful to Prof. Frans Kruseman Arctz and Dr Arthur Veen for their time, effort, and input.

In particular I would like to thank Prof. Rem for the many enjoyable years that we have spent working together. His professionalism, encouragement, and most of all his patience has been inspirational. I am greatly in his debt.

Thanks to my many colleagues, past and present, for the interesting discussions and constructive criticisms.

Not least, to my family, whose interest is continual and whose love is unconditional, thank you.

To Sinéad, Valerie, and Robin
—in lieu of time spent apart.
Condensed Graphs:
Unifying
Availability-Driven, Coercion-Driven
and Control-Driven Computing

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE
TECHNISCHE UNIVERSITEIT EINDHOVEN,
OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR. M. RIM,
VOOR EEN COMMISSIE AANGEWZEEN DOOR HET COLLEGE VAN DEKANEN IN HET
OPENBAAR TE VERDEIGEN
OP WOENSDAG 30 OKTOBER 1996 OM 16.00 UUR

DOOR

John Patrick Morrison

GEBoren Te CORK, IERLAND.
Dit proefschrift is goedgekeurd door
de promotoren
   Prof. dr. M. Rem
en
   Prof. P.G. O'Regan

"Everything Flows"
... Heraclitus of Ephesus, 500 B.C.
# Contents

## Prologue

## 1 Introduction
1.1 Overview .................................................. 1
1.2 The Field in Brief ........................................... 3
1.2.1 The Functional Approach ................................. 3
1.2.2 The Data-driven Approach ............................... 6
Dataflow Graphs ........................................ 8
Safe Dataflow Graphs ....................................... 8
The Static Approach ......................................... 9
The Dynamic Approach ....................................... 9
1.3 Persistent Problems ........................................ 9

## 2 Condensed Graphs
2.1 Overview .................................................. 13
2.2 Types of Directed Graph .................................... 13
2.3 Condensation ............................................... 15
2.4 Condensation Sequences .................................... 20
2.5 Seeded DAGs ............................................... 21
2.6 Seeded Condensation Sequences ............................ 23
2.7 Summary .................................................. 25

## 3 Computing With CGs
3.1 Representing Algorithms ..................................... 29
3.1.1 Ports and Nodes .......................................... 31
3.1.2 Algorithms and Condensation Sequences ............... 32
3.1.3 Stemming and Grafting ................................... 35
3.2 The CG Execution Model .................................... 37
3.2.1 The V-graph ............................................. 39
3.2.2 Subgraph Constant Evaluation .......................... 40
3.2.3 CG Node Deconstruction ............................... 41
3.2.4 Mutual Reduction ....................................... 42
3.2.5 Some Examples ......................................... 43
3.3 Summary .................................................. 46
Prologue

At the outset of this research, our interest lay in addressing some of the problems which have traditionally thwarted advances in general purpose dataflow computing. We were, and indeed still are, of the opinion that problems such as the poor communication-processing ratio, remaining packet garbage, large tag sizes and cumbersome tag management could, and can, be solved by using graph encapsulation. This encapsulation can be used to map modularity in a dataflow program into locality on a dataflow machine. To accommodate this mapping we proposed to work with graphs whose nodes represented either simple operations or other similar graphs. With this formalism, nodes representing graphs can be allocated and deallocated as a whole, tags can be associated with nodes, tag management can be associated with memory management, remaining packet garbage can be cleaned by a single graph deallocation, and indeed with appropriate processors the granularity of execution can be raised and hence the communication-processing ratio improved.

We soon discovered that these graphs, which we termed condensed graphs, could be used to model data as well as operations. Furthermore, when we allowed these graphs to flow on arcs of other condensed graphs we not only had a mechanism to model the flow of arbitrarily complex data, but also a natural way of representing lazy evaluations. We realized immediately that the resulting model of computation was far more powerful than the dataflow model alone.
Chapter 1

Introduction

1.1 Overview

Most would agree that the computer has become an integral, and in some cases indispensable, part of modern living. Many sophisticated computer applications have been, and indeed are being, developed and at faster and faster rates. These applications perform a wide diversity of tasks ranging from the management of corporate finances to the diagnosis of illnesses and the control of life-saving medical equipment. Artificial intelligence, whose very name implies anthropomorphic qualities, adds further to the mystique by seeming to endow the computer with the ability to think.

In reality, the computer is a machine capable only of executing simple specified instructions (albeit at superhuman speed) and according to what we call a computing model. There is a number of these models and they are distinguished from each other by the manner in which they sequence instructions for execution. In the traditional imperative model, the programmer explicitly determines the sequencing constraints; in an availability-driven model, the sequencing of instruction execution depends only on the availability of operand data; and, in a coercion-driven model, instructions are executed when, and only when, their results are needed.

In the imperative model, the sequencing constraints are determined by the flow of control from one instruction to the next. Consequently, we term it the control-driven model. The control is used in the stepwise manipulation of named data, and due to the emphasis on sharing these data, it is inherently sequential. Parallel implementations are actively being investigated and in some cases are resulting in the design of new languages for capturing modular design and for expressing communications subsystems. These systems, however, remain imperative and place the onus on the programmer to construct parallel solutions from collections of interacting sequential parts.

In contrast, availability-driven and coercion-driven models are inherently parallel. This parallelism comes from having computations which are free from side-effects, allowing subcomputations to be performed concurrently without interfering with each other. Side-effects result from resource sharing. Therefore, the concept of a shared mutable memory — which is central to the control-driven model — cannot be handled naturally by availability-driven and coercion-driven computations. The availability-driven
and coercion-driven models both arrive at a result by the process of reduction. Each
applies a different reduction order. In availability-driven computations, reduction be-
gins at function operands. This facilitates a lot of parallelism, however, the eagerness
of these reductions also can lead to wasted computations. In contrast, reduction in a
coercion-driven system begins at function definitions and only the minimum amount of
work necessary to produce a result is precipitated. Thus, there is no wasted effort, but the
overall computation time is lengthened by the delay in starting subcomputations, which
are necessary to produce the result.

The rationale which these models employ for sequencing instruction execution are,
in many respects, opposites. As such, they are suggestive of different implementa-
tion strategies and so each has been studied separately. Occasionally, researchers have tried
to combine both in hybrid architectures [1, 16, 55], but with limited success.

We maintain that there is a model of computation — and an associated implementation
strategy — which contains these three models as special cases. We call it the condensed
graphs model. This model is not a hybrid. It exploits topological properties of directed
acyclic graphs to define the notion of a computation triple. This triple embodies three
prerequisite requirements for the evaluation of any function, namely: an appropriate set
of inputs, a function description, and a destination — somewhere to put the result.

The condensed graphs model of computation operates by associating graphs, which
contain complementary triple elements, to form new condensed graphs. These new
graphs contain all the complementary elements of its associates and when a graph with
all three elements is constructed, it represents an executable instruction. Upon execution,
a condensed graph is produced containing less than three elements. In this way, the
sequencing of instruction execution is determined by the order in which computation
triples are constructed. When any two elements are fixed in the static description of
a condensed graph, the sequencing is determined exclusively by the third element. We
shall see that when that element corresponds to the operands, the availability-driven model
results; when it corresponds to the function, the control-driven model results; and when
it corresponds to the output destination, the coercion-driven model results. Thus each
of the three principle computation paradigms is a special case of the condensed graphs
model.

In Chapter 2 a formal foundation for the condensed graphs model of computation is
introduced. We review some concepts of directed graphs and the process of condensation.
A mechanism is introduced for retaining the information normally lost during condensa-
tion and an inverse process called evaporation is defined. Sequences of condensed graphs
are constructed by repeated application of the condensation process and a mechanism
for navigating a condensation sequence is presented. The terminology introduced in this
chapter is used repeatedly throughout the thesis.

Chapter 3 introduces the condensed graphs model of execution and shows how the
condensation sequences of Chapter 2 can be used to represent algorithms. The concept of
a computation triple is introduced and two distinct ways of associating triple elements to
form instructions are considered. The chosen triple association type, and the mechanism
for navigating condensation sequences introduced in Chapter 2, are used as the basis for
determining the sequence of instructions in the execution model.
1.2. **THE FIELD IN BRIEF**

In Chapter 4 a machine model for constructing and executing computation triples is outlined. Practical machine aspects are considered, including the handling of speculative parallelism, the implementation of a throttling mechanism and the rolling-back of deadlocked computation. Some example programs and results of simulations are presented.

In Chapter 5 we show that structured data can be represented naturally in the condensed graphs model. Moreover, these structures are manipulated (created, changed, and destroyed) in the same manner as every other condensed graph. We show how data structures are created lazily, how infinite data structures are represented, and how the condensed graphs model is garbage-free. In that chapter we also consider control-driven aspects of the condensed graphs model in conjunction with stored data and mutable memory.

Chapter 6 summarizes the work done and the results obtained. In particular, we look at how well we have addressed some of the more well-known and persistent problems of the field.

In the remainder of this chapter we shall look briefly at the context of this research. This will enable us to introduce some standard terminology and concepts as they apply to this field of study. Finally, we summarize some of the more persistent problems which have been encountered, by other researchers, and which will be addressed again in subsequent chapters from the perspective of the condensed graphs model.

## 1.2 The Field in Brief

### 1.2.1 The Functional Approach

Back in 1977, in his ACM Turing Award Lecture [14], John Backus vividly portrayed conventional, Von Neumann, programming languages as 'large', 'complex', 'inflexible', and with a limited power of expression. The reason for this, he states, is that the succession of programming language designs, based on the Von Neumann model of computing, simply tends to add feature upon feature to the earlier, more simple designs which, while becoming larger, did not become justifiably more powerful.

Backus emphasized that the Von Neumann model of computing is built around a *bottleneck*: the word-at-a-time communication channel which connects the central processing unit and the memory store. The Von Neumann bottleneck results from programs employing a *state-transition* semantics in which complex states are successively transformed until a termination state is reached. These state-transition semantics are implemented by pumping single words back and forth through the bottleneck. This inherent characteristic has percolated into the design of conventional languages where programming still concerns itself with the word-at-a-time traffic; this time through a slightly more sophisticated linguistic bottleneck represented by the assignment symbol. A functional program is built entirely of nested functions. The 'main program' is written as a function and is typically defined in terms of other functions which in their turn are defined in terms of still more functions until, at the bottom level, functions are language primitives known as *atoms*. Functional programs receive their inputs as function arguments and deliver outputs as function results. In the functional paradigm, there is no explicit notion of state,
and 'variables' (a term which in this case is somewhat of a misnomer) are associated with specific values which never change throughout the execution of a program. The application of a function to its arguments is analogous to the evaluation of an instruction in a conventional language. Functional programs can be evaluated by rewriting functions (also called definitions) and so reducing them to a simpler form. A function which is in its simplest form and cannot be reduced any further is said to be in normal form. When reducing a function to a simpler form, its parameters can be manipulated in at least two ways. This gives rise to two forms of function reduction known as string reduction, and graph reduction, respectively. In a string reduction each function accessing a particular parameter will take and manipulate a separate copy of that parameter. In contrast, graph reduction is based on the sharing of arguments using pointers. Since there is never more than one copy of any parameter in the graph reduction mechanism, none is reduced more than once, regardless of the number of times it is used.

Functions of \( n \) parameters but which are supplied with less than \( n \) parameters cannot be reduced fully to normal form.

This is the case, for example, with

\[
\lambda x y. (+ x y)(* 3 4)
\]

which is a function of two formal parameters and only one actual parameter. A reducible function, or a reducible expression, is known as a redex. Thus, a redex is a function of \( n \) parameters applied to \( n \) arguments.

In some functions, not all parameters are necessary for every invocation. This characteristic gives rise to the definition of strict and non-strict functions.

**Definition 1.2.1 (Strict)**

A function is said to be strict in one or more of its arguments if it requires those arguments to be reduced (usually to normal form) before the function itself can be reduced.

A function which is strict in all its arguments is said to be a strict function.

If a function is not strict in any of its arguments it is said to be a non-strict function.

Just as there are two ways of manipulating parameters in a function reduction, there are also two distinct ways a reduction can be applied. These refer to the order in which the reduction takes place: the parameters first, or the function first. They are known as applicative-order, and normal-order reductions, respectively.

In an applicative-order evaluation, all functions are considered to be strict and all arguments are evaluated exactly once before the function is reduced. In lambda calculus terms, applicative-order evaluation is defined to be leftmost innermost first (the leftmost part here is chosen by convention; in a parallel execution this may not have any real significance). For example, consider

\[
(\lambda z. z z)(* + 3 4)
\]
1.2. THE FIELD IN BRIEF

If this is to be reduced using an applicative-order evaluation, the subexpression \((+ 3 4)\) would be evaluated first, the result would be substituted for \(x\), and finally \((+ 7 7)\) would be evaluated.

In a normal-order evaluation, a function may be non-strict in zero or more of its arguments. Also, arguments may be evaluated zero or more times as appropriate. In lambda calculus terms normal-order is defined to be leftmost outermost first (again, the leftmost part here is chosen by convention and in a parallel execution it may not have any real significance). Considering our previous example again, the following sequence of reductions would result:

\[
(\lambda x.x x)(+ 3 4) \rightarrow
\]
\[
= (+ 3 4)(+ 3 4) \rightarrow
\]
\[
= (+ 3 4) \rightarrow
\]
\[
= 7 \rightarrow
\]
\[
= 49
\]

The 'outermost first' part of the rule tries to reduce the function \((\lambda x.x x)\) without considering its arguments. This is not possible and so the arguments are substituted unevaluated into the function. Now, since \((\ast)\) is strict in both its arguments, both must be evaluated before \((\ast)\) can be reduced. This then will result in the argument \((+ 3 4)\) being evaluated twice. A graph reduction variation on the normal-order evaluation scheme attempts to reduce the number of argument evaluations by exploiting named shared subexpressions. Thus, a shared subexpression is evaluated only once and the result of that evaluation is shared with all other occurrences.

A normal-order reduction of a function will always terminate provided that a fixed-point solution for the function exists. In some cases this fixed-point may not be reached by an applicative-order reduction (although if the applicative-order reduction does terminate, it will do so with the same result as the normal-order reduction — Church-Rosser Theorem). As an example consider the following

\[
\text{Let } E = (\lambda x.x x)(\lambda x.x x)
\]

In this example the evaluation of \(E\) will never terminate. Now consider

\[
(\lambda x.3)
\]

This function takes one argument and always returns the value 3. If we try to evaluate \((\lambda x.3)E\) in a normal-order evaluation we will get the result 3, however, an applicative-order evaluation will attempt to evaluate \(E\) first and hence will never terminate.

A property which makes functional programs amenable to mathematical manipulation is that a reducible function and its result are referentially transparent. Thus, in particular, a functional program is mathematically equivalent to its result in the same way, for example, that the expression \(3 + 3\) is equivalent to the number 6.

Function execution has no effect other than to compute a result. We say that functional programs are free from side-effects. This property can help to eliminate a major source of bugs by relieving the programmer from having to explicitly specify the flow of control
in his program, since the execution order of functions, from a semantic point of view, is unimportant.

To build large and maintainable systems it is important that software be well-structured. The functional language features of higher-order functions and lazy evaluation help greatly in increasing the modularity and reuse of functional programs. In turn, the development of reusable modules helps to reduce future programming costs and also aids software writing and debugging.

Thus, in summary, we have seen that an applicative-order evaluation has an innermost select rule, consequently favouring a computing model which is driven by the availability of data. Similarly, a normal-order evaluation has an outermost select rule and the most natural computing model for its implementation is based on coercion-driven semantics.

Coercion-driven computing has become known in the literature as demand-driven computing. Quite a number of projects have been undertaken to investigate this computing model and more details can be found in [42, 39, 16, 45, 29].

Availability-driven computing is known in the literature as dataflow (or data-driven) computing and this area has also been the focus of much research [12, 27, 49, 44, 56, 37, 22, 46, 2, 20, 19].

1.2.2 The Data-driven Approach

The concept of data-driven execution has been around for nearly thirty years. Originally dataflow graphs were used in the design and verification of operating systems. However, it was not long before their potential as the base language for a new data-driven architecture was recognized.

Dataflow schemas were introduced in 1966 by Karp and Miller [40] and in 1969 by Rodriguez [47]. These ideas were further developed by Dennis [26] who, together with Misunas, proposed the first data-driven architectural design in 1974 [28].

Since then numerous other dataflow projects have been undertaken and today many are still going strong.

From the beginning, much optimism has been voiced about the suitability of dataflow as a viable alternative to the traditional, imperative, control-driven model. For instance it has been said that

“The most significant and distinguishing aspect of this new scheme of interpretation is that it permits a literal exchange of processing elements for computation time, in a very general and mechanical way.” [7]

“If sufficient resources are provided, the [dataflow] processor can exploit all concurrency present in the program. This approach extends naturally to an arbitrary number of processors.” [2]

“Among parallel computer architectures for large-scale numerical computations, dataflow computer architecture is one of the most promising, since the dataflow concept exploits intrinsic parallelism in programs at the architectural level. In dataflow, execution of an instruction can be performed any time after
1.2. THE FIELD IN BRIEF

the arrival of all of the operand data. Since this is the only requirement to be satisfied to sequence execution, a high degree of parallelism can be achieved.” [57]

“The advantage that dataflow architectures have over other approaches to high-performance computation is that the scheduling and synchronization of concurrent activities are built in at the hardware level, enabling each instruction execution to be treated as an independent concurrent action. This allows efficient fine grain parallelism, which is precluded when synchronization and scheduling functions are realized in software or microcode.” [23]

Despite such claims we find today that dataflow is far from taking over from control flow as a model for general purpose computing. The reasons for this are many and varied. Undoubtedly social and economic factors always play a significant role when the new infringes upon the established: one cannot replace fifty years of experience and familiarity, not to mention code, overnight. The wealth of FORTRAN code still in use today is a case in point.

Apart from those social problems of change which have hindered its progress, the field of dataflow computing has been hampered with what seem to be inherent problems of a technical nature.

For example, the exploitation of fine-grain parallelism, which is the technical basis for the above claims, also turns out to have many negative characteristics, namely:

1. Fine-grain dataflow machines have a high communication to processing ratio.

2. They need tag management. Every datum needs to be tagged with context information. The amount of context information is usually large, necessitating high communication bandwidths. Also, the issue and recycling of tags imposes significant processing overhead.

3. They can give rise to copious amounts of parallelism. In some cases the parallelism of the algorithm can be much more than the parallelism of the machine. This can lead to poor resource management (mainly memory) and even to deadlock.

4. They usually need special hardware to bring together (that is, to match) data belonging to the same instruction. This ideally, it has been suggested, is Content Addressable Memory (sometimes called Associative Memory), which while effective, is also expensive and difficult to produce in large amounts.

5. They are poor at exploiting locality.

Still other issues which have hindered the progress of general purpose dataflow computing, and which have been addressed with varying degrees of success by different researchers, include

- The efficient handling of structured data.
- The design of a general purpose dataflow operating system.
INTRODUCTION

Dataflow Graphs

In general, a dataflow graph is a collection of nodes connected by directed edges (usually called arcs). A node may have many incoming and many outgoing arcs.

Certain nodes have input arcs that do not originate from other nodes: these are called global input arcs. Conversely, arcs that do not go to any node are called global output arcs.

Data can flow along arcs in special 'containers' known as data tokens. Data flowing into a node constitute the input data for that node and operationally each datum flows into a distinct input port on that node.

Nodes transform their input data by performing some operation that is statically associated with that node. This is usually achieved by constructing an instruction token comprised of input data, an operation and a list of destination addresses (which may vary from execution to execution) to receive the result. The subsequent execution of this instruction results in the production of output data tokens that, in turn, become the input data to the nodes in the destination address list.

In a strict data-driven execution all input data must be present in the input ports of a node before an instruction token can be formed. This condition is sometimes referred to as the strict firing rule. Some implementations allow non-strict firing: only a particular subset of input data needs to be present before an instruction token can be created for execution. These implementations need further mechanisms to deal with the superfluous input data when, and if, they eventually arrive. Non-strict firing can expose more parallelism but the enabling mechanism (which determines if a node can be fired) is more complicated.

Safe Dataflow Graphs

A safe dataflow graph is one in which no input port of any node contains more than one data token — provided that only one data token is ever placed on any global input arc.

In an acyclic graph, safety can be determined statically — if more than one arc is targeted at the same input port of a node then the graph is unsafe.

In a cyclic graph, however, safety determination is more difficult. This is considered, for example, by Veen in [53]. Related to the decision of whether or not a graph should be cyclic is the mechanism used for the safe implementation of reentrant code, i.e., pieces of the code that are executed more than once. This is a major factor in differentiating between different dataflow models of computing. Broadly speaking, there is the static[15, 25, 24], and the dynamic approach.

The Static Approach

The static approach to handling reentrant code imposes the precondition that at most one data token be allowed to reside on any arc at any time. There are basically two ways of guaranteeing this: by locking or by acknowledgments. The acknowledge method can be shown to allow more concurrency than the locking method, however, it requires double the amount of arcs and tokens.
1.3 Persistent Problems

Finally, we provide here a synopsis of some of the persistent problems associated with availability-driven and coercion-driven systems.

- **The matching bottleneck.** The matching unit in a dataflow machine sequentially matches and synchronizes operand data. If there is a lot of matching to be performed, traffic in the pipeline between the matching unit and the processing unit becomes rare. This reduces the utilization of the processing unit, and so the pipeline beat of the dataflow ring becomes dictated by the speed of the matching unit.

- **Remaining packet garbage.** Tokens waiting in the matching store after program execution has terminated are called remaining packet garbage (RPG). RPG typically results when a non-strict firing rule is adopted in a dataflow machine; superfluous arguments will eventually be produced and never consumed.

- **Control of parallelism.** Dataflow computations can exhibit huge amounts of parallelism by the breadth-first execution of dataflow graphs. This type of execution can use excessive amounts of storage, since many partial results are created long before they can take part in an instruction execution. A mechanism is thus needed to restrict the uncovering of unnecessary parallelism and to match the parallelism of the machine to that of the computation. Such a mechanism is called a throttle. The design of an effective throttling mechanism remains an open question, although preliminary work has been carried out in the area by the Manchester group [48].
• **Load balancing.** In a distributed dataflow architecture (like the ID machine [5, 6, 4, 46, 3, 2, 8, 9, 10, 7], or the multi-ring Manchester machine [37, 34, 36, 33, 32, 35, 54]) unbalanced load distribution can degrade system performance. Therefore, efficient techniques are needed to exploit nodal locality and so reduce inter-processor communication.

• **Linear subgraph processing.** A linear subgraph in a dataflow graph represents a sequential computing segment in the dataflow program. The execution of such a segment, in which the program parallelism is less than the machine parallelism, can have a dominating effect on the performance of the total computation. Such computations are not uncommon as they appear in recursive procedure calls and in iterative computations that rely on results generated in a previous iteration.

• **Data Structure handling.** The absence of side-effects in the dataflow approach means that mutable data structures are not allowed. Thus, from theoretical requirements, modification of a data structure necessitates the copying of that structure with some elements appropriately modified. From an implementation perspective this copying precipitates large performance and memory overhead. Therefore, we would ideally like to retain the side-effect-free nature of data structure handling at the language level and do away with the need for excessive copying at the implementation level.

This usually implies a careful trade-off in maintaining the functionality of structure handling and in avoiding excessive processing overhead.

The data structure processing speed is affected by

- the data structure accessing scheme
- locality of execution of the operations on the data structure
- employing more sophisticated processing elements to execute data structure operations of higher complexity
- memory management:
  - size of memory required
  - Garbage Collection algorithms employed

The real effect of any of these points, however, depends on how we view our structure. There are two ways of doing this: as a single complex value, or as a collection of simple values.

In the 'collection of simple values' view all elements of a data structure are labeled to distinguish them from each other. For example, one-dimensional arrays are represented as a fixed number of tokens traveling on one arc with an index field of the token signifying its position in the array. Similarly, records are implemented by designating one arc of the directed graph to each record field.

Recursive data structures cannot be supported by the 'collection of simple values' view alone. To support these types of structures, and also to reduce the amount of
1.3. **PERSISTENT PROBLEMS**

'data traffic' in a dataflow ring, the 'single complex value' view of data structures is used.

In the 'single complex value' view, the data structure remains in memory and a pointer to it is circulated around the dataflow ring. The use of explicit pointers to selectively update a data structure can be very efficient. However, dataflow single-assignment languages cannot support the use of explicit pointers, because such pointers can be used to produce side-effects.

Thus there are restrictions on how pointers can be manipulated in a dataflow environment.

- **Exploiting locality** How can a high degree of spatial locality be achieved simultaneously with a high degree of processor utilization?

- **Exploiting Parallelism** In a coercion-driven machine a delay in initiating a parameter evaluation may result in a delay in the overall computation. This characteristic has led to speculative computations which have their own associated problems.

- **Speculation** A speculative task is one which is performed in the hope that its result will be used in the future. If this is the case, we say that the speculation has become conservative: in effect, it was worthwhile. On the other hand, speculative parallelism may become irrelevant and should then be terminated. Killing irrelevant speculative tasks, and giving priority to conservative tasks, when a system is loaded is difficult [45].
Chapter 2

Condensed Graphs

2.1 Overview

We know from Chapter 1 that the structure of a computation can be modeled using a graph; in fact, graphs are extensively used to describe data-driven computations. This is not surprising; after all, what is more natural than to represent the structure of a computation using that body of mathematics concerned with the abstract notion of structure?

The field of mathematics known as topology and graph theory began with Euler in the eighteenth century. Since then, many contributions have been made by such men as Cayley (who was interested in structural problems in chemistry), Kirchoff (famous for his laws of electrical network theory), and by countless other engineers, mathematicians and logicians.

The theory of directed graphs, or more briefly digraph theory, deals with abstract configurations called digraphs — which consist of vertices and directed edges. When these terms are given concrete referents, digraphs serve as mathematical models of empirical structures, and properties of digraphs reflect structural properties of the empirical world.

Digraph theory offers the researcher an enriched vocabulary, techniques for calculating certain features, and a wealth of theorems which may give additional information when dealing with empirical structures. [38]

In this chapter we review the concepts of directed graphs, acyclic graphs and graph condensation. We introduce a number of new concepts, which we use to define condensed graphs.

With regard to the notation used in this chapter, we represent all graph vertices using the typewriter font, for example: p, v, and w. All sets will be denoted by uppercase, italic letters, such as V, P, and E. Finally, we use calligraphic letters such as \( \mathcal{G} \), \( \mathcal{CG} \), \( \mathcal{V} \) and \( \mathcal{P} \) to represent graphs and subgraphs.

2.2 Types of Directed Graph

To be precise, digraph theory is a special case of the more general mathematical theories of nets and relations. The axiom systems of all three theories have two primitives in common: a finite set, \( V \), of "vertices", and a finite set, \( E(\subseteq V \times V) \), of "directed edges".
Thus, nets, relations and digraphs are all graphs with directed edges. However, they differ in a number of ways. In a net, more than one edge may connect two specific vertices. Such edges are called “parallel lines” (or “multiple arcs”). This is the most general system for that reason. In a relation, there are no parallel lines. Thus, every relation is a net but not *vice versa*. Finally, a digraph may be described as an irreflexive relation. Schematically, this means that no vertex is connected directly to itself. In graph theory parlance: there are no loops. Fig. 2.1 illustrates a net, a relation and a digraph.

![Net](image1.png) ![Relation](image2.png) ![Digraph](image3.png)

Figure 2.1: Part (a) shows a net with parallel lines and loops. Part (b) shows a relation, there may be loops but no parallel lines. Part (c) depicts a digraph: no loops or parallel lines are allowed.

We use the generic term *directed graph* for each of these three types of graph, and will use the term digraph only in the restricted sense described above.

Every vertex in a directed graph has zero or more incoming edges. The number of these edges is called the *in-degree* of that vertex. We write the in-degree of a vertex $v$ in a graph $\mathcal{G}$ as $\text{InDeg}(v, \mathcal{G})$. Similarly, the number of out-going edges from a vertex is its *out-degree*. We write $\text{OutDeg}(v, \mathcal{G})$ to denote the out-degree of a vertex $v$ in a graph $\mathcal{G}$.

Our main interest is in Directed Acyclic Graphs, or DAGs. In every DAG there exists, by definition, a non-empty set of vertices called *transmitters*, whose in-degree is zero, and a non-empty set of vertices called *receivers*, whose out-degree is zero. A vertex may be both a transmitter and a receiver — if both its in-degree and out-degree are zero. We call all other vertices of the DAG *internal vertices*. A graph consisting of one vertex and no edges is called a *trivial* DAG, with this single vertex acting as both transmitter and receiver.

Associated with every vertex, $p$, of a DAG are two disjoint sets of vertices — known as the immediate predecessors, and immediate successors, of $p$. Either or both of these sets may be empty depending on whether $p$ itself is a transmitter, a receiver, or a trivial DAG. The cardinality of the set of immediate predecessors of $p$ is the *in-degree* of $p$ and a vertex, $x$, is in this set if there is an edge from $x$ to $p$ in the DAG. Similarly, the cardinality of the immediate successor set of $p$ is the *out-degree* of $p$ and a vertex $y$ is in this set if there is an edge from $p$ to $y$ in the DAG. We call the immediate predecessor set of $p$: the *input environment* of $p$, and the immediate successor set of $p$: the *output environment* of $p$. We refer to the input environment and the output environment of $p$, collectively, as the *environment* of $p$. The environments of a number of vertices are depicted in Fig. 2.2.

It is sometimes useful to be able to consider parts of a graph in isolation. We can do this by employing the notion of an induced subgraph.
2.3. CONDENSATION

Figure 2.2: The figure illustrates the environments of various vertices. \( P \) (marked by a black dot). In (a), \( P \) is a transmitter; it has no input environment and its output environment is the set of shaded vertices. In (b), \( P \) is an internal vertex, and its environment is the set of shaded vertices. Those vertices which are not part of the environment of \( P \) are not shaded. In (c), \( P \) is a receiver; it has no output environment and its input environment is the set of shaded vertices.

**DEFINITION 2.2.1 (Induced Subgraph)**

Given a graph, \( G = (V, E \subseteq V \times V) \), and a set \( W \) such that \( W \subseteq V \), we say that the graph \( H = (W, E \cap (W \times W)) \) is an induced subgraph of \( G \).
We use the notation: \( H = IG(G, W) \).

\[ \diamond \]

2.3 Condensation

The process of graph condensation is an effective way of gaining insight into the structural properties of a directed graph. In this process a new, simpler, directed graph is constructed by replacing certain subgraphs of the old directed graph by vertices, and by joining these new vertices in a specified manner.

The subgraphs of the directed graph to be condensed are identified by partitioning its vertex set.

**DEFINITION 2.3.1 (Partition of a set)**

A partition of a set \( S \) is a decomposition of \( S \) into pairwise disjoint nonempty subsets whose union is all of \( S \).

\[ \diamond \]

Thus, each element of a set appears in exactly one element of a partition of that set. A condensation of a directed graph is always performed with respect to a particular partition of its vertex set.

**DEFINITION 2.3.2 (Condensation of a directed graph)**

The condensation of a directed graph \( D \), with respect to a partition \( P \) is the directed graph whose vertices are the elements of \( P \) and whose edges are determined by the following rule: There is an edge from vertex \( P_i \) to vertex \( P_j \)
(i \neq j) in the new directed graph if and only if in D there is at least one edge from a vertex of P_i to one of P_j.

Although it is not explicitly stated in the description given in [38], we assume that i \neq j in the above description.

We will use the notation cond(D,P) to represent the condensation of D with respect to the partition P, or simply cond(D) when the partition is obvious from the context.

As an example of condensation, consider the following.

EXAMPLE 2.3.3 (Computer Communications)

Let us assume that the directed graph in Fig. 2.3 represents the flow of information between machines on a computer network.

![Figure 2.3: Computer Communications](image)

Figure 2.3: Computer Communications

Now, suppose that these machines are distributed over the three buildings: B_1, B_2 and B_3. Furthermore, partitioning the set of machines according to the buildings they occupy, gives us the following:

\[ P = \{ B_1, B_2, B_3 \} = \{ \{ m_1, m_4, m_6 \}, \{ m_2, m_7 \}, \{ m_8, m_9 \} \} \]

Condensing the computer communications graph with respect to this partition gives us the graph shown in Fig. 2.4. In this graph we see clearly that communication may enter and leave Building B_1, may only enter Building B_2, and may only leave Building B_3.

![Figure 2.4: Condensed Computer Communications](image)
2.3. CONDENSATION

Here then, we see an advantage of condensation in that the communication structure evident from the condensed graph is not immediately apparent from the original graph.

There are of course many ways of partitioning the elements of a vertex set; the insight gained into the structural properties of the graph depends entirely upon the partition chosen.

For completeness we now introduce two partitions which, even though extreme, are nonetheless interesting. The first of these is the graph-trivial partition.

**Definition 2.3.4 (Graph-Trivial Partition)**

A partition will be called graph-trivial if it consists of a single element.

A graph-trivial condensation is obtained by condensing a graph with respect to graph-trivial partition and will reduce the graph to a single vertex and no edges. That is, it becomes a trivial DAG.

At the other end of the spectrum we have the point-trivial partition.

**Definition 2.3.5 (Point-Trivial Partition)**

A partition will be called point-trivial if each element is a singleton.

A point-trivial condensation is obtained by condensing a graph with respect to the point-trivial partition and will leave the number of elements in the vertex set unchanged (although the ‘names’ of the vertices may change due to labeling). On the other hand, however, the number of edges may change. In particular, all parallel lines and loops will be removed. In other words, every net and relation is turned into a digraph by such a condensation.

Having looked at both ends of the spectrum, it is easy to see that the number of vertices of a graph does not increase under condensation and indeed will decrease if the partition is not point-trivial.

Condensation with respect to the point-trivial partition illustrates a change in a topological property of a directed graph. It thus becomes important to know the extent of influence of condensation on an arbitrary graph. Are there any properties that are always preserved? How can we preserve certain desired properties?

Since we are most interested in DAGs, we note that the acyclic property is not necessarily preserved by condensation as Fig. 2.5 illustrates:

The partition chosen in Fig. 2.5 places all the transmitters and receivers of the graph into one partition element. The resulting condensation is cyclic and has no transmitters or receivers. In general, determining a partition in this manner will either lead to a trivial DAG (if every vertex of the graph is either a transmitter or a receiver) or it will lead to a cyclic graph, that is a graph which contains a cycle, with no transmitters or receivers. In fact, in the latter graphs every vertex lies within the cycle.

It is also possible to construct partitions which will either eliminate all the transmitters or all the receivers separately from a DAG (Fig. 2.6) — but again this will always be
Figure 2.5: Part (b) shows the condensation of Part (a) with respect to the partition: \( \{ \{a, b\}, \{c\} \} \)

at the cost of introducing a cycle, since a DAG has both transmitters and receivers by definition.

Figure 2.6: It is possible to condense a DAG so as to eliminate all the receivers or all the transmitters from a DAG. The figure illustrates each of these cases. In Part (b) all the receivers are eliminated, and in Part (c) all transmitters are eliminated.

It is possible to turn any digraph, cyclic or acyclic, into a DAG by choosing appropriate DAG enforcing partitions. The graph-trivial partition is an obvious example of a DAG enforcing partition.

We assume that all our graphs are DAGs, therefore we will only be concerned with DAG preserving partitions.

**Definition 2.3.6 (DAG Preserving Partition)**

We define a partition to be DAG preserving if the condensation of a DAG with respect to that partition is itself a DAG.

We say that a condensation performed with respect to a DAG preserving partition is a DAG preserving condensation.

The three element partition \( P = \{ \{\text{transmitters}\}, \{\text{internal}\}, \{\text{receivers}\} \} \) which groups together all transmitters, all internal vertices, and all receivers into separate partition elements, is an example of a DAG preserving partition.

Condensations with respect to DAG preserving partitions have pleasing properties which are expressed in the following lemmas.
2.3. CONDENSATION

LEMMA 2.3.7 (Transmitters to Transmitters, Receivers to Receivers)

Let \( P = \{ P_1, P_2, \ldots, P_n \} \) be a DAG preserving partition of a DAG \( \mathcal{D} \). If \( P_i \) is a transmitter (receiver) in cond(\( \mathcal{D}, P \)) then it contains at least one transmitter (receiver) of \( \mathcal{D} \).

PROOF
Suppose \( P_i \) is a transmitter of cond(\( \mathcal{D}, P \)) and does not contain a transmitter of \( \mathcal{D} \). In this case, all elements of \( P_i \) by definition, have at least one incoming edge. Since \( \mathcal{D} \) is a DAG, at least one such edge must originate from outside \( P_i \), and so \( P_i \) could not be a transmitter of cond(\( \mathcal{D}, P \)).
A similar argument can be made for the case that \( P_i \) is a receiver in cond(\( \mathcal{D}, P \)).

As a consequence of Lemma 2.3.7, we note that the number of transmitters (receivers) in cond(\( \mathcal{D}, P \)) is less than or equal to the number of transmitters (receivers) in \( \mathcal{D} \).

LEMMA 2.3.8 (Nesting Single Transmitter/Receiver)

If \( \mathcal{D} \) is a DAG with a single transmitter (receiver) and \( P \) is a DAG preserving partition of \( \mathcal{D} \), then cond(\( \mathcal{D}, P \)) will also have a single transmitter (receiver) and this transmitter (receiver) will contain the transmitter (receiver) of \( \mathcal{D} \).

PROOF
The proof follows directly from Lemma 2.3.7. In an extreme case \( \mathcal{D} \) may be condensed to a trivial DAG — which by definition is a single vertex acting both as a transmitter and a receiver.

The choice of partition greatly influences the form of, and consequently the information which can be derived from, a condensed graph. In practice, partitions are determined by grouping together vertices of the original graph which are empirically "related". These "related" vertices can be represented as a subgraph of the original — induced by the partition element.

DEFINITION 2.3.9 (Induced Partition Subgraph)

If \( P = \{ P_1, P_2, \ldots, P_n \} \) is a partition of a graph \( \mathcal{D} \), we call \( P_i = IG(\mathcal{D}, P_i) \) the \( i \)th induced partition subgraph of \( \mathcal{D} \).

We use the notation \( \mathcal{P} = IG(\mathcal{D}, P) \) to denote the set of all partition subgraphs of \( \mathcal{D} \) induced by the elements of \( P \). Thus, \( \mathcal{P} = \{ P_1, P_2, \ldots, P_n \} \).
Each induced subgraph \( P_i \) will have its own induced subgraph transmitters and induced subgraph receivers. We sometimes refer to these as subgraph transmitters.
CONDENSED GRAPHS

(subgraph receivers) and distinguish them from transmitters (receivers) of \( D \), which we refer to as global transmitters (global receivers).

The set of transmitters of a partition subgraph, \( P_i \), is called the entry-points of \( P_i \). Similarly, the set of receivers of a partition subgraph, \( P_i \), is called the exit-points of \( P_i \).

**Definition 2.3.10 (Entry-points and Exit-points of an induced subgraph)**

The set of transmitters of a partition subgraph, \( P_i \), is called the entry-points of \( P_i \). This set may include both global transmitters and induced subgraph transmitters.

Similarly, the set of receivers of a partition subgraph, \( P_i \), is called the exit-points of \( P_i \), and this set may include both global receivers and induced subgraph receivers.

We define the input and output environments of an induced partition subgraph in a similar way as we did for vertices. Thus, if \( P = \{ P_1, P_2, \ldots, P_n \} \) is a partition of a DAG \( D = (V, E) \) and \( \text{cond}(D, P) = (P, E') \) then the input environment of an induced partition subgraph, \( P_i \), is \( \{ v : (v, p) \in E', v \in P_j, j \neq i \} \). That is, the set of immediate predecessors of \( P_i \). Also, the output environment of an induced partition subgraph, \( P_i \), is \( \{ v : (p, v) \in E', v \in P_j, j \neq i \} \), the set of immediate successors of \( P_i \).

We refer to the input environment and the output environment of \( P_i \), collectively, as the environment of \( P_i \).

2.4 Condensation Sequences

It is possible to repeatedly apply the condensation process and so give rise to a sequence of condensed graphs — a **condensation sequence**.

Condensation with respect to a partition which is not point-trivial leads to a graph with fewer vertices than the original, and so a condensation sequence derived from such partitions will be finite. Furthermore, if the process is repeated often enough, the last graph in the sequence will be a trivial DAG.

The length of a condensation sequence depends on the sequence of partitions used in its construction. Assuming the original graph is part of the sequence, this length will be at least 2, and will be at most equal to the size of the vertex set of the original graph. An illustration of a sequence of condensations is presented in Fig. 2.7.

The process of condensation involves two levels of abstraction: an abstraction on a vertex set, and an abstraction on an edge set. Consider again the "computer communications" in Example 2.3.3, the partitioning of machines on the basis of their physical location constitutes an abstraction on a vertex set. Similarly, the subsequent construction of edges between the vertices in the condensed graph constitutes an abstraction of the communication relationship. These new edges represent possible communications taking place between buildings. Furthermore, in the condensed graph, a communication between buildings may be made up of many one-to-one communications at the individual machine level. However, from the condensed graph alone it is not possible to determine which machines are communicating.
2.5 SEEDED DAGS

The figure shows a condensation sequence and the partitions chosen for each condensation. For convenience, we label each vertex in each graph of the sequence according to its corresponding element in the partition.

2.5 Seeded DAGs

As we have seen, condensed graphs are useful in highlighting structural properties by allowing us to abstract away certain details which may be obscuring the information we are seeking. However, by construction, the more abstract graph lacks the detail which may be needed at other times.

A condensation sequence represents the same information at many levels of abstraction, and different levels provide us with the desired amount of abstraction or detail. We now look at a structured way of moving between different graphs in this sequence. In effect we wish to be able to start at a specific graph in a condensation sequence, to get more abstract information by moving up through the sequence, and to get more specific information by moving down through that sequence.

Before this navigation can take place we need to preserve the information which is normally lost during the condensation process.

To do this we choose an appropriate partition \( P = \{ P_1, P_2, \ldots, P_n \} \) of a DAG \( \mathcal{D} \), and add new vertices to each of the induced subgraphs \( \mathcal{P} = IG(\mathcal{D}, P) \). This gives us a set of extended partition subgraphs, which we call seeded partition subgraphs and denote by \( \mathcal{P}^\theta = \{ \mathcal{P}_1^\theta, \mathcal{P}_2^\theta, \ldots, \mathcal{P}_n^\theta \} = IG(\mathcal{D}, P^\theta) \). Two vertices, called \( E \) (for Entry) and \( X \) (for eXit), are added to each partition subgraph \( \mathcal{P}_i \). Edges are drawn from the \( E \) vertex to every entry-point in \( \mathcal{P}_i \) in the following way: a single edge is drawn from \( E \) to \( u \) if \( u \) is a global transmitter, and \( \text{InDeg}(u, \mathcal{D}) \) edges are drawn from \( E \) to \( u \) if \( u \) is an induced subgraph transmitter. Thus, the \( E \) vertex becomes the only transmitter in the partition subgraph. Similarly, edges are drawn from every exit-point in \( \mathcal{P}_i \) to the \( X \) vertex in the following way: a single edge is drawn from \( v \) to \( X \) if \( v \) is a global receiver, and \( \text{OutDeg}(v, \mathcal{D}) \) edges are drawn from \( v \) to \( X \). In this way the \( X \) vertex becomes the only receiver in the partition subgraph.

A seeded partition subgraph may contain parallel lines between an \( E \) vertex and an entry-point and between an exit-point and an \( X \) vertex. These subgraphs thus form a net — although they contain no loops.

We can now connect the seeded partition subgraphs to form a seeded graph, \( \mathcal{D}^\theta \). To do this we employ a construction procedure similar to that used when determining edges of a condensed graph: an edge is drawn from the \( X \) vertex in the seeded partition subgraph, \( \mathcal{P}_i^\theta \), to the \( E \) vertex in the seeded partition subgraph, \( \mathcal{P}_j^\theta \), if and only if, in \( \mathcal{D} \) there is at least one edge from a vertex of \( \mathcal{P}_i \) to a vertex of \( \mathcal{P}_j \).
In the graph, $D^\oplus$, all transmitters will be $E$ vertices and all receivers will be $X$ vertices. Furthermore, the input environment of all internal $E$ vertices will be $X$ vertices and the output environment of all internal $X$ vertices will be $E$ vertices.

Vertices $E$ and $X$ act as the input environment and the output environment respectively of each seeded partition subgraph. The function of the $E$ vertices is to make concrete the abstract relationship on its inputs. In terms of the computer communications example (Example 2.3.3), an $E$ vertex would connect to those machines which are the target of a message. However, it does not identify the machine sending the message. This latter information is available at the input environment (i.e., the successor $X$ vertices) of the $E$ vertex. In contrast then, the function of an $X$ vertex is one of information hiding — representing concrete relationship information as a single abstract relationship.

Although the graph, $D^\oplus$, is in many ways different to the $\text{DAG}$, $D$, $\text{cond}(D^\oplus, P^\oplus)$ is indistinguishable from $\text{cond}(D, P)$ — they have the same structural properties. In particular, any and all parallel lines in $D^\oplus$ are removed by condensation so that $\text{cond}(D^\oplus, P^\oplus)$ is once more a $\text{DAG}$.

The following example illustrates the process of extending and subsequently condensing a $\text{DAG}$.

**Example 2.5.1** (Seeding and Condensing a DAG)

Let $D = (V = \{1, 2, 3, 4, 5\}, E = \{(1, 3), (2, 3), (3, 4), (3, 5)\})$ and $P = \{P_1, P_2, P_3\} = \{\{1, 2\}, \{3\}, \{4, 5\}\}$.

![Figure 2.8: Seeding and Condensing the DAG $D$](image)

Since there are three partition elements, there are three induced partition subgraphs:

- $P_1 = \text{IG}(D, P_1) = \{V_1 = \{1, 2\}, E_1 = \{\}\}$
- $P_2 = \text{IG}(D, P_2) = \{V_2 = \{3\}, E_2 = \{\}\}$
- $P_3 = \text{IG}(D, P_3) = \{V_3 = \{4, 5\}, E_3 = \{\}\}$

![Figure 2.9: The induced partition subgraphs. (a) $P_1$. (b) $P_2$. (c) $P_3$.](image)

These partition subgraphs are extended to give three seeded induced partition subgraphs:
2.6 Seeded Condensation Sequences

\( \mathcal{P}_1^\oplus = \text{IG}(\mathcal{D}, \mathcal{P}_1^\oplus) = (V_1 = \{E, 1, 2, x\}, E_1 = \{(E, 1), (E, 2), (1, x), (2, x)\}) \)
\( \mathcal{P}_2^\oplus = \text{IG}(\mathcal{D}, \mathcal{P}_2^\oplus) = (V_2 = \{E, 3, x\}, E_2 = \{(E, 3), (E, 3), (3, x), (3, x)\}) \)
\( \mathcal{P}_3^\oplus = \text{IG}(\mathcal{D}, \mathcal{P}_3^\oplus) = (V_3 = \{E, 4, 5, x\}, E_3 = \{(E, 4), (E, 5), (4, x), (5, x)\}) \)

Figure 2.10: The seeded induced partition subgraphs. (a) \( \mathcal{P}_1^\oplus \), (b) \( \mathcal{P}_2^\oplus \), (c) \( \mathcal{P}_3^\oplus \)

The seeded graph \( \mathcal{D}^\oplus \) formed by combining these subgraphs is illustrated in Fig. 2.11(a). Fig. 2.11(b) depicts \( \text{cond}(\mathcal{D}^\oplus, \mathcal{P}^\oplus) \). A similar DAG is constructed by \( \text{cond}(\mathcal{D}, P) \).

Figure 2.11: (a) The graph \( \mathcal{D}^\oplus \). Note that this graph is not a digraph since it contains parallel lines. (b) The graph, \( \text{cond}(\mathcal{D}^\oplus, \mathcal{P}^\oplus) \), is a DAG — all parallel lines have been removed by the condensation process.

2.6 Seeded Condensation Sequences

We can construct a condensation sequence of seeded DAGs — a seeded condensation sequence — which will not only be representative of information at different levels of abstraction, but, together with an appropriate navigation procedure, will facilitate free and meaningful movement through that information.

This navigation depends on being able to relate vertices from different graphs in the sequence. Movement through the sequence is facilitated by condensation (from the specific to the abstract) or by evaporation (from the abstract to the specific). We define an evaporation function (\( \text{evap} \)) which maps every condensed vertex of a graph \( \mathcal{D}^\oplus \) to the
E vertex of a seeded partition subgraph of another graph $C^*$ — which appears before $D^*$ in the sequence. If a vertex was not created by condensation (for example, the X and E vertices, and all vertices in the first graph of the sequence) then the evaporation function maps that vertex to itself.

**DEFINITION 2.6.1 (Evaporation )**

For $v \in \text{cond}(D^*, P^*)$.

- if $v$ is a condensed vertex $P^*_v$: $\text{evap}(v) \rightarrow E \in$ the vertex set of $P^*_v$
- if $v$ is not a condensed vertex : $\text{evap}(v) \rightarrow v$

The first element in a seeded condensation sequence is the initial seeded graph. This graph is condensed to form the second element in the sequence. If a third element is required, it is formed by seeding the second and subsequently condensing. Thus, by construction, every element of a seeded condensation sequence except the last will contain E and X vertices.

There will always be at least two elements in a seeded condensation sequence. Furthermore, the length of every such sequence is finite since the sequence construction will eventually produce a trivial DAG. We note that a sequence may be terminated before a trivial DAG is produced and that, in general, the sequence length will be determined by empirical considerations — given the constraints already mentioned.

Each element in a seeded condensation sequence is called an *horizontal graph*, or more concisely an H-graph.

As an example, consider Fig 2.12: each H-graph of the sequence is separated by a horizontal line, with the most abstract graph drawn at the top.

Now, to fulfill our prime motivation of being able to move freely around the same information represented at different levels of abstraction, we construct another graph from a seeded condensation sequence, which we call a *vertical graph*, or more concisely a V-graph.

We construct the V-graph by connecting the vertices in all H-graphs. Thus, the output environment of every vertex, $v$, in every H-graph ($v \neq X$) is replaced by the output environment of $\text{evap}(v)$; and the output environment of the X vertex in $P^*_v$ is replaced by the output environment of the vertex $P^*_v$.

The vertex set of a V-graph will thus consist of the union of the vertex sets of all the H-graphs in the seeded sequence.

We note the following:

- The transmitters and receivers of the most abstract graph in the sequence become the transmitters and receivers of the V-graph.
- Every vertex from the sequence appears once and only once.
- If vertex $v$ is a predecessor of vertex $w$ in any H-graph, then $v$ is a predecessor of $w$ in the V-graph.
- The V-graph corresponding to a seeded condensation sequence is unique.
2.7. SUMMARY

Figure 2.12: Depicting a CG sequence. Each H-graph is obtained by condensing its seeded predecessor in the sequence. Extending an H-graph with E and X vertices only takes place as a precursor to condensation; consequently the final H-graph in the sequence is not seeded.

- There is an equal number of E and X vertices in every V-graph.

Fig. 2.13 illustrates the V-graph corresponding to the seeded condensation sequence of Fig 2.12.

The arcs connecting seeded partition subgraphs in an H-graph are used for determining connections in the next level of a condensation sequence. In contrast, they are unimportant in the construction of the V-graph for that sequence. In this construction process, seeded partition subgraphs can be considered as independent of each other. To enforce this perspective we will call a seeded partition subgraph a definition graph (or sometimes, a definition, for short).

Every vertex in a V-graph will be called a condensed graph and denoted by CG. Some CGs called condensed vertices are associated with definitions by the evap function; others have no associated definitions and are called atomic CGs. The E and X vertices are examples of atomic CGs.

2.7 Summary

In this chapter we reviewed a number of concepts from the theory of directed graphs; these included the definition of a digraph, DAG, and net, and the process of condensation. Motivated by the desire to navigate through the same information represented at various levels of abstraction, we introduced the process of DAG seeding, and we constructed
Figure 2.13: (a) depicts the order in which the vertices of the seeded sequence are connected to form V-graph. Lines from the H-graphs which do not appear in the V-graph are shown dashed. In (b) the V-graph constructed in (a) is redrawn.
seeded condensation sequences. In a seeded graph, $E$ vertices convert abstract relationships to concrete relationships and $X$ vertices convert concrete relationships to abstract relationships. Each abstraction level in a condensation sequence is called an $H$-graph and every node in an $H$-graph is called a $CG$. To navigate between the various $H$-graphs in a condensation sequence we constructed a $V$-graph from the condensation sequence according to specific rules.

In the next chapter we shall see how condensation sequences can be used to represent algorithms, and also how they may be used as the structural basis for a model of computation.
Chapter 3

Computing With CGs

3.1 Representing Algorithms

An algorithm is a finite set of instructions performed in a particular order to execute a specific task. Graphs provide us with a natural way of expressing algorithms. Operations and data can be modeled by vertices, and sequencing constraints can be modeled by edges. Repetition can be expressed as a cycle in a cyclic graph or by self-reference in an acyclic graph. In addition, decisions can be modeled by the conditional execution of subgraphs. At least two different types of operator can be defined to implement decision making in a graph. The first of these is the branch operator (see [53] for example) and we call the second, the filter operator.

The branch operator (illustrated in Fig. 3.1) conditionally places input values into one of two disjoint subgraphs representing the branches of the decision. This type of operator is widely used in data-driven systems that employ a strict firing rule. Although the branch operator is effective, it has some disadvantages:

![Branch Operator Diagram]

Figure 3.1: The branch operator. The input is routed to either Subgraph A or to Subgraph B depending on the value of Bool.

- The two disjoint successor subgraphs of the branch operator are usually reconnected at the vertex representing the next logical operator in the algorithm after the
branch. This merging can give rise to safety problems as outlined in Chapter 1, Section 1.2.2.

- Delaying the evaluation of the instructions in one or other of the subgraphs until the Boolean expression has been evaluated may delay the overall computation.

In contrast, the filter operator (illustrated in Fig. 3.2) is executed after both branches of the decision and the Boolean expression have been evaluated; this allows these three subgraphs to be evaluated concurrently. When the result of the Boolean expression is calculated, the appropriate branch of the decision is chosen to be the result of the filter and the evaluation of the other branch can be ignored (or terminated). The filter operator does not give rise to safety problems. Furthermore, it does not delay the overall computation — provided that enough resources are available to execute all subgraphs concurrently. This operator is typically not used in availability-driven systems employing a strict firing rule since it cannot be used to model recursion under those circumstances (see Example 3.20). Any computing model employing a non-strict firing rule could however make use of the filter instruction. In these systems, only the Boolean expression and the appropriate decision branch need to be evaluated before the filter is executed.

![Diagram of the filter operator](image)

**Figure 3.2: The filter operator**

The filter operator facilitates speculative parallelism since both decision branches may be evaluated concurrently and only one result is ever required. When sufficient resources are available speculation can lead to faster execution times. However, speculation also has an associated disadvantage: when a speculative computation is determined to be irrelevant it must either be terminated, to avoid wasting computing resources, or it must be left run to completion — resulting in the production of remaining packet garbage.

As already mentioned, repetition can be modeled in acyclic graphs using self-reference. In particular, the hierarchical structure of a condensation sequence can be exploited to model self-reference in well-defined manner. Consider for example the condensation sequence in Fig. 3.3. The point $u$ represents the condensation of the definition subgraph, so, $evap(u) = \Sigma$ vertex in that definition. Now, by defining $evap(v)$ to be equal to $evap(u)$, we introduce the concept of self-reference.
3.1. REPRESENTING ALGORITHMS

Figure 3.3: Modeling recursion in a condensation sequence. The result of \( evap(v) \) is defined to be equal to \( evap(u) \). This association cannot be produced directly by the condensation process since the concept of self-reference belongs (in this case) to the domain of algorithm specification and not to the domain of hierarchical composition.

This definition does not change the structure of the definition, although it does alter the V-graph of the condensation sequence.

We choose to use the \texttt{filter} operator in the \( CG \) model primarily because it does not lead to the merging of output environments from distinct subgraphs. Consequently, every \( CG \) can be guaranteed to have a unique input environment. In Chapter 4 we shall also see how the speculative computation facilitated by the \texttt{filter} operator can be used to advantage without the introduction of the associated disadvantage listed above.

3.1.1 Ports and Nodes

Vertices may have a number of different incoming and outgoing edges; to distinguish between them we introduce the concept of a \textit{port}. Intuitively, a port is an entry or an exit point to or from a vertex of a graph. When modeling algorithms we define the number of input ports of a vertex \( v \) to be equal to the in-degree of \( v \). In contrast, we limit the number of output ports to one. In this way, vertices are used to model single-valued functions\(^1\). This decision does not limit the out-degree of a vertex which is used to indicate the number of destinations to which the result of the vertex is sent.

Since it is not common to speak of distinct entry and exit points of a vertex, we use the word \textit{node} when referring to a vertex in a condensation sequence being used to model algorithms. We will also use the term \textit{condensed node} in this context to mean the same as a condensed vertex.

We can even derive the concept of a node directly through the condensation of a graph of ports and a partition of these ports into operations. Input ports have an in-degree of 1 and an out-degree of 0, and output ports have an in-degree of 0 and an out-degree of at least 1. An example is given in Fig. 3.4. It is interesting to note that in this construction every node is enclosed by an E-X pair, forming a self-contained evaluation context.

In Fig. 3.5 we illustrate all the input ports of a \( CG \) node. There are three distinct kinds: one for each element in the computation triple. In general, there will be zero or many

\(^1\)This is a simplification, rather than a limitation, of the \( CG \) model. We see later that this decision also affects the number of operand ports of an \( X \) node.
Figure 3.4: Condensing a port graph to an operator graph. The partition chosen for this condensation is: \( P = \{+, -, \} \times \{\{i1, i2, o1\}, \{i3, i4, o2\}, \{i5, i6, o3\}\} \).

Operand input ports, one function input port, and one output environment input port. The single output port is not shown in this figure.

Figure 3.5: The CG node depicting input ports for operands, a function, and an output environment. To distinguish the operand from the function and output environment ports, we use a dashed line with a solid arrow head to identify edges entering the function port and a dashed line with a hollow arrow head to identify edges entering the output environment input port. Edges entering operand ports will always be drawn as solid lines.

When drawing CG nodes, we place the name of a function inside a node, but only when the function is bound to that node. Operands are drawn on incoming edges to operand ports, and an output environment, when bound to the node, will be depicted as a number of outgoing edges from the single output port of that node.

### 3.1.2 Algorithms and Condensation Sequences

The details of an algorithm can be derived through a process of hierarchical decomposition, or stepwise refinement: starting with an abstract description and refining the details as the structure is uncovered. In this approach, the levels of refinement are reminiscent of H-graphs in a condensation sequence — insofar as each represents a complete algorithm description at a particular level of abstraction. Furthermore, the refinement step is reminiscent of the \( evap \) function: taking us to a lower level of abstraction. In stepwise refinement, construction is driven by specification based on knowledge of the application domain, rather than abstraction based on the grouping together of parts as is
3.1. REPRESENTING ALGORITHMS

the case with condensation. Once the levels of refinement have been constructed however it is easy to see how a condensation sequence could be constructed to model them. A condensation sequence can reflect a complete algorithm design procedure in which every H-graph contains a description of the algorithm at a level of abstraction corresponding to the position of that H-graph in the sequence.

We can also approach the construction of a condensation sequence by constructing definitions (representing sub-algorithms) whose condensations are subsequently incorporated in other definitions. Thus, those definitions containing only atomic nodes are defined to be in the first H-graph of the sequence; definitions containing condensations of these are defined to be in the second H-graph in the sequence, and so on. This construction process has the advantage of allowing atomic nodes to appear directly in any H-graph without the need for performing a lot of graph-trivial condensations. These sequences are formed by hierarchical composition much like abstract data types or objects from an Object-Oriented analysis and design methodology.

We model algorithms by considering instructions to be composed of three essential elements that we collectively call a computation triple or more simply a triple. These elements are: an appropriate set of operands, a function which operates on those operands, and a destination, that is, a place to put the result of the operation. Every node in a condensation sequence (that is, every CG) is used to represent one or more elements of a computation triple. An instruction corresponds to a CG representing all elements of a triple and is constructed by relating sufficient CGs, each representing complementary triple elements. Relationships are made between different CGs through their input ports. Thus in the CG of an instruction, the operand element corresponds to CGs associated with its operand ports; the function corresponds to a CG associated with the function port; and the destination corresponds to a CG associated with the output environment input port. CG-to-input-port relationships can be created either statically or dynamically. A static association is made when a CG is related to a port before the algorithm is executed. We call these CGs constants. A CG can also be associated dynamically with an input port: Instructions create CGs containing less than three triple elements when they are executed, so a dynamic association is made between the CGs produced by the instruction execution and the input ports in the output environment of that instruction.

In general, the form of a computation is determined by the choice of association types made for each element of the computation triple. Static associations determine the form of the algorithm and dynamic associations determine the sequencing constraints in an execution of that algorithm. We can illustrate the influence of association types by looking at the cases in which only one element of the triple takes part in a dynamic association:

- If all functions and output environments are constants, the sequencing of instructions is determined by the dynamic association of operands to operand ports. This scenario is analogous to the data-driven computing model. When sufficient operands have been supplied to all transmitters, all other operands are dynamically created by the execution of instructions and their creation drives the computation. Fig. 3.6 illustrates the dataflow graph corresponding to the expression \((3 + 4) + 5\).
Figure 3.6: A CG graph of dynamic operands

- If all operands and output environments are constants, the sequencing constraints are determined by the dynamic association of functions to function ports. A graph of this situation using our simple expression is illustrated in Fig. 3.7(a). In this case, functions are not produced as part of the computation: a sufficient number of functions must be supplied by some external mechanism to all CGs before the execution can be completed. In contrast, Fig. 3.7(b) illustrates a graph in which functions are created dynamically as part of the computation. The algorithm utilizes a currying technique in which single or zero arity functions are propagated. This graph and that in Fig. 3.7(a) both represent availability-driven computations: in this case, the availability of functions drives the computation. In addition, all data are constant and are not changed by the computation. In Chapter 5 we will revisit computations which are sequenced by function availability after showing how mutable CGs are represented in the CG model. At that point we shall also see how this special case of the CG model corresponds to control-driven computations.

Figure 3.7: A CG graph of dynamic operators

- If all operands and functions are constants, the sequencing constraints are determined by the dynamic association of output environments to output environment input ports. In this case, every operand is a constant and some operands correspond to subgraphs which have their own operand sets. We call these subgraph constants. The dynamic association of output environments to output environment input ports is made by grafting subgraph constants to operand ports. This process is outlined in Sections 3.1.3 and 3.2.2. Fig. 3.8 illustrates the type of graph that would result for our simple expression. Every node in this graph has no output environment and so lacks one essential ingredient to make an instruction.

In general, an algorithm expressed as a condensation sequence can incorporate — in the same definition — any or all of the sequencing constraints outlined above. Availability-
3.1. REPRESENTING ALGORITHMS

3.1.3 Stemming and Grafting

We use the processes of stemming and grafting to change between CG association types in an algorithm.

**DEFINITION 3.1.1 (Stemming Process)**

The stemming process replaces an edge connecting two CGs by a subgraph constant. Thus, a potentially dynamic association is changed into a static association.

**DEFINITION 3.1.2 (Grafting Process)**

The grafting process connects a subgraph constant to an operand port by giving the constant an output environment consisting of that port. Thus, a static association is removed so that a subsequent execution will result in a dynamic association.

In Fig. 3.9(a) we illustrate an availability-driven algorithm of the Boolean expression: \((x \text{ OR } (y \text{ AND } z))\). Fig. 3.9(b) shows the same graph after the stemming process is applied to the AND node. A subgraph constant is created corresponding to the subexpression \((y \text{ AND } z)\) and this subgraph is statically associated with the former output environment of the AND node. We say that the AND node is **stemmed** and will refer to such nodes as **stemmed nodes**.

Grafting is the opposite process to stemming. In going from Fig. 3.9(b) to Fig. 3.9(a) we say that the AND node is **grafted** onto an operand port of the OR node. We refer to such nodes as **grafted nodes**.

In our terminology, all nodes in an availability-driven graph which are not involved in the sequencing of the computation are **grafted nodes**. Also, every node in a coercion-driven graph is a **stemmed node**.
Figure 3.9: Example of graph stemming. Part (a) shows the graph before stemming and Part (b) shows it after stemming. We refer to the AND node as a stemmed node since it has lost some (in this case all) of its outgoing edges.

(a) 

(b)

Figure 3.10: Stemming which does not result in an unconnected graph

In the example just considered, the graph is broken into two disjoint subgraphs by the stemming process. This may not always be the case. In Fig. 3.10 the stemming of Node b does not result in an unconnected graph.

In the examples seen so far, the stemming process resulted in the removal of all outgoing edges from a node. Such nodes are said to completely stemmed. The stemming process could equally well be applied to remove only some edges from a node. For example, in Fig. 3.11(b), Node b from Fig. 3.11(a) is partially stemmed and is statically associated with Node d.

(a) 

(b) 

(c)

Figure 3.11: Partial Stemming

In Fig. 3.11(c) the subgraph is replicated to distinguish the subgraph constant from the rest of the graph. In doing this we have not altered the logical meaning of the graph: Fig. 3.11(c) and Fig. 3.11(b) are logically equivalent.

Grafting the subgraph constant of Fig. 3.11(c) back onto the operand port of Node d yields the graph depicted in Fig. 3.12.
3.2 The CG Execution Model

We first give a general overview and then look at the specific details of the CG execution model.

CGs are the nodes of a condensation sequence and come in two varieties: incomplete CGs, representing less than three elements of a triple, which are capable of combining with other incomplete CGs; and complete CGs, representing all three elements of their computation triple, which are not capable of combining further. Computation triples are formed by combining incomplete CGs with complementary triple elements. To facilitate this combining process some of the incomplete CGs are considered to be stationary while others are mobile, and flow on the edges of the stationary CGs. When a mobile CG arrives at a stationary node, it becomes bound to that node; and when sufficient have arrived to form a computation triple the node constitutes a complete CG.

The subsequent execution of that node, which occurs only once for that triple combination, produces an incomplete CG which is replicated appropriately and flows on the output environment of the node. After execution, the node lies dormant and will be discarded when the graph of which it is part is deallocated; alternatively, it may be reactivated more than once before the deallocation takes place. A complete CG is reactivated by making it once more incomplete in a process we call deconstruction. This process discards one or more elements from the triple of a complete CG and so leaves the remaining elements free to combine in the future with other incomplete CGs.

We define the terms reducible node and fireable node to help describe the various stages in the construction of a computation triple.

**Definition 3.2.1 (Reducible Node)**

A node is reducible if there is a CG associated with its function port and with each of its operand ports.

A CG may be associated with the input port of a node in one of three ways: it may be a constant placed there as part of the algorithm description; it may be placed there to instantiate the computation; or it may be created by the execution of a complete CG.

\[ A \text{ chemistry of computation perhaps?} \]
A node must be reducible before it is complete; we say that a node is reduction-strict if all operands must be present before the node is reducible. At first this may seem to be at variance with the idea of a coercion-driven execution. However, to say that a node is reduction-strict does not necessarily imply that all operands of a function need to be evaluated before that function can be executed. If a function requires an operand to be in normal form before the function itself can be evaluated, the function is strict in that operand. We will say that an operand port is function-strict if the function element of the node is strict in that operand. Thus, we define two levels of "strictness": reduction-strictness which tells us how many operands are needed to make a node reducible, and function-strictness which tells us which of those operands must be in normal form before the function associated with the node can be executed. Every node in the CG model is reduction strict. Function strictness, however, depends on the particular operator associated with each node.

A reducible node must be complete before it can be executed. To conform to a common term used in the literature we will use the terms fireable and complete interchangeably.

**Definition 3.2.2 (Fireable Node)**

A node is fireable when it is complete. That is, when it is reducible and has an output environment.

A fireable node may or may not be suitable for execution. It can be executed, or fired, only when all of its operands are in an appropriate form. For example, an arithmetic operator may require its operands to be simple data values. If it is made reducible with a subgraph constant, the arithmetic expression cannot be executed until the constant is evaluated. In this case, the execution of the subgraph constant is required and some form of coercion is needed to cause its evaluation.

When a node is fired, a CG is dynamically associated with the operand ports of the nodes constituting its immediate successors in the V-graph of the condensation sequence. In effect, the V-graph of the condensation sequence representing the algorithm is dynamically constructed by the execution of that algorithm. For all atomic nodes (except the X node) these successors are determined by their respective output environments. The immediate successors of condensed nodes are determined by the evap function; and those of an X node are identified by the output environment of the condensed node whose evaluation invoked the definition containing the X node.

Operationally, when a condensed node is fired a new instantiation of its definition is created and becomes part of the V-graph. The operand CGs of the condensed node are associated with the E node of this instantiation and the CG of its output environment is associated with the X node of the instantiation. It is assumed that condensed nodes are constructed so that their execution supplies sufficient operands for their definition to produce a result. The invocation of a definition by the execution of a condensed node is illustrated in Fig. 3.13.
3.2. THE CG EXECUTION MODEL

Figure 3.13: Firing a Condensed Node. The CGs representing the operands of the condensed node u are associated with the operand parts of the E node in the definition graph of u; and the CG representing the output environment of u is associated with the output environment input part of the X node in the definition graph of u -- this latter CG is depicted by a node containing a hollow arrow head.

The operand parts of all condensed nodes and all E nodes are non-function-strict; the only operand port\(^2\) of the X node is function-strict; and only the Boolean operand port of the filter node is function-strict.

3.2.1 The V-graph

The execution of nodes in a condensation sequence proceeds by associating more and more CGs with the input parts of other CGs in its V-graph. When a condensed node is executed, the V-graph is extended to include the definition of that node. In a recursive algorithm, the size of the V-graph is infinite. However, in a well-formed algorithm the actual size of the V-graph constructed during an execution of that algorithm will be finite but may differ from one execution to another.

If nodes in the condensation sequence have been stemmed then the construction of the V-graph may begin from many places in the sequence simultaneously, and connections between nodes will be drawn from successor to predecessor nodes (in an availability-driven computation) or from predecessor to successor nodes (in a coercion-driven computation). Thus, depending on the association types chosen in the design of the algorithm, the V-graph may be constructed "top-down", "bottom-up", "inside-out" or in any combination of these directions simultaneously. The V-graph is defined to be constructed when a result from the algorithm execution is produced and this may occur before all nodes in all H-graphs have been connected. Subgraph constant operands are associated with nodes lower in the condensation sequence until their evaluation is needed and at that point they are grafted onto an appropriate node and become part of the V-graph. If the evaluation of a subgraph constant is never needed, the V-graph is consequentially smaller.

The aim of the algorithm designer should be to construct the condensation sequence in such a manner that the length of the longest path through the V-graph is minimized. This design should be tempered with a knowledge of the resources available to execute the algorithm so that an effective balance can be struck between the amount of processing to be performed and the time needed to perform that processing. This is a process in

---

\(^2\)We consider the X node to have only one operand since we are interested in modeling single-valued functions.
which the algorithm designer endeavours to match the parallelism of the algorithm to the parallelism of the machine on which it is to run.

3.2.2 Subgraph Constant Evaluation

By fulfilling the reduction strictness criterion, subgraph constant operands allow their associated nodes to fire earlier than they would had they to wait for the constants themselves to be evaluated. This only works, however, if the operand port associated with the subgraph constant is non-function-strict. In particular, when subgraph constants are operands to condensed nodes, the execution of the definition graphs of these nodes can begin immediately. In an optimum execution of an algorithm, appropriately chosen subgraph constants may be consumed in the evaluation of filter nodes and so may never have to be evaluated. In this way the computing resources needed to execute the algorithm are reduced. Inevitably, however, the evaluation of some subgraph constants will be needed when they are associated with the function-strict operand ports.

The evaluation of a subgraph constant is triggered by the attempted firing of the node to which it is an operand. At that point the subgraph constant is grafted onto the function-strict operand port of that node. This makes the subgraph constant fireable since by definition it is already reducible. The grafting process constructs a CG to represent the appropriate output environment and associates it with the output environment input port of the stemmed node in the subgraph constant. Finally, the node is deconstructed to remove the subgraph constant from its operand port, thus making the port receptive to the dynamically created CG resulting from the subgraph constant evaluation.

As illustrated in Fig. 3.11, the same subgraph constant may be replicated by stemming. In addition, when a subgraph constant passes down through the V-graph it may be replicated many times and sent to different destinations. All of these subgraph constants, if evaluated, would yield the same result: each is just an unnormalized copy of the same reducible expression.

Manipulating multiple copies of the same subgraph can be expensive and wasteful. To optimize the treatment of a subgraph constant we create a sub-definition corresponding to that subgraph constant, and propagate condensed nodes of this sub-definition rather than the subgraph itself. Condensed nodes of a sub-definition differ from condensed nodes of a definition in that they contain no operand ports. These are not needed, since its sub-definition is reduced when the definition (of which it is a part) is reduced. Sub-definitions will only be fireable when their condensed nodes are grafted. We illustrate these concepts by example in Fig. 3.14.

A condensed node of a subgraph constant is evaluated in a similar manner to every other condensed node. However, in this case the process is less elaborate: there is no need for the creation of a new instantiation of the sub-definition, since no new operands are associated with the sub-definition. Thus, the same sub-definition is utilized to evaluate every one of its condensed nodes. The net effect of evaluating these condensed nodes is the construction of an output environment as outlined before and the association of that output environment with the X node of its sub-definition.

Many condensed nodes of a sub-definition may have been grafted before its X node is
3.2. **THE CG EXECUTION MODEL**

![Diagram](image)

Figure 3.14: Optimizing subgraph constant evaluation using sub-definitions. Part (b) shows the sub-definition corresponding to the subgraph constant $S$. This sub-definition is reduced when the definition of $u$ is reduced. The condensed node $C$ of the sub-definition is propagated through the $V$-graph until it is either consumed or grafted.

Reducible. Since the execution of a node makes no distinction between various destinations in an output environment (the same resulting $CG$ is simply sent to all destinations) we can safely accommodate a concatenation of output environments resulting from the simultaneous association of more than one output environment with the output environment input port of a node. (This is the only input port of a $CG$ node for which this type of behaviour is allowed.) As a consequence the eventual execution of the $X$ node may send a result to many destinations — simultaneously satisfying the execution of many condensed nodes.

When the $X$ node of a subgraph constant has fired it is deconstructed by removing its output environment. A subsequent invocation of its sub-definition is then possible. Once the $X$ node is reducible all evaluation of condensed nodes of the subgraph constant degenerate into the firing of that $X$ node.

We note that, since the $X$ node is defined to be function-strict, no subgraph constant can be associated with nodes higher up in the condensation sequence.

For simplicity, we will sometimes refer to condensed nodes of sub-definitions as **subgraph constants** and restrict the term **condensed node** to condensed nodes of definitions only.

### 3.2.3 **CG Node Deconstruction**

A complete $CG$ can be turned into an incomplete $CG$ by the removal of one or more elements from its triple. This process is called **deconstruction** and it is associated with distinct, well-defined events in the execution of a $CG$. Deconstruction is triggered by the firing of a complete node. It may forestall the firing of the node, as is the case in a subgraph constant evaluation; or it may be applied after the node is fired, to effect a form of graph rewriting. Thus:
1. In the evaluation of subgraph constants, two nodes are deconstructed. The first is the node to which the constant is grafted. It is deconstructed to remove the constant from its operand port so that the result of the subgraph constant evaluation can be associated with that port. In addition, the X node of the subgraph constant sub-definition is deconstructed to remove its output environment, enabling it to be fired again to service future subgraph constant evaluations. In the latter case, deconstruction is needed purely because the same sub-definition is shared between many subgraph constants. In contrast, the evaluation of every condensed node refers to a unique instantiation of its associated definition and so no deconstruction is required by the X nodes in definitions.

2. When a condensed node is evaluated the function element of that node can be removed and replaced with the result of the evaluation. Thus, for example, the function element can be replaced by a zero-arity function. This scheme can be applied to condensed nodes of subgraph constants and also to condensed nodes of definitions. At this point in the development of the $CG$ execution model the advantage of graph rewriting is unclear since the same instance of a node is never executed more than once. However, in Chapter 4 Section 4.8 we see how this rewriting can be useful in the context of backtracking.

We note that the need for deconstruction in sub-definitions is motivated by the sharing of that sub-definition by many subgraph constants. In Chapter 5 we will see a further need for associating the deconstruction process with specially designed functions for sharing stored data and for evaluating those functions in a control-driven manner.

### 3.2.4 Mutual Reduction

In general, the operands of a node are input parameters which reduce the function element of that node. Functions may be atomic and so can operate directly on their parameters (if they are in the required form); or they may be condensed nodes, in which case the parameters are associated with other nodes of function definition graph when the condensed node is fired. We say that the operands are used to populate the function definition graph.

As we shall see in Chapter 5, the $CG$ computing model also uses condensation sequences to represent structured data. As such, a $CG$ execution may contain condensed nodes of structured data definitions. These are processed by the dynamic association of functions with the function input ports of its nodes --- similar to the situation depicted in Fig. 3.7. Thus, in the execution of a condensed node of a structured data definition, the definition is populated by functions. Rather than adopting two different computing models --- one in which data populate function definitions; and the other in which functions populate data definitions --- we define a set of rules for mutual reduction which facilitates both situations. These rules can be expressed as follows:

1. If both the operand and the operator are condensed nodes then the operand populates the definition of the function.
3.2. **The CG Execution Model**

2. If the operand is atomic it is used to populate the definition of the function.

3. If an operand is a condensed node and the function is atomic then the function populates the definition of the operand.

The X node is the only CG that is not allowed to take part in a mutual reduction. We shall see in Chapter 5 that E nodes are mutually reduced in the dynamic creation of finite data structures.

### 3.2.5 Some Examples

We now consider some simple examples to illustrate some of the ideas presented so far.

**Example 3.2.3** (Executing a system of functions)

Fig. 3.15 illustrates a condensation sequence which represents the following system of functions:

- \( F(x, y) :: C(x, B(y, A(2))) \)
- \( A(x) :: x + x \)
- \( B(r, s) :: \text{if } r \text{ then } 1 \text{ else } s \)
- \( C(p, q) :: \text{if } p \text{ then } 2 \text{ else } q \)

The function \( F \) accepts two Boolean parameters and returns an integer. The dummy parameters in these functions are used to determine the connectivity in the definition graphs of Fig. 3.15.

![Definition Graph of F](image)

![Definition Graph of A](image)

![Definition Graph of B](image)

![Definition Graph of C](image)

**Figure 3.15**: *Definition graphs of some simple functions*

Node A is fireable immediately and can be executed regardless of whether Node E is ever executed. A profile of the execution of Node A is illustrated in Fig. 3.16.
Figure 3.16: (a) In the definition of Node A, operand 2 is associated with the operand port of the + node and the output environment input port of the X node is associated with the output environment of Node A. (b) When the + node is executed, two copies of its operand are made and associated with the input ports of the (+) node. (c) The (+) node executes to associate data value 4 with the operand port of the X node. (d) The execution of the X node associates data value 4 with the output environment of Node A.

At this point, no other node is reducible so the execution stops. Nothing more happens until the Node F is made fireable, and when F is subsequently executed, all the CGs in the condensation sequence will be executed.

We can avoid executing some nodes by stemming Node B. This is depicted in Fig. 3.17.

Figure 3.17: Stemming of Node B in the graph of some simple functions

As before, the execution of Node A takes place immediately. When Node F is executed, and when the filter node in the definition of C becomes fireable, it will execute to associate either the data value 2, or the subgraph constant B, with its outgoing edges (Fig. 3.18). In the former case, the execution of Node B is avoided and in the latter case, the subsequent firing of the X node in the definition of C will graft Node B onto the X node.

Thus, in contrast to the previous representation, Node B is only executed if the second parameter to Node F has the Boolean value FALSE.

We can stem Node A in Fig. 3.17 and possibly avoid its execution. In the graph of Fig. 3.19, neither Node A nor Node B is evaluated if F is reduced as
3.2. THE CG EXECUTION MODEL

Figure 3.18: Executing Node C in the graph of some simple functions

F(FALSE, FALSE).

Figure 3.19: Stemming of Node A in the graph of some simple functions

EXAMPLE 3.2.4 (Calculating factorial)

We consider the evaluation of the factorial function (n!) as an example of a recursive algorithm. Factorial(n) is defined recursively as follows:

\[ n! :: \begin{cases} 1 & \text{if } n=1 \\ n \times (n-1)! & \text{else} \end{cases} \]

A first representation of this algorithm is shown in the condensation sequence depicted in Fig. 3.20.

Figure 3.20: An availability-driven version of Factorial(n)

This representation is completely availability-driven. As a consequence, neither the filter node nor the (*) node in any recursive invocation will ever be reducible: there is never a CG associated with the output environment of the factorial node. In other words, neither the filter node nor the (*) node will ever appear in the construction of the V-graph of this sequence; and so the algorithm can never terminate. We can make the algorithm terminate by
stemming the (*) node and constructing a subgraph constant operand to the filter node. The filter node can then be fired as soon as its Boolean operand is available.

Figure 3.21: The Factorial(n) graph with stemmed (*) node

In the recursive step, the subgraph constant is the result of executing the filter node and it is associated with the X node. When the X node attempts to fire, the subgraph constant is grafted onto its operand port, making the (*) node fireable and so determining a result for that recursive level. The algorithm represented in Fig. 3.21 contains an unexpected feature: the (1) node is immediately fireable upon every recursive invocation; this can result in a speculative computation. The speculation can be avoided by stemming the (1) node, i.e., by making it a subgraph constant of the (*) node, as illustrated in Fig. 3.22.

Figure 3.22: The Factorial(n) graph with stemmed (1) node

The V-graph corresponding to the execution of Factorial(2) is illustrated in Fig. 3.23.

3.3 Summary

Condensation sequences can be used to represent algorithms and even algorithm design strategies. Successive levels of abstraction can be provided by successive H-graphs in the
Figure 3.23: The V-graph of Factorial(2) resulting from the execution of the graph in Fig. 3.22. When the (1) node in H-graph 2 is executed, its definition in H-graph 1 is populated with the data value 2. The (*) node in H-graph 1 is the result of the execution of the filter node and is grafted onto the operand port of the X node. Attempting to fire the (*) node in H-graph 1 results in the (1) node being grafted onto its second operand port. The (1) node is now fireable and executes to create a new instance of its definition (H-graph 0) and to populate it with the data value 1. The filter node in H-graph 0 fires to consume its subgraph constant operand and to produce the data value 1 as a result. This value is associated with the second operand port of the (*) node in H-graph 1 by the X node in H-graph 0. This (*) node can now be fired and the resulting data value is associated via the X node in H-graph 1 with the output environment of the (1) node in H-graph 2.
condensation sequence and self-reference can be used to model iteration and recursion.

The purpose of an algorithm is to specify instructions and the manner in which they are combined and sequenced. In the CG model, instructions are composed of three essential ingredients, together called a triple. All nodes in a condensation sequence (that is all CGs) are capable of representing the elements of a triple. Those CGs which have all three elements are called complete, and represent instructions. When complete CGs fire they produce incomplete CGs containing less than three triple elements. The CG execution model is based on the association of incomplete CGs containing complementary triple elements to form instructions. Two association types were outlined and exploited to implement different sequencing constraints. As the execution of an algorithm proceeds the V-graph of its condensation sequence is dynamically constructed — possibly from many locations in the condensation sequence simultaneously.

In the next chapter we shall develop a machine model for the construction and execution of computation triples, and we shall also examine the execution profiles of a number of example algorithms.
Chapter 4

Machine Model

4.1 Overview

We now present a $\mathcal{CG}$ machine model which is based on the construction and execution of computation triples. For the moment we assume that the function element of each triple is constant and so we will only consider computations which are driven by operand and output environment elements. We look at computations driven by function elements in Chapter 5. The machine as described has been simulated and results of sample simulations are presented. Furthermore, it is shown that availability-driven and coercion-driven executions constitute special cases of the execution mechanism described here.

4.2 The $\mathcal{CG}$ Machine Model

Our idealized machine is shown in Fig 4.1. It consists of a Triple Manager (TM) connected via a communication network to Ancillary Processors (APs).

![Network Diagram]

Figure 4.1: The $\mathcal{CG}$ idealized machine

The task of the TM is to construct triples — which constitute the instructions of the $\mathcal{CG}$ machine. Some of these instructions are executed locally in the TM and their only effect is to move $\mathcal{CG}$s around in memory. These $\mathcal{CG}$-moving instructions correspond to the — by now familiar — nodes which we have been using in the construction of condensation sequences: the $E$ nodes, $X$ nodes, $\mathcal{CG}$-type nodes and condensed nodes. The actions of these instructions are independent of the algorithm in which they occur, and together they constitute a minimal instruction set for the $\mathcal{CG}$ machine. We refer to them collectively as TM-instructions. Using these instructions only, it is possible to animate the execution of any $\mathcal{CG}$ completely within the confines of the TM (this can be an interesting way of
examining execution profiles as seen in Section 4.7.2.). The APs perform any and all of the value-transforming computations of an algorithm. They are the CG machine's only means for transforming operand data into result data.

The TM uses the network to send instruction tokens to the APs. These tokens contain three components: operand data, an operation identification, and a destination to which the result data are to be sent. An AP can be arbitrarily complex. It may be a functional unit capable of floating-point arithmetic, a microprocessor capable of executing sequential programs, or another parallel machine — possibly even another CG machine. In general, an AP is an independent computing engine: it simply transforms operand data into result data and transmits them over the network.

The three components of an instruction token mirror the elements of a computation triple. However, only CGs which correspond to simple data values are ever included on an instruction token. This is a design decision which maintains the distinction between the TM and the APs and also has the effect of rendering all APs functionally equivalent in all operands. Thus, rather than placing a CG representing a non-simple data value (a subgraph constant, for example) in an instruction token, the TM must first evaluate that CG.

Result data from the APs are packaged into result tokens and sent to the TM via the network. Each result token consists of a single result datum and an output environment copied from the instruction token. In the TM, a CG representing this datum is replicated and placed appropriately on the output environment.

We see that the association of CGs to the input ports of nodes in other CGs is achieved either by moving CGs around in memory (as is the case when TM-instructions are executed) or by moving newly created CGs to appropriate destinations. Operationally, we say that CG associations are achieved by letting CGs "flow" on the edges of other CGs.

Fig. 4.2 shows the TM in more detail. It is composed mainly of memory, but also contains an automaton which we call the Triple Constructor Process (TCP). The definition of graph memory is used to hold the definitions of the condensation sequence. The TCP copies these definitions appropriately to the V-graph memory to dynamically extend the V-graph as the computation proceeds. The V-graph memory contains a representation of the V-graph and so reflects the current state of the computation. The triple list contains all fireable CGs and it is rebuilt anew in each step by the TCP.

The memory in the TM is updated by the actions of the TCP which are performed repeatedly as follows:

1. CGs representing result data from the network are placed into the V-graph as specified by the output environment in the result token.

2. Search the V-graph for fireable nodes and add them to the triple list.

3. Process each CG in the triple list:

\footnotesize

1 Many alternative and potentially useful CG architectures are possible. For example, we could construct one in which an "ancillary" processor (the name is now somewhat of a misnomer) would recognize CG operand data representing complex data structures and would process them appropriately. In Chapter 5, we outline another approach which preserves the distinction between the TM and APs.

2 Following a decision made in Chapter 3 we restrict our consideration to single-valued functions only.
4.2. THE CG MACHINE MODEL

Figure 4.2: The Triple Manager

(a) If the CG represents a TM-instruction, execute it (see Section 4.2.1).

(b) If the CG's function element is an ancillary operation and all the operand data are simple data values, make an instruction token and dispatch it to an APs.

(c) If the CG's function element is an ancillary operation and at least one operand is a subgraph constant, remove the CG from the triple list and graft the subgraph constant onto the V-graph at the appropriate place.

4.2.1 The TM-Instructions

TM-instructions are executed by the TCP; no instruction token is created and no network communication is necessary. We now look at the action of each of these instructions in turn.

The Condensed Node

From the execution model described in Chapter 3, Section 3.2, we know that firing a condensed node extends the V-graph by inserting the appropriate definition between the condensed node and its output environment. In this way, operands of the condensed node can be passed to the definition and results can be passed, via its X node, to the output environment of the condensed node.

In the CG machine model, the execution of a condensed node causes an Activation Frame (AF) to be allocated in the V-graph memory. This is a block of contiguous memory used to hold a copy of the condensed node's definition. When the AF has been allocated, the operand data of the condensed node are copied to the output environment of the X node in the definition and the definition is populated with operands. Finally, to facilitate the communication of results from the subgraph, the X node receives a copy of the condensed node's output environment. In this way the output environment of the X node is dynamically determined at execution time, and the X node can be fired as soon as it is reducible.

A representation of the V-graph memory is shown in Fig. 4.3. Operationally, this is a dynamic tree structure.

1Note that in this scheme the firing of the X node is avoided.
Figure 4.3: Activation Frames in the V-graph. The V-graph changes dynamically as the computation proceeds; AFs are added (allocated) as condensed nodes are invoked and removed (deallocated) as X nodes are executed.

The E Node

In general, we can avoid executing the E node; its output environment can be used by the TCP to directly populate its definition. Under these circumstances, the E node is not reduced and so does not fire. This is an optimization which results in a saving of one sequential instruction execution per subgraph invocation. We return to the question of E node execution in Chapter 5.

The filter Node

The filter node executes by moving CGs from one location to another within a single AF. When it is fired, either its then or its else operand is copied unchanged to its output environment by the TCP.

The X Node

The X node is strict in its single operand. When it is fired its operand is simply copied to its output environment. This will be a location in a parent AF. The execution of X nodes gives the TCP an opportunity to deallocate memory. Thus, for example, when the X node in a subgraph, D, is fired, its associated AF and all other AFs created by the execution of condensed nodes in D are deallocated. In other words, once a result has

\footnote{Remember, we decided to model single-valued functions only.}
been determined by an AF, all other activity associated with that AF ceases to be relevant and can be terminated.

The decision to make the operand port of the X node function-strict prohibits subgraph constants being passed up through the V-graph and results in a simple memory management policy. The alternative would lead to a much more complicated memory management policy since AFs could not be deallocated until all subgraph constants are evaluated. In general, this cannot be determined easily.

### 4.2.2 Grafting in the TM

The evaluation of a subgraph constant is required if it is associated with a function-strict port. The TCP reacts to this situation by constructing an output environment and associating it with the output environment input port of the X node in the sub-definition. Thus, the subgraph constant is effectively grafted onto the function-strict port.

If other instances of the subgraph constant are grafted before the X node in the sub-definition has fired, the TCP simply concatenates the new output environments to the X node.

After firing, the X node is deconstructed by the TCP to remove its output environment. This facilitates a further firing of the X node to service the subsequent grafting of its subgraph constants.

The X node will remain reducible and will be fireable when an output environment is supplied — when another associated subgraph constant is grafted to a strict node. After execution, the X node is once more deconstructed. These actions of grafting, firing, and deconstructing continue until all instances of the subgraph constant have been reduced.

It is interesting to note at this point that the arrival of output environments at the X node can be regarded as a request for data — equivalent to a memory read request. We return to this point again in Chapter 5 when we shall discuss stored data. As the execution of an algorithm proceeds, the evaluation of certain nodes will degrade into memory read operations, and the flow of output environments will dominate that part of the computation. Consider, for example, Fig. 4.4.

![Figure 4.4: A “lazy chain” of subgraph constants](image)

If the subgraph constant C is sent to many locations in the definition of D, then the first evaluation of C will trigger the evaluation of the complete “lazy chain” C → B → A. This evaluation will consume all the subgraph constants in the chain and replace them with appropriate data values.

Subsequent requests for the evaluation of C will result in the firing of C’s X node only: the previous parts of the chain will have “decayed” to the normal form now present in C’s operand set. This scheme is reminiscent of graph-rewriting.
4.3 A $CG$ Language

A simple nested expression language has been developed for textually describing $CG$s. In this language all condensed nodes and definition names are represented as uppercase strings. Lowercase strings represent either basic ancillary operations (these depend on particular implementations of the $CG$ machine) or parameters. In this language, the parameters are used simply as a mechanism for connecting nodes together. For simplicity, a prefix notation is adopted in which all parameters are enclosed in parentheses.

The filter node is the only TM-instruction which appears explicitly in a $CG$ program; the others are automatically constructed by the compiler. We write $\text{filter}(B,E,F)$ to mean $if \ B \ then \ E \ else \ F$.

All other nodes are taken to correspond to ancillary operators and are expected to be found in the computation environment when the $CG$ program is executed. In the $CG$ simulator the ancillary operators correspond to functions written in the C language. These are provided by the programmer and are linked to the simulator by the $CG$ compiler.

The programmer can explicitly stem a node by enclosing it in braces: "{""}"". This has the effect of creating an anonymous subgraph constant consisting of that one node. This provides a simple syntactic mechanism for specifying static associations. Dynamic associations are assumed by default.

4.4 Example Simulations

In this and subsequent sections we shall concentrate mainly on the execution of one example program. This will serve to illustrate the wide diversity of execution profiles which can be generated by making small changes to node association types. Limiting ourselves to one example, at this point also provides continuity between this, and the sections to come. The example we choose is a recursive algorithm for the calculation of powers of 2. An initial program is presented in Code Segment 4.4.1.

**CODE SEGMENT 4.4.1 (2^x, non-terminating)**

```
POF2(x) := filter(eq(x,0), 1,
    filter(even(x),
        mul(POF2(div(x,2)), POF2(div(x,2))),
        mul(2, POF2(sub(x,1)))
    )
)
```

This program as written will not terminate since the filter nodes never become reducible. An execution of this algorithm will simply continue to recurse until the machine runs out of memory.

We can make this program terminate in a number of different ways. A simple way is to arrange for the outer filter node to be reducible as soon as its Boolean is evaluated. This is done by stemming the the inner filter node thereby turning it into a subgraph
constant. Syntactically, this is expressed by enclosing the inner filter node in braces as shown in Code Segment 4.4.2.

**CODE SEGMENT 4.4.2** (2*, with stemmed filter node)

\[
P_{OF2}(z) \leftarrow \text{filter}(\text{eq}(z, 0), 1, \{\text{filter}(\text{even}(x)),
\quad \text{mul}(P_{OF2}(\text{div}(z, 2)), P_{OF2}(\text{div}(z, 2))),
\quad \text{mul}(2, P_{OF2}(\text{sub}(x, 1)))
\})
\]

The definition corresponding to this program is illustrated in Fig. 4.5, and the execution profile for this graph, with \( z = 10 \), is shown in Fig. 4.6.

**Figure 4.5:** CG of Code Segment 4.4.2. A subgraph constant has been made of the inner filter node which then becomes the else operand to the outer filter node. For clarity we omit the E node of the definition and both the E and X nodes of the sub-definition.

**Figure 4.6:** Execution profile of Code Segment 4.4.2: The simulation takes 61 simulated time steps to execute for \( z = 10 \).

In the generation of this and all subsequent profiles we assume the following:
1. All instructions take the same amount of time to execute.

2. All fireable instructions are executed in the same time step.

Note the explosion of memory usage and parallelism in this profile. This is caused by the unconstrained recursive invocations of POP2. These invocations will very quickly be populated with the value 0 and with successively increasing negative integer values — generating a lot of irrelevant computations. These computations will not add to the measured execution time since they are handled in parallel with the computations performing useful work. As we can see, however, the impact of these extra computations on the memory usage and on the parallelism is substantial.

It is worth noting that the time measured for this execution would be expected from a crude analysis of the applicative reduction of this graph: the graph is approximately 6 levels deep and 10 recursions would result in 60 simulation time steps.

By constructing more subgraph constants, we can again change the execution characteristics. For example, by stemming the mul nodes we break the graph into approximately four levels. The program text to do this is:

**CODE SEGMENT 4.4.3** (2x, with stemmed mul nodes)

```
POP2(x) ← filter(eq(x,0), 1,
   {filter(even(x),
    {mul(POP2(div(x,2)), POP2(div(x,2)))},
    {mul(2, POP2(sub(x,1)))}
   )
)
```

The definition corresponding to Code Segment 4.4.3 is illustrated in Fig. 4.7, and the execution profile for this graph, with \( x = 10 \), is shown in Fig. 4.8.

![Diagram](image)

Figure 4.7: CG of Code Segment 4.4.3. A subgraph constant has been made of the two mul nodes which then become the else and then operands of the inner filter node. For clarity we omit the \( E \) node of the definition and both the \( E \) and \( X \) nodes of the sub-definitions.

The amount of parallelism and memory usage is the same as in Fig. 4.6. This is to be expected since we have not curtailed the recursive invocations. However, this profile has
a very interesting feature: the number of simulation time steps is half that shown in Fig. 4.6.

Where does this result come from? In Code Segment 4.4.3, the inner filter node is reducible as soon as its Boolean is evaluated. This contrasts with Code Segment 4.4.2 in which the inner filter node forms a sequential bottleneck at the end of the computation. The execution of each sequentially executed filter node takes 1 time unit. In total there are $3 \times 10$ filter nodes (i.e., 5 inner filter nodes in each recursive call and 10 such calls) representing 30 simulation time steps. In the profile of Fig. 4.8, this bottleneck is eliminated by the early firing of the filter nodes.

Since only one of the mul nodes can flow to the X node, the other can no longer do useful work and could, in principle, be terminated. In Code Segment 4.4.3, we cannot directly influence this speculative subcomputation so it is difficult to stop it. We will see later in Section 4.6.1 how this can be done.

For the moment we continue with the example and next consider stemming the POS2 condensed nodes. The program text is given in Code Segment 4.4.4:

**CODE SEGMENT 4.4.4** (with stemmed POS2 nodes)

```plaintext
POS2(x) = filter(even(x), 1,
    {filter(even(x),
        {mul({POS2(div(x, 2))}, {POS2(div(x, 2)))},
         {mul(2, {POS2(sub(x, 1)))})
    )
}
```

---

The filter nodes from irrelevant computations do not constitute part of the sequential chain. When a result is determined, these computations are terminated when they are deallocated along with the AF which produced the result.
The definition corresponding to Code Segment 4.4.4 is illustrated in Fig. 4.9, and the execution profile for this graph, with \( z = 10 \), is shown in Fig. 4.10.

Figure 4.9: CG of Code Segment 4.4.4. Subgraph constants are node of the three POP2 nodes. For clarity we omit the \( \Xi \) node of the definition and both the \( \Xi \) and \( X \) nodes of the sub-defintions.

Figure 4.10: Execution profile of Code Segment 4.4.4: The simulation takes 42 simulated time steps to execute for \( z=10 \).

Since the \( \text{mul} \) nodes are strict ancillary operators, we do not expect this to improve the simulated execution time of the program.

However, by stemming the POP2 nodes we restrict the recursion: a recursive invocation will now only take place when the results of that invocation are needed by the \( \text{mul} \) nodes — only then will a POP2 node receive an output environment. The explosion of parallelism and memory usage is thus quelled.

In our next example, we remove the output environments from the arguments of the condensed nodes. The program text is shown in Code Segment 4.4.5.

The definition corresponding to Code Segment 4.4.5 is illustrated in Fig. 4.11, and the execution profile for this graph, with \( z = 10 \), is shown in Fig. 4.12.

In this version of the graph we have succeeded in eliminating all the eager evaluations from the predecessor nodes of the inner \( \text{filter} \), and execution thus proceeds in a
CODE SEGMENT 4.4.5 (2^k POP2 nodes with subgraph constant operands)

```
POF2(x) — filter(eq(x, 0), 1,
    {filter(even(x),
        {mul({POF2({div(x, 2)})}, {POF2({div(x, 2)})}),
        {mul(2, {POF2({sub(x, 1)})})}}
    )
)
```

Figure 4.11: CG of Code Segment 4.4.5. Subgraph constants are made of the sub and div nodes. For clarity we omit the E node of the definition and both the E and X nodes of the sub definitions.

Figure 4.12: Execution profile of Code Segment 4.4.5: The simulation takes 51 simulated time steps to execute for x=10
coercion-driven manner. We note that the maximum memory allocation is the same
as in Fig. 4.10. This is because for both algorithms the recursive invocations are only
performed — and hence memory is only allocated — when needed. In Fig. 4.12, the
amount of parallelism has dropped. This is directly attributable to the fact that there are no
availability-driven evaluations. This is illustrated by Fig. 4.13 which shows the execution
profile of the filter nodes for Code Segments 4.4.4, and 4.4.5. We see that the same
number of filter nodes are executed in both cases, however, for Code Segment 4.4.5
the execution of those nodes is delayed.

![Filter node executions](image)

(a) Number of simulation steps

(b) Number of simulation steps

Figure 4.13: A comparison between the relative simulated execution times of the filter
nodes (a) Code Segment 4.4.4, and (b) Code Segment 4.4.5. The executions are the same,
however, Profile (b) is phase-shifted to the right.

Removing the output environments from the eq and even nodes is not advantage-
ous. This only leads to longer simulated execution times since both are strict ancillary
operators: an execution time of 71 units results.

The time taken to execute Code Segment 4.4.5 is smaller than the time needed to
execute the more eager program in Code Segment 4.4.2. From the foregoing, however,
we see that if we include the eq and even nodes in the coercion, the time is greater than
the eager case, as expected.

For the final experiment in this section we consider restoring the output environments
on the POP2 condensed nodes, but keeping their subgraph constant operands. This is
illustrated in Code Segment 4.4.6.

CODE SEGMENT 4.4.6 (2*, breeding parallelism quickly)

\[
\text{POP2}(x) \rightarrow \text{filter}(\text{eq}(x, 0), 1),
\]

\[
\{ \text{filter}(\text{even}(x),
\quad \{ \text{mul}(\text{POP2}([\text{div}(x, 2)]), \text{POP2}([\text{div}(x, 2)]))
\quad \{ \text{mul}(2, \text{POP2}([\text{sub}(x, 1)]))
\})
\}
\]

□
4.4. EXAMPLE SIMULATIONS

The definition corresponding to Code Segment 4.4.6 is illustrated in Fig. 4.14, and the execution profile for this graph, with $x = 2$, is shown in Fig. 4.15.

Figure 4.14: CG of Code Segment 4.4.6. For clarity we omit the E node of the definition and both the E and X nodes of the sub-definitions.

Figure 4.15: Execution profile of Code Segment 4.4.6: The simulation takes 15 simulated time steps to execute for $x=2$

This program should breed more parallelism than Code Segment 4.4.2 since the condensed nodes are reducible immediately. In fact this configuration breeds very quickly indeed.

Due to the limited resources of the simulator it was not possible to naively simulate this program with $x = 10$: the machine memory is used up too quickly with irrelevant, speculative computations.

The high degree of speculation is reflected in the smoothness of the execution profile which contrasts sharply with those of previous simulations. The end of the computation is shown in more detail in Fig. 4.16. This shows that from time 11 onwards only X nodes and mul nodes remain to be executed.

In Section 4.5 we see that for large values of $x$ this program can indeed be simulated.
4.4.1 Some Heuristics

The lessons learned from studying the foregoing simulations suggest a number of heuristics which can be used in the design of CG programs. These are as follows:

- By placing a subgraph constant in a function-strict port we strengthen the execution time due to mandatory grafting. However, these associations may also have the desirable effect of reducing the amount of memory needed to execute the algorithm.
- By giving subgraph constant operands to a filter node we trigger its execution early in the computation and thus may prevent filter node executions from forming a sequential bottleneck at the end of the computation.
- By stemming condensed nodes we quell speculative computation.
- Giving subgraph constant operands to grafted condensed nodes results in executions which breed parallelism very quickly.

4.5 Throttling

Throttling is a mechanism for controlling parallelism. As the name suggests the amount of parallelism in a computation can be increased by "opening the throttle" and decreased by "closing the throttle." In the CG machine two implementations of this concept are employed. They work by altering the TM-instruction evaluation order in response to the status of certain machine resources. The CG machine bases its throttling decisions on the amount of memory used and on the size of the triple list. From these two resources we define two throttles which we call the memory throttle and the triple list throttle. These limit the amount of speculation in the machine and so enable highly-speculative algorithms to be executed.

---

6A throttle limits the amount of speculation in a computation, however, it cannot force a non-terminating program to terminate.
4.5. THROTTLING

4.5.1 The Memory Throttle

When in use, the memory throttle will not allow any fireable condensed nodes to be executed if the memory usage has exceeded a predefined threshold value. As the memory usage drops below that threshold (i.e., when other activations terminate and so make memory available) all fireable condensed nodes will be executed in the same time step. The memory throttle thus has a dampening affect on the parallelism by not insisting that the threshold value act as a strict upper limit on the amount of memory usage.

We can illustrate the effectiveness of the memory throttle by executing the (previously intractable) program in Code Segment 4.4.6. We do this for a wide range of memory throttle values and depict the resulting execution profiles in Fig. 4.17.

As the throttle values get smaller, the parallelism is reduced and the parallelism profile gets less and less smooth — indicating that some activations are being given a chance to terminate before others are initiated. Furthermore, as the memory usage becomes too constrained, the execution time begins to lengthen. If we continue to decrease the memory throttle threshold further and further, we eventually reach a cut-off point below which the computation does not terminate. At that cut-off point all the available assigned memory has been used up and no more is available to accommodate those condensed nodes which need to be fired to allow the computation to proceed. Moreover, there is no guarantee that much of the already allocated memory will ever be needed in computing a result. Thus, although the memory throttle is successful in curtailing speculation — allowing otherwise unmanageable problems to be computed — the rate of the speculation becomes a factor in determining the minimal threshold value for the throttle.

The strength of locality in the $CG$ machine offers us an alternative to increasing the memory threshold value for breaking the deadlock. We discuss this in Section 4.8.

4.5.2 The Triple List Throttle

When in use, the triple list throttle will not allow any fireable condensed nodes to be executed if the size of the triple list has exceeded a predefined threshold value. If the threshold value is exceeded, condensed nodes are simply removed from the triple list before they are executed. When the triple list is built again in the next cycle of the TCP, these nodes will once again be placed in the list. The position in the list, and the value of the triple list throttle, will determine the chance of a triple being executed in that cycle. Thus, although both the memory and triple list throttles appear to act in very similar ways, the fact that they are based on the status of different machine resources, and the dependence of the triple list throttle on the way in which the triple list is constructed, actually results in both throttles having radically different impacts on execution profiles.

The triple list throttle ensures that the memory usage grows slowly. In fact, if the threshold value is set to 1, the memory allocation will be at most one per execution time unit. Thus, the execution will not use more memory than is absolutely necessary for termination. The actual memory size is of course determined by the algorithm. These points are illustrated in the profiles in Fig. 4.18 which represent the execution of Code Segments 4.4.2 and 4.4.6 using a throttle threshold value of 1.

By comparison, Fig. 4.18(a) takes much longer to execute. This is because it is
Figure 4.17: Execution of Code Segment 4.4.6 for x=10 using a range of Memory Throttle Thresholds
4.5. THROTTLING

Figure 4.18: The execution profiles obtained for x=10, using a triple list throttle threshold value of 1, (a) for Code Segment 4.4.2, and (b) for Code Segment 4.4.6

largely an availability-driven computation constrained by the throttle to execute in a low-
parallelism environment. Code Segment 4.4.6, translating mainly to a coercion-driven
computation, performs much better in this situation since only those activations which are
needed are made fireable. In both of these profiles the triple list is constructed by the TCP
using the pre-order walk of the V-graph. A reverse pre-order walk produces the triple
list in the reverse order and so alters the order in which condensed nodes are executed,
assuming a sufficiently low throttle threshold value. Fig. 4.19 shows the relative effect
of these triple list orderings by again depicting the execution profiles of Code Segments
4.4.2 and 4.4.6 with a throttle threshold of 1.

Figure 4.19: The execution profiles obtained for x=10 using a triple list throttle threshold value of 1 and a reverse pre-order walk of the V-graph (a) for Code Segment 4.4.2, and (b) for Code Segment 4.4.6
There is much scope for investigation into an optimal way of building the triple list. Such an investigation would be useful if it resulted in quantifying the relationship between the ordering of triples in the triple list, amount of speculative parallelism in the algorithm, and the threshold value of the triple list throttle. We identify this as an area of future research outside the scope of this dissertation.

We believe that there is a lot of scope for combining the memory and parallelism throttle for highly-speculative algorithms. In particular, it may be advantageous to be able to dynamically swap from one throttling mechanism to the other. This might be considered, for example, in the context of backtracking discussed in Section 4.8.

Finally, in Fig. 4.20, we depict the execution profiles of Code Sequence 4.4.6 using various values of the triple list throttle. These profiles contrast with Fig. 4.17 and so illustrate the differences between the two throttles.

### 4.6 Conservative and Speculative Parallelism

Conservative parallelism is so defined because it leads to results which are always needed. Speculative parallelism, on the other hand, leads to results which may or may not be needed[45].

In Code Segment 4.4.2, speculation is in the form of the eager evaluation of both the then and else branches of the filter node. In contrast, Code Segment 4.4.6 contains no such speculation since only one of the filter node’s branches is ever reduced in any given invocation. Thus in the algorithms seen so far, the speculation is either “turned-on” from the start — in which case it remains on until the computation terminates, or it is “turned-off” from the start — in which case it remains off. In Section 4.6.1, we address the problem of killing-off irrelevant speculations. For the moment, however, it is interesting to note that the amount of speculation in the algorithm can be altered by a judicious choice of CS associations in the construction of a condensation sequence. In effect, we can design our algorithms with speculation in mind.

To illustrate this, consider Code Segment 4.6.1, which is an alternative form of the powers of 2 algorithm. For the purposes of making comparisons, we note that if we were to replace every condensed node with its corresponding definition in Code Segment 4.6.1, we would get the algorithm expressed in Code Segment 4.4.3 (2*, with stemmed mul nodes).

**Code Segment 4.6.1 (2*, an alternative condensation)**

```
POF2(x) ← filter(seq(x, 0), l, \{GENCASE(x)\})
GENCASE(x) ← filter(even(x), \{MUL1(x)\}, \{MUL2(x)\})
MUL1(x) ← mul(POF2(div(x, 2)), POF2(div(x, 2)))
MUL2(x) ← mul(2, POF2(sub(x, 1)))
```

The definitions corresponding to this program are depicted in Fig. 4.21.
Figure 4.20: Execution of Code Segment 4.4.6 for x=10 using a range of Triple List Throttle Thresholds
Figure 4.21: The four CG definitions corresponding to Code Segment 4.6.1. (a) is the graph of POP2, (b) is the graph of GENCASE, (c) is the graph of MUL1, and (d) the graph of MUL2.

Figure 4.22: Execution profile of Code Segment 4.6.1: The simulation takes 82 simulated time steps to execute for x=10.
4.6. CONSERVATIVE AND SPECULATIVE PARALLELISM

In the execution of this program only one of MUL1 or MUL2 is invoked for each invocation of POP2. The one invoked will subsequently execute eagerly but speculation is avoided. The execution profile is shown in Fig. 4.22.

The simulated execution time is longer than that shown in Fig. 4.8 because the effective depth of the POP2 V-graph is greater and also because there is no speculation. We note the low parallelism at the end of the computation; this represents the firing of X0 and mul nodes only (all of the filter nodes have been executed by time step 43).

If we now graft nodes MUL1 and MUL2 in the GENCASE graph (Code Segment 4.6.2) we get the profile illustrated in Fig. 4.23. The longer simulated execution time comes about because we are postponing the evaluation of the filter node to the end of the computation. However, we see that the speculation is once again present.

CODE SEGMENT 4.6.2 (², with eager MUL nodes)

\[
\text{GENCASE}(x) \leftarrow \text{filter}(\text{even}(x), \text{MUL1}(x), \text{MUL2}(x))
\]

![Figure 4.23: Execution profile of Code Segment 4.6.2: The simulation takes 130 simulated time steps to execute for x=10](image)

We can reduce the amount of memory used without affecting the parallelism to any great degree by combining the definitions of GENCASE, MUL1 and MUL2. Fig. 4.24 illustrates the effect of computing GENCASE as specified in Code Segment 4.6.3.

CODE SEGMENT 4.6.3 (², with combined definitions)

\[
\text{GENCASE}(x) \leftarrow \text{filter}\left(\text{even}(x), \begin{cases} \text{mul}(\text{POP2}(\text{div}(x, 2)), \text{POP2}(\text{div}(x, 2))) \end{cases} \right)
\]

\[
\left\{ \text{mul}(2, \text{POP2}(\text{sub}(x, 1))) \right\}
\]

We note that this form also avoids the near-sequential execution at the end of the algorithm since, once again, the filter nodes are executed earlier.
4.6.1 Taming Speculation

A machine containing much memory and many processors can afford the cost of executing a reasonable amount of speculation — perhaps utilizing throttle mechanisms to keep it under control. However, employing a throttle can sometimes be compared to a firefighting exercise — addressing the problem of damage control when speculation gets out of hand rather than tackling the root cause, i.e., the elimination of irrelevant speculation. A more sensible approach then is to provide mechanisms for recognizing when speculation has become irrelevant and for stopping it; and for recognizing when speculation has become conservative so that it can be executed eagerly.

The careful reader will have already realized that the simulation of the CG machine has a mechanism for killing irrelevant parallelism: by recursively deallocating descendant activation frames as soon as a result is determined in an ancestor activation frame. (This is in contrast to other machines which require all speculative tasks to run to completion [11].) This mechanism, however, is only employed when a result from a graph invocation is determined — at that point it is easy to see what computations are irrelevant.

The determination of irrelevant, speculative computations at an earlier stage is more difficult but the locality of execution in the CG machine provides us with a solution.

First, however, we must define a new node which we call 


spec

— for speculation. The spec node requires two inputs before it can fire. At the language level, however, the programmer is only aware of one of those inputs. We call this input the programmer supplied operand. The second input to the spec node is supplied by the compiler, we call it the compiler supplied operand, and it consists of a compiler generated subgraph constant of the programmer supplied operand.

We envisage the spec node working in tandem with the filter node: two spec nodes acting as surrogate output environments for the then and the else subgraphs respectively, allowing both subgraphs to execute in a speculative manner. In addition the spec nodes are stemmed, and their associated subgraph constants are used to reduce the
filter node.

The action of the TM is also amended so that when firing a filter node with spec
node operands it transfers the filter node's output environment to the programmer de-
finied operand of one of those spec nodes, and deallocates the activation frame identified
by the compiler generated operand of the other spec node.

We illustrate this situation in Fig. 4.25.

![Diagram](image URL)

**Figure 4.25:** Both the then and else subgraphs start a speculative execution since they can be fired as soon as they are reducible. When filter nodes fire, the output environment of the else subgraph is replaced by the output environment of the filter node. The speculative execution of the else subgraph has thus turned to conservative parallelism. At the same time, the AF of the then subgraph, identified from the then subgraph constant, is deallocated and so the speculative execution of the then subgraph is killed.

In the first instance the speculative parallelism is made conservative, and in the second the speculative parallelism becomes irrelevant and is killed-off. This mechanism is possible due to the connection between subgraph constants and the activation frame in which they are executed.

### 4.7 The Communication-to-Processing Ratio

By maintaining the distinction between the TM and the APs in the CG machine, it is possible to define arbitrarily complex ancillary operations. To the TCP the complexity of the ancillary operations is irrelevant. Thus, the communication-to-processing ratio in the CG machine can be tuned to the complexity of the available ancillary processors. From this perspective, the TM part of the CG machine can be used as a unit for scheduling any type of computation. From an operational point-of-view, we note that the amount of communication in the CG network is kept to a minimum by only transmitting one result value per instruction execution — regardless of the number of destinations of that value. This is facilitated by the strong locality in the CG machine: any necessary copies of this result can be created in the TM. The size of the token information transmitted in the CG machine is also small: a single activation frame identification is sufficient to specify a result token destination.

For a given set of ancillary operations, we can also raise the granularity of executions in the CG machine (and thus lower the amount of communications) by the optimized
execution of certain subcomputations.

4.7.1 Identifying Subcomputations

Sometimes we can identify subgraphs created during program execution which, if executed as a single instruction, would lead to a faster overall execution. The first and most obvious of this type is the linear subgraph.

A linear subgraph could conceptually be executed as a single sequential computation provided that it can be recognized as such by the TCP during the construction of the triple list. This linear graph could then be wrapped as a sequential program for execution on a sequential machine. The input to the first instruction in this program is the input to the first node in the sequence and its output environment is the output environment of the last node in the sequence.

Of course, if these sequential subgraphs appear statically in the condensation sequence they can be identified as such and "wrapped-up" by the compiler.

The simplest example of these sequential subgraphs that are created dynamically is the sequence of X nodes that appears at the end of a tail-recursive computation. At present, these are handled completely within the TM and an arbitrarily long sequence of X nodes can be executed in a single memory write operation — the input to the first node in the sequence being written to the output environment of the last node in the sequence.

For non-tail-recursive functions the subgraph unfolded at the end of the computation is not linear. Nodes in this graph belong to all but one recursive invocation: the first node comes from the second last invocation and its operands are supplied by the final invocation; and the last node comes from the first invocation. This subgraph is thus a "vertical cross-section" through the V-graph. It could in principle be dynamically wrapped into an instruction and executed sequentially on a suitable processor. The cost-benefit of this is unclear and would depend on the complexity of each instruction in the subgraph and the amount of data involved; in some instances it might even be more cost-effective to move instruction sequences to the site of large amounts of data rather than vice versa.

By using the coercion-driven semantics of the CG model, and with appropriate scheduling, it may also be possible to construct a subgraph which is a "vertical cross-section" through the V-graph for the initial portions of a CG execution. Thus by using speculative computation we may be able to construct a subgraph (most probably non-linear) composed of subgraph constants. The TM can be used to connect these constants together and so form a completely availability-driven subgraph. This subgraph can then be wrapped as an instruction and subsequently executed on a dataflow processor, say (or possibly on another ring of a multi-ring CG machine). It is interesting to note that this subgraph would contain no function invocations, i.e., no condensed nodes, so it would be portable from one ring of a multi-ring CG machine to another. Again, this is an area for future research.

4.7.2 Studying Parallelism Profiles

Another advantage resulting from maintaining the distinction between the TM and the APs in the CG machine is that we can study the parallelism profiles of algorithms without
4.8. BACKTRACKING

having to compute a result. We can do this by defining ancillary operators which do little, or nothing.

For example, consider the following algorithm in Code Segment 4.7.1 to sort an array of numbers in parallel:

CODE SEGMENT 4.7.1 (Sorting an array)

\[
\begin{align*}
\text{OUT} & \leftarrow \text{SORT}(\text{thearray}) \\
\text{SORT}(\text{array}) & \leftarrow \text{filter}(\text{eq}(\text{size}(\text{array}), 1), 1, \\
& \{\text{merge}(\text{SORT}(\text{firsthalf}(\text{array})), \text{SORT}(\text{secondhalf}(\text{array})))\})
\end{align*}
\]

Here the \text{merge()} function need only return an arbitrary integer; the \text{thearray()} function returns an integer (100 in the profile); the \text{size()} function returns its parameter unchanged, and the \text{firsthalf()} and \text{secondhalf()} functions return their parameter divided by 2. The profile for this is shown in Fig. 4.26. Assuming an equal cost for executing each instruction, this profile will not change as the instructions are made more functionally complete. This approach may be useful for preliminary algorithm analysis.

![Figure 4.26: Profile of a parallel sort on an array of size 100](image)

4.8 Backtracking

In the CG machine it is possible to discard AFs from the leaves of the V-graph. The execution information thus lost can be recalculated provided that the condensed nodes whose execution produced those AFS in the first place are once more made fireable. In this way the discarded AFS will subsequently be reconstructed.

Thus if the machine runs out of memory while executing a graph, the ensuing deadlock may be broken by discarding some leaf activations and by recomputing according to a more conservative scheduling strategy. The triple list throttle, with its threshold set to 1, might be of use in this case.
If the machine deadlocks again, backtracking could once more be performed: perhaps this time to a greater extent, and yet another scheduling strategy could be chosen during the re-computation. This could be performed until the computation was completed or until it was determined that it was not reasonable to continue trying to terminate the execution given the available resources.

It is interesting to note that a re-execution after backtracking does not necessarily give rise to as much memory consumption or as much computation as the first execution. There are two reasons for this: subgraph constants, reduced to normal form in the first evaluation, may not need to be reevaluated; and condensed nodes may be replaced by the value resulting from their invocation (see Chapter 5, Section 5.2).

In general, a CG can be deconstructed by the removal of any, or indeed all, triple elements. For example, removing only those elements which are not statically bound effectively puts a graph into its initial state. This facilitates the reuse of the same activation frame for successive invocations of a tail recursive function — thus saving on the cost of memory management.

### 4.9 Input and Output

The execution order of some instructions can be vital to the correct evaluation of an algorithm. In particular, the correct sequencing of input and output instructions — which act at the interface to the computing environment — can be difficult to express in some paradigms. A general CG solution to the handling of input and output has not yet been investigated. However, we can illustrate how coercion can be used in the CG model to control output instruction execution for a specific example — the Towers of Hanoi problem. The CG code for this problem is shown in Code Segment 4.9.1.

**CODE SEGMENT 4.9.1 (The Towers of Hanoi)**

```plaintext
OUT(0) ← HANOI(3, 1, 2, 3, 0)
HANOI(count, source, dest, spare, x) ←
    filter(eq(count, 1),
    (writeout(source, dest)),
    RECUR(count, source, dest, spare, x))

RECUR(count, source, dest, spare, x) ←
    HANOI(sub(count, 1), spare, dest, source,
    HANOI(1, source, dest, spare,
    HANOI(sub(count, 1), source, spare, dest, x)))
```

The first four parameters to the HANOI definition represent the number of discs and the identification of the three "towers", respectively. The fourth parameter is used to establish the sequential dependency between the three RECURsive calls to the HANOI definition.
4.9. INPUT AND OUTPUT

The function defining `writeout` is treated as an ancillary instruction and is implemented by the following C language function:

```c
#include<stdio.h>

int writeout(a, b)
int a, b;
{
    printf("move disc from peg %d to peg %d \n", a, b);
    return 1;
}
```

The CG machine executes HANOI program to produce correctly sequenced output. The actual output for the given parameters is:

```
move disc from peg 1 to peg 2
move disc from peg 1 to peg 3
move disc from peg 2 to peg 3
move disc from peg 1 to peg 2
move disc from peg 3 to peg 1
move disc from peg 3 to peg 2
move disc from peg 1 to peg 2
```

Execution Steps = 57

The `writeout` function in this algorithm is a subgraph constant and will only be made fireable when needed. That is, when the value of `count` is 1. In contrast, if we remove the braces enclosing the `writeout` function it will be executed eagerly and the following erroneously sequenced output will result (albeit in less simulation time steps):

```
move disc from peg 1 to peg 2
move disc from peg 1 to peg 3
move disc from peg 1 to peg 2
move disc from peg 1 to peg 3
move disc from peg 2 to peg 3
move disc from peg 1 to peg 2
move disc from peg 3 to peg 2
move disc from peg 3 to peg 1
move disc from peg 3 to peg 2
move disc from peg 1 to peg 2
```

Execution Steps = 50

In general, the same control can be exerted over input instructions: these can also be coerced to provide input as required by the algorithm.
4.10 Summary

In this chapter we presented a high-level machine description for implementing the CG computation model. This machine, which is based on our notion of a computation triple, is capable of reducing functional expressions using a mixture of coercion-driven and availability-driven computations. These reductions are performed using a single, uniform reduction mechanism. Furthermore, we can recognize the data-driven model and the functional reduction model as special cases of the CG computation model: the former corresponds to executing a condensation sequence with no subgraph constants, and the latter to executing a condensation sequence in which all inputs are subgraph constants.

We constructed a simulator which gave us valuable insight into the potential impact of the CG philosophy on the design of a practical machines.

The CG computation model as evidenced in the simulator addresses, or manages to avoid, many of the classical problems associated with traditional coercion-driven and availability-driven computations. There is a strong mapping from the modularity of the algorithm description to locality of the executions in the machine. This locality is subsequently exploited in redressing the high communication processing ratio usually associated with token ring architectures; it allows us to tie token management to memory management, thus at once both reducing the size of token tags and obviating the need for separate tag management resources. The locality is also used to implement and manage speculative parallelism, making the killing of irrelevant parallelism relatively straightforward. The removal of graph activations in their entirety by the action of the X node also means that there is no problem with remaining packet garbage. The locality also lets us consider raising the execution grain by facilitating the recognition of simple sequential subgraphs — created dynamically throughout the CG execution.

In the next chapter we show how to represent structured data in the CG model and examine control-driven computations.
Chapter 5

$CG$ Structuring and Storage

5.1 Overview

The concept of stored data has traditionally proven difficult to incorporate into availability-driven and coercion-driven computing models. This is because it has no natural interpretation there. In principle, these computing models have no need of a data store — all their computations are free of side-effect and have no need of a shared state. In practice, however, these models can exploit data storage to model and manipulate structured data. This structuring can be used to reduce data communication by facilitating the manipulation of single coherent representations of data structures. As such, investigators have tried various ways of incorporating stored data into availability-driven and coercion-driven computing models [50, 51, 52, 11, 30, 41, 43]. In some cases ancillary units have been employed in which stored data are handled in a manner separate to the main computing paradigm.

In contrast, an updateable data store is a prerequisite for the imperative paradigm whose state transition semantics depend on the mutation of shared data. By designing an algorithm from the perspective of mobile function elements, the concept of a $CG$ store appears as a natural consequence in the $CG$ model. When we use these $CG$s to represent data, a $CG$ store becomes a data store. The $CG$ store, however, is much more general and can be used to store any incomplete $CG$. In the $CG$ model no distinction is made between functions and data; every datum is a zero-arity function. Strictly speaking a node representing a zero-arity function needs no operand ports. Therefore (by construction) a node representing a simple data value is never complete and so can never fire.

We shall see how finite and infinite $CG$ stores can be dynamically created and processed. These stores are manipulated in the $CG$ model in a similar manner to every other $CG$ definition. In particular, this means that $CG$ structure processing does not require a garbage collection phase.

By using the process of node deconstruction, $CG$s in a $CG$ store can be mutated and so the store can be used as a state element, facilitating the expression of imperative-like semantics to implement what we shall call a control-driven computation.

---

1A distinction is, of course, made between functions and operands.
5.2 Mobile Function Elements

Up to now we have been mainly concerned with mobile operand, and output environment elements. Thus, in the graphs considered so far, function elements have been treated as stationary. When nonzero-arity function elements of a triple are mobile (as defined in Section 3.2), however, nodes can be interpreted as \( CG \) stores. We illustrate this with a simple example.

**Example 5.2.1** (Mobile function elements)

We consider the expression \( g(f(y)) \) where \( y \) is a simple datum, and \( f \) is either an ancillary function or a condensed node representing an arbitrarily complex \( CG \) computation. For the sake of the example, we choose \( g \) to be function-strict so that it will coerce the evaluation of \( f(y) \). Fig. 5.1(a) shows a definition of this expression. We assume that the operand \( y \) is statically bound to a node, and that the functions \( f \) and \( g \) are mobile.

![Diagram](image)

**Figure 5.1:** Four stages in an execution with mobile function elements

In Fig. 5.1(a) a node is depicted having the operand \( y \) statically bound to its operand port and the function \( f \) arriving via its function element port. This node is itself an operand element arriving on the operand port of its destination node. The function \( g \) also flows into this destination node, but via its function element port. Fig. 5.1(b) shows the fireable node containing \( g \). In Fig. 5.1(c) the function \( f \) is bound to its node and in Fig. 5.1(d) the evaluation of \( g \) coerces the evaluation of \( f(y) \) by grafting.

If the definition of this expression contained statically bound function elements the resulting execution would be the same, even though the sequencing constraints in both cases are very different.

The interesting feature of this example is that the operand of \( f \) appears to be stored on the operand port of a node and is accessed by a mobile function element.

\[ \square \]

5.3 CG Storage

Motivated by Example 5.2.1, we conclude that the operand ports of a node can be used to store any \( CG \) in the \( CG \) model. For the explicit storage of \( CGs \), however, we will restrict ourselves to nodes having a single operand port whose operand element represents the
5.3. CG STORAGE

stored CG, and for reference purposes we call these nodes store\(^2\) nodes. In Fig. 5.2(a)
the operand d is depicted. In general, a stored operand may be used as the parameter to
any suitable function which can be passed to the function element port of a store node.
In particular, we can define a new (TM) operator, called the fetch function, which will
retrieve a stored CG from a store node. This function executes by copying its single
operand element to its output environment (and so has actions similar to those of the X
node).

![Diagram](image)

Figure 5.2: Retrieving a stored CG from a store node

To fetch the operand d from the store node in Fig. 5.2 and to communicate it to
a particular destination, we need to generate two CGs: one whose function element is
the fetch function, and the other whose output environment is the desired destination.
These CGs are sent to the function element port and the output environment input port
of the store node (Fig. 5.2(b) and (c)). When the store node fires, a copy of its operand
is sent to the output environment in the usual way (Fig. 5.2(d)). To preserve the illusion
of storing a CG on the operand port of a store node, any function sent to the store node should
have associated deconstruction semantics which result in the removal of the function and
output environment elements from the node after execution (Fig. 5.2(e)). This leaves the
stored CG ready to be operated on once more.

The action of fetching a CG from a store node is very similar to the action of refiring
the X node in a subgraph constant evaluation. However, the deconstruction semantics
are different — the X function is not removed from the node after firing. Thus, the
difference between the X function and the fetch function lies only in the type of node
deconstruction they precipitate.

One consequence of the similarity between the X function and the fetch function is
that the process of grafting can be viewed as the reading of a memory location.

Note that the operand in our example does not have to be statically bound to the store
node, i.e., the order in which a computation triple is formed is irrelevant. Thus, a stored
CG could in fact be under construction in some ‘producer graph’ while the fetch and
output environment CGs are being generated in a ‘consumer graph’. In this way the store
would act like a single element buffer. This type of store node is semantically similar to

Furthermore, if the stored CG is a subgraph constant a ‘lazy store’ can be created. CGs
fetched from a lazy store can subsequently be coerced by grafting them onto the consumer
graph when their evaluation is needed.

\(^2\)A “store node” is not special in any way! It does not differ in design from every other node. The name
refers only to the way in which this node is implicitly used.
5.3.1 Finite Structures

We can use condensation to create a single coherent representation of a stored $CG$ structure. Firstly, however, we examine how definitions for these structures can be constructed.

**Storing One $CG$**

We begin by making a $CG$ definition for storing a single $CG$. This is illustrated in Fig. 5.3(a). The output environment of the $E$ node in this definition is the function port of a store node. As a consequence, the stored $CG$ must be statically associated with the store node — it cannot be placed there dynamically as part of the definition invocation.

![Diagram of $CG$ structure](image)

**Figure 5.3: A simple $CG$ storage structure containing a single store node**

In Fig. 5.3(b), we show a fetch node with a condensed node operand. (This is our first time encountering a condensed node being used as an operand.) According to the Mutual Reduction rules of Chapter 3 Section 3.2.4, the atomic fetch function will reduce the condensed ival node. Thus a triple is formed from the ival operand, the fetch function and the output environment of the fetch node. When this triple is executed an instantiation of the ival definition is created and it is populated by the fetch function. The fetch function is sent via the E node to the store node, and its output environment is sent to the X node of the instantiated definition in the usual way (Fig. 5.3(c)). The stored $CG$, $d$, is sent to the output environment of the fetch node when the store and $X$ nodes fire. If the stored $CG$ is a subgraph constant, its execution is coerced by the $X$ node. On the other hand, if it is a condensed node, whose evaluation would precipitate the instantiation of another definition, then, just as if it were an atomic $CG$, it can pass uncoerced through the $X$ node — without affecting the deallocation policy.

We note that any function of arity one will process the ival node just as if it were a simple operand.

**Storing Many $CG$s**

Storing more than one $CG$ in a structure introduces the problem of distinguishing between them for retrieval. Storage elements at the same abstraction level tend to be unrelated — constituting no more than a collection of unconnected nodes.

We could organize these stored elements using a number of nested filter nodes to distinguish between them. This solution does not generalize very well and so our preferred approach is to design specific semantics for the firing of the $E$ node of a definition graph.
5.3. \textit{CG} STORAGE

The definitions that we have seen so far have been composed of single, connected graphs. In contrast, a storage structure definition is, in general, composed of a number of unconnected subgraphs, each of which represents a stored element. We shall see that in both types of definition, the E node performs a different task and so is configured differently. In Fig. 5.4 we illustrate all input and output ports of an E node. If a definition graph is composed of a single connected subgraph then in its E node each set of output environments identifies the destinations of each of its operands. On the other hand, if a definition graph is composed of unconnected subgraphs, each set of its E node’s output environments is used to identify the destinations of all the E node’s operands. The latter E node configuration also has two extra inputs: a route input (a value between 1 and the number of E node operands) which determines which set of output environments are used per definition invocation; and an output environment which identifies the definition’s X node.

![Diagram of a general E node](image)

\textit{Figure 5.4: The ports of a general E node}

Condensing a collection of disjoint subgraphs into one coherent structure takes place with one major change in the condensation process: \textit{no transmitter is connected statically to the X node}. Instead, an edge is created from the E node to the output environment port of every transmitter. In this way each transmitter can be connected dynamically to the X node — if required.

By using the route input to choose between the set of output environments one — and only one — of the disjoint subgraphs is populated per definition invocation. Thus, the semantics of the E node execution allows us to dynamically invoke one of a number of disjoint subgraphs.

To illustrate this process, we consider storing two \textit{CG}s in a single structure which we call \textit{2val}. The definition graph of \textit{2val} makes use of two store nodes. This is shown in Fig. 5.5(a). The output environments of the store nodes are supplied dynamically by the E node, if required. A condensed node of the \textit{2val} definition accepts two operands. The first is an integer which becomes the route input of the definition’s E node. The second operand is a function which is used to operate on the stored element identified by the route input. In Fig. 5.5(b) the latter operand is supplied by mutual reduction. The populated definition, shown in Fig. 5.5(c), is similar to the \textit{1val} structure of Fig. 5.3.

Suppose that the stored elements (d1 and d2) in the \textit{2val} structure are themselves condensed nodes of the \textit{1val} definition. When one of these nodes is fetched, the X node will not coerce its evaluation but will simply pass it to its output environment. Indeed the
Figure 5.5: Definition of a two-\textit{CG} storage structure

\(X\) node cannot coerce the evaluation of a condensed node since it is not allowed to take part in mutual reduction. This invocation of the \textit{2val} definition is not needed to evaluate the \textit{1val} condensed node and so can be deallocated in the normal way when its \(X\) node fires.

A many-\textit{CG} store structure definition can be created following the \textit{2val} store structure design by simply adding extra store nodes. This type of structure facilitates direct access to each of its stored elements.

\textbf{List Structures}

Sometimes it is more appropriate to access stored elements sequentially. The simplest sequential access structure is a list and we can construct a list using a number of \textit{2val} definitions as list elements. For example, to construct a list with three elements \([1, 2, 3]\), we create three definition graphs which represent the three lists \(L1 = [1, L2], L2 = [2, L3]\) and \(L3 = [3, []]\). These are depicted in Fig. 5.6. We note that \([]\) is a special value which represents an empty list.

In the construction of the definition graph of \(L3\), the node \(L3 + 1\) represents a condensed node of the definition of \(L3 + 1\).

We define \texttt{head} and \texttt{tail} functions as illustrated in Fig. 5.7(a) and (b) to process list structures. These definitions are populated with condensed nodes representing list structures. An operand list populating these definitions flows, via the \texttt{E} node, to the function element input port of a node as illustrated. In the \texttt{head} definition this node has a statically associated operand \(1\), and the corresponding node in the \texttt{tail} definition has the statically associated operand \(2\). When the list becomes bound to this node it is mutually reduced by the \texttt{fetch} function. The \texttt{head} and \texttt{tail} definitions illustrate how the graph in Fig. 5.5(b) can be constructed.
5.3. **CG STORAGE**

![Diagram](image)

Figure 5.6: *Definitions representing a 3-element list. The first element in each structure is the head of that list and the second element is its tail.*

In Fig. 5.7(c) and (d) we illustrate the effect of taking the head and tail of the list L1 from Fig. 5.6.

![Diagram](image)

Figure 5.7: *Definitions of head and tail functions for list processing*

Thus CG store structures can be processed by both direct access and sequential access mechanisms.

**Dynamic Storage Structures**

In the preceding sections we were concerned with accessing structure definitions in which stored CGs were statically associated with store node operands. We refer to these as static storage structures. In contrast, a dynamic storage structure is created when CGs are dynamically associated with store node operands.

Condensed nodes of static storage structures can (just like every other condensed node seen so far) be evaluated independently of each other since each condensed node evaluation invokes its own copy of its associated definition. For dynamic storage structures, however, a causal relationship is established between the filling of the structure and the subsequent manipulation of its elements. The first invocation of a dynamic storage...
structure definition fills the structure with all storage elements simultaneously (for non-mutable structures this can happen only once). This invocation is made by firing a special condensed node called a constructor node. In contrast to a condensed node, a constructor node can populate its definition independently of the $E$ node. Thus it has an extra operand (an output environment) which indicates the destination of its other operands in its corresponding definition.

Condensed nodes of a dynamic storage structure can only be created as a result of firing a constructor node of that structure. This enforces the desired causal relationship. Until now, condensed nodes have been created from the algorithm description and not dynamically as part of the computation. We define a dynamically created condensed node to be the $E$ node of the definition invocation. Thus, in the definition of a dynamic storage structure a copy of the $E$ node is statically bound to the operand port of its $X$ node, so that when the definition is invoked for the first time the result is that $E$ node. When the $X$ node fires it should be deconstructed to remove its operand and output environment elements. In addition, the invocation should only be deallocated if the populated definition is to be handled in a non-shared manner as outlined below.

We illustrate the firing of a constructor node ($\text{const}$) in Fig. 5.8. The result of this execution is the $E$ node of the populated definition invocation. This node is passed to an ancestor graph and there it can be reduced by mutual reduction. When it fires it populates the storage structure with its operand elements.

![Diagram](image)

**Figure 5.8:** The constructor node, $\text{const}$ populates the two-CG storage structure with $a_1$ and $a_2$. The result of the $\text{const}$ node execution is a copy of the $E$ node of the newly created definition.

The populated version of the dynamic storage structure definition should be used for the evaluation of the condensed nodes (the $E$ nodes) of the structure. There are two ways of doing this:

1. The non-sharing approach involves placing the populated invocation in the definition graph memory of the TM. It can then be subsequently used like a static storage structure and an ‘own copy’ can be created every time one of its condensed nodes is fired.

2. The sharing approach involves leaving the populated invocation in the V-graph and reusing it to evaluate all condensed nodes.
5.3. **CG STORAGE**

In practice, when dealing with non-mutable structures, the non-sharing approach is most appropriate. On the other hand, if the sharing approach is adopted then the following considerations must be made:

- The evaluation of the condensed nodes of the shared invocation must be executed in sequence. If more than one condensed node is fired at the same time more than one function element may be associated with a node in a shared invocation. In a non-mutable structure this sequential execution can be enforced either in the algorithm design or by the triple constructor process in the TM.

- The firing of the \( X \) node should not deallocate the shared invocation. In fact, we can use a very simple criterion to trigger this behaviour: if an \( E \) node ever passes through an \( X \) node, the deallocation semantics of the \( X \) node must be "turned off" for the remainder of the computation. (We see in Section 5.4 that this decision does not lead to problems in deallocating structures when they are of no further use.)

- The function elements and output environment of all store nodes and the operand and output environment elements of the \( X \) node should be removed by deconstruction so that the shared invocation can be reused.

To illustrate a shared dynamic storage structure, we consider constructing dynamically the list \( L = [1, [2, [3, 1]]] \). The graph to do this is shown in Fig. 5.9. It consists of three \( \text{cons} \) nodes. When the first is fired, it invokes a definition similar to that shown in Fig. 5.8 and populates it with the data value 1 and the empty list value \([1]\). The result of firing the first \( \text{cons} \) node is the \( E1 \) node of the invoked definition. This \( E1 \) node is one of the operands to the second \( \text{cons} \) node. Since both the \( E1 \) node and the \( \text{cons} \) node are non-atomic, the rules of mutual reduction tell us that the \( E1 \) node is used as an operand to the \( \text{cons} \) node. Thus, when the second \( \text{cons} \) node is fired it invokes another instance of its definition and populates it with the \( E1 \) node and the datum 2. The result of firing the second \( \text{cons} \) node is the \( E2 \) node corresponding to its invoked definition. This process can be continued for lists of any finite size and results in a series of definition invocations, each holding a populated instance of the two-\( CG \) storage structure — in effect a dynamically created version of the definitions shown in Fig. 5.6.

The list \( L \) can even be created lazily if we choose to build it as a series of subgraph constants. This is illustrated in Fig. 5.10. In this definition the second and subsequent elements in the list are created only when needed. The appropriate \( X \) nodes will then coerce their evaluation by grafting.

### 5.3.2 Infinite Structures

Fig. 5.10 depicts a lazily finite \( CG \) store. This store is finite because it contains a finite number of constructor nodes, and it is lazy because these nodes are represented as subgraph constants. We can use recursion to create an infinite store by not limiting the number of constructor node invocations. As a first example we consider defining the list of all positive integers. The definition graph for generating this list (which we call
Figure 5.9: Eager construction of a finite list. The three constructor nodes dynamically create activations of the definitions for each list element. The construction of the list is availability-driven; the last element in the list is created first.

Figure 5.10: Lazy construction of a finite list. The three constructor nodes of Fig. 5.9 are shown here as subgraph constants. The operand to the first constructor node is stored in the storage structure and it is not evaluated until it is needed. Thus, only one activation of the definition is initially created. The list is coercion-driven; the first element of the list is created first.
5.3 Cg STORAGE

![Diagram of definition of ints](image)

**Figure 5.11:** (a) Definition of ints: a recursive definition of all positive integers. (b) The subgraph constant representing the recursive part of the definition is grafted onto the X node when the tail of ints is executed.

The `ints` is shown in Fig. 5.11(a). We employ the 2val definition as the basis of the `ints` definition.

In contrast to previous examples, the second element in this store is statically defined to be a constructor node of the definition. Every time the definition is invoked, this constructor node is made reducible and the `E` node of the definition is returned as a result. When the constructor node is fetched from the store it is grafted onto the `X` node (Fig. 5.11(b)). This coercion results in the creation of another invocation of the definition; its store nodes are populated with the operands of the subgraph constant, and the `E` node of this invocation is returned as the result. In Fig. 5.12 we illustrate the invocations resulting from the execution of `head(tail(ints(0)))`.

The firing of the `ints(0)` constructor node creates the first invocation of its definition graph. This invocation is populated with the operand, `0`, and the `E` node of this invocation is returned as a result. This `E` node is subsequently reduced by the `tail` function and so the subgraph constant element is fetched. The evaluation of this constant is coerced by the `X` node and as a result a second invocation of the definition is created. This invocation is populated with the operand `1`, and its `E` node is returned as a result. This `E` node is subsequently reduced by the `head` function which fetches the first element from the store structure. No further invocations of the store structure are created until the `tail` of the second invocation is fetched.

Thus, the `ints` list is processed lazily: the next element in the list is only generated when it is needed.

A condensed node of the `ints` list is a single coherent representation of an infinite graph. In Fig. 5.13 we illustrate how it can be used as an operand. A definition of a graph called `inirs` is depicted which creates a reverse list of the first `n` natural numbers. By using the `ints` definition, no theoretical limit is placed on the size of `n`. 
Figure 5.12: Executing head(tail(ints(0)))
5.4 Garbage-Free

The $CG$ model needs no explicit garbage collection — it is totally self-cleaning. All static and dynamic store structures which are implemented in a non-shared manner can be recursively deallocated when their $X$ nodes fire. However, dynamic store structures which are implemented in a shared manner cannot be immediately deallocated since they are needed to evaluate their associated condensed nodes. These condensed nodes are represented by $E$ nodes and are passed to ancestor graphs where they are reduced by mutual reduction. Whenever an $E$ node is the result of firing an $X$ node, the deallocation semantics of that $X$ node are "turned off". That invocation must then be kept until it is explicitly deallocated by an ancestor.

The rule for deallocation can then be simply expressed as follows:

- If the operand to an $X$ node is not an $E$ node then firing that $X$ node results in the deallocation of its own activation frame and the activation frames of all its children.
- If the operand to an $X$ node is an $E$ node then neither this nor any subsequent firing of that $X$ node will cause a deallocation.

This simple memory management policy is only possible because of the hierarchical nature of $CG$ computations: every child activation frame has a unique parent, and children can only pass results to their parent.

Thus, the issue of garbage collection, which accounts for much processing and overhead in other machines, is not a problem in the $CG$ model — there is no garbage.
5.5 Update In-Situ

Prior to this we have been using CGs to model computations that are free of side-effects. As such, we have not had occasion to deconstruct store nodes by removing their operand elements. Storing CGs to construct coherent structure representations is one thing; to change stored CGs, however, implies a completely different computing paradigm. In the CG model we can use node deconstruction, sequenced by mobile function elements, to effectively mutate stored CGs. In this way we can use a control-driven computation to implement state transition semantics in which the state of the computation is represented by a shared mutable storage structure. The definition of a mutable structure has exactly the same topology as the shared structures presented in Section 5.3.1. The elements of a storage structure can be mutated by functions with appropriate deconstruction semantics. By definition, these functions replace one CG by another but do not produce any explicit result value. However, the CG model relies on the production of explicit result values for the construction of computation triples. Thus, we define the result of executing every side-effect function to be the E node of the storage structure on which it operates. (Functions which are free of side-effects can still operate on the same structure to produce explicit result values.) The E node of a storage structure can be used to sequence side-effect functions by flowing from one to the next while being mutually reduced en route. The requirement on Page 84 that constructor nodes populate a complete storage structure simultaneously can be relaxed since the correct sequencing of instructions in this control-driven approach is the responsibility of the programmer.

All side-effect functions simply move CGs around in memory and so they can be implemented in the CG machine as TM operators.

As an example of a side-effect function, consider mutating the CG stored on the i\textsuperscript{th} store node of a storage structure. This action is analogous to the execution of the assignment statement in the imperative paradigm. We illustrate the CG approach in Fig. 5.14.

![Figure 5.14: The CG equivalent of an imperative assignment statement](image)

In Fig. 5.14(a), the E node of the storage structure is partially reduced, its operand i identifies the i\textsuperscript{th} store node of its corresponding definition. In addition, the binary side-effect function assign is partially reduced by p and the 1-ary function assign.p
is used to mutually reduce the storage structure. When this TM operator is executed the operand element of the \( r \)th store node is mutated and the \( B \) node of the storage structure emerges on the output environment of the assign node.

In a computation which utilizes side-effects the potential parallelism of an algorithm is limited by the access dependencies of its state variables. If the complete state is represented in a single storage structure then the algorithm graph needs to be linear where there is a danger of more than one function simultaneously reducing the same store node in the storage structure.

If the state is distributed across more than one storage structure then the potential parallelism increases monotonically. In fact, traditional data flow takes this scenario to its logical conclusion — every storage structure consists of only one "variable" — and so mutation degenerates into value transformation.

The shared mutable storage structure used to represent the state of control-driven \( CG \) computations can be allocated and deallocated in the same manner as every other \( CG \) structure. As a result, these computations are also garbage-free.

Notwithstanding the sequential execution enforced by the mutation history of a shared, mutable store, the power of expression associated with control-driven computations is substantial. In the \( CG \) model we can exploit this power while at the same time confining the essential sequential parts of an execution to the definitions of specific condensed nodes.

### 5.6 Summary

When the function elements of a triple are mobile, the operand elements are viewed as being stored. Mutual reduction can be used for manipulating and retrieving stored \( CGs \).

As such the concept of which \( CGs \) are mobile, and which are stationary becomes relative.

There is no obvious relationship among \( CGs \) stored in the same abstraction level of a structure. Each element is an independent \( CG \). To distinguish between these \( CGs \), we introduced semantics for the firing of the \( B \) node. These semantics are used to dynamically instantiate one and only one element of a store per \( B \) node execution.

Constructor nodes are introduced to create and populate store definitions dynamically. Dynamically populated stores are identified by their associated \( B \) node. These are passed into the ancestor graphs and "turn-off" the deallocation semantics of their associated \( X \) node \textit{en route}. In ancestor graphs, \( B \) nodes of dynamically populated stores are manipulated like ordinary operands and are reduced by mutual reduction.

We can create infinite \( CG \) store structures using recursive definitions containing constructor nodes. If these nodes are represented as subgraph constants then the infinite store can be built lazily.

All structures built in a \( CG \) computation form part of a hierarchical tree of activation frames. This topology, along with a simple rule for activation frame deallocation ensures that no garbage collection phase is needed in the \( CG \) machine.

Being able to store \( CGs \) and mutate them using mobile function elements allows us to implement state transition semantics and so makes the control-driven model of computing a special case of the \( CG \) model.
Chapter 6

Epilogue

6.1 The Recent Past

Apart from the rich computational framework offered by the CG model, the CG approach also has a number of positive characteristics that can be exploited in the construction of practical machines. Many of these have already been encountered throughout the preceding chapters. We summarize them here.

Locality of Execution

The CG machine presented in Chapter 4 implements what we call a Tagged-Data-Space computing model. In this model (and also in a similar approach outlined in [21, 17]) token matching can be performed using random access memory instead of needing expensive associative memory (as utilized in [22, 18], for example). Consequently, those parts of a program which logically belong together can be kept local to each other in the CG machine. The management of token matching space is expensive for every tagged-token machine and many trade-offs exist as outlined by [53]. Notwithstanding these restrictions, the modularity of a high-level language can be mapped onto spatial locality in the CG machine. Many of the positive features of the CG model outlined here are directly derived from this fact.

A Simple Firing Rule

One of the major advantages of data-driven computations is the simple way in which instructions are sequenced for execution. This simplicity has probably contributed more to the popularity of data-driven machines than any other single factor. The firing rule of the CG model, while being much more powerful, is just as simple.

Reduced Network Communication

By using output environments in the CG model, we can eliminate the need for transmitting more than one copy of a result token over the network. A result token can be copied by the TCP to the (many) destinations specified in the output environment. This contrasts
with the Manchester Machine in which there is less spatial locality. In that machine it is necessary to generate a separate result token for every destination. A DUPlcation operator is employed to replicate a result if it is to be sent to more than two destinations. Each execution of a DUPl operator results in the generation and communication of two new result tokens containing the same value.

**Variable Grain of Execution**

Because many levels of abstraction can be represented in a condensation sequence, individual abstract nodes may be executed as a single instruction provided an appropriate ancillary processor is available to the CG machine. We illustrated this in the CG simulator by defining all ancillary instructions to be external to the simulator and arbitrarily complex.

In addition, it is sometimes possible to identify certain subgraphs of the V-graph which can be executed as single instructions (Chapter 4 Section 4.7.1). For example, linear subgraphs can be encapsulated in a CG and executed as a sequential program on a sequential machine. Alternatively, a linear graph can be sent one node at a time to the same ancillary processor. Result data generated from the execution of each node can be held locally in the registers of that processor and act as the operand data for the next node. Thus, data in the registers can be mutated by the flow of instructions from the linear graph. This is an example of a control-driven execution in the machine organization as opposed to a control-driven execution at the algorithmic level.

We can view the CG machine as a sophisticated scheduling unit acting to dispatch instructions to a collection of heterogeneous machines and collating their results to construct further instructions.

**Garbage-Free**

Data structures are a special case of CG structures and as such they are dynamically created and destroyed using the same mechanisms for memory allocation and deallocation (Chapter 5 Section 5.4).

Unwanted structures and remaining packet garbage are collected as part of a simple memory management policy.

**A Mutable Store**

By adopting an appropriate perspective, certain CGs are considered to be stored on the operand ports of nodes. These CGs can be fetched, can take part in computations, and can be mutated using the deconstruction (Chapter 5 Section 5.5). The concept of shared, mutable store is thus inherent to the CG model.

**Backtracking**

A CG computation can be "rolled-back" if the machine becomes deadlocked due to lack of resources (Chapter 4 Section 4.8). A subsequent execution of the graph can then adopt an alternative sequencing strategy. Due to the evaluation of subgraph constants and the
6.1. THE RECENT PAST

Graph-rewriting performed in the first execution the next and subsequent executions of the graph may involve less work.

Coarse-Grain Throttling

The execution of complete definitions can be postponed until it is convenient to process them as a whole. This is coarse-grain throttling and it can be performed in the CG model because many different abstraction levels are represented in a condensation sequence and because CGs are reduction-strict.

The same structural model is used to represent data structures and algorithms so the same throttling mechanisms can in principle be employed in the dynamic construction of data structures.

Handling Speculation

Speculative parallelism can utilize otherwise idle processors and, if the work done is subsequently needed, can result in a reduced execution time compared to a purely conservative approach.

Speculation is independent of reduction order and both availability-driven and coercion-driven systems can be speculative.

With regard to speculation, the following curious behaviour has been reported by the MIT dataflow group[12]: "on our dataflow machine this phenomenon is reflected in the sometimes surprising behaviour that an answer may be printed long before termination is reported." Furthermore, in this implementation it is essential that all the speculative tasks are completed: only then can it be concluded that no run-time error occurred.

In the CG machine, speculation needs no explicit management. When a result is obtained from a computation, any associated speculation is automatically terminated by the memory management policy. Furthermore, the inherent speculation associated with certain language constructs can be explicitly "turned-off" by the programmer. The amount of speculation in a computation can be determined in the program rather than at a language design level.

Free Label Management

In a tagged token machine the allocation and recycling of token labels can be costly activities. In the CG machine model, token label management is synonymous with memory management and so represents no extra cost. In addition, the CG model needs no explicit label manipulation operators.

Small Label Size

A token label consisting of two addresses identifying an output environment and an activation frame is sufficient in the CG machine model. This contrasts with other models, some of which require a separate label to identify activation level, iteration level and structure element.
Other CG machine organizations could lead to an even smaller token label. For example, by encoding the output environment information in every activation frame, the token label could be reduced to a single address.

6.2 The Near Future

Despite a significant amount of work already carried out, we have only just begun to realize the potential of the CG model. Many research topics remain to be investigated. Among them we identify the following:

Language Issues

It should be possible to compile many existing imperative and functional languages into condensed graphs. However, it is still unclear how a resulting execution would compare with a traditional execution of these languages.

To exploit the full power of the CG model it will probably be necessary to develop a dedicated CG language. In Chapter 4 Section 4.3 we introduced a preliminary CG language which allowed us to easily choose between CG association types. However, this nested expression language is not ideal since it prevents the exploitation of common subexpression and so increases the number of instructions to be executed.

In addition, a specialized CG programming language could give the programmer more control over the reduction order and the amount of speculation.

Software Engineering

The CG model is based on abstraction and encapsulation. These concepts are essential in the construction of large, reusable software systems and we would like to see how appropriate the CG approach is for this activity. For example, we are of the opinion that it is possible to implement an object-oriented design using CGs. The 'operations type' interface characteristic of object-oriented systems could be modeled by sending messages to nodes that represent the operations of an object, and the object itself could be represented as a condensation of its constituent operation and data nodes. Messages sent to the 'object node' could be forwarded to its appropriate 'operation node'. The messages themselves could be represented by the pair (input environment, output environment). The input environment could represent the parameters of the destination operation and the output environment could represent the identification of the sender object.

Computing with Relations

In Chapter 3 Section 3.1.1 we chose to model single-valued functions only. This is not a fundamental restriction and in fact CGs can be used to model multi-valued functions by altering the number of output ports of certain nodes appropriately. In addition, it would be interesting to see if the CG model could be used to model a system of general relations [13].
6.3. AFTERWORD

Architecture

The applicability of the CG machine is wide, ranging from being the front-end scheduler for a network of work-stations running Parallel Virtual Machine (PVM)[31], to being a fully fledged parallel machine with dedicated ancillary processors, Triple Management unit and network. We are interested in CG machines spanning this complete spectrum. At the workstation end, we are interested in seeing if the CG philosophy aids the programming task and wish to study the resulting impact on performance.

At the other end of the spectrum we believe that the locality of the CG execution can be exploited in the construction of a multi-ring architecture. Here, we would hope to exploit spatial locality to address load balancing problems.

Special Hardware

In the realization of a CG machine, there are a number of specialized hardware units which could be developed to increase execution speed. One such unit is a hardware scheduler.

Every condensed graph is acyclic and so in principle a compiler could perform a critical path analysis to associate an estimated execution time with each node. (For the purpose of this analysis a recursive graph could be defined to be on the critical path.) In addition, a slack time (the amount of time which an instruction can remain unfired without affecting the overall finishing time) could be associated with every node. Now, if in some step of the computation there are more executable instructions than available processors, we ideally would like those nodes with the smallest slack time to be executed first. Fireable instructions are accessed in the hardware scheduler — according to their slack time. When some instructions are dispatched for execution, the slack field of every other instruction in the scheduler is decremented — indicating that these instructions are becoming more critical.

With this piece of specialized hardware we could trace a dynamic critical path through the execution of a condensed graph.

Hardware scheduling is a specific approach to the more general problem of optimal Triple List construction discussed in Chapter 4 Section 4.5.2.

6.3 Afterword

"Uniformity leads to simplicity". In the CG model we have endeavoured to harness this design maxim by providing a single uniform formalism for expressing three computing paradigms. We have shown that each member of the trinity has an associated "active ingredient" in the form of an associated element in the CG's computation triple. When that element drives the computation (while the other two elements remain fixed in the static description of the CG) its associated paradigm is a special case of the CG model. In addition, by having more than one element of the triple drive a computation, we have seen that it is possible for the CG model to either side-step, eliminate or mitigate some of the traditional problems associated with each paradigm.
EPILOGUE
Summary

In the field of computer science different computation models have been developed over time. The oldest is the imperative model. There the sequence of instruction execution is determined by the programmer. In contrast, the more recently developed dataflow and functional language models employ different sequencing constraints. In the dataflow model instruction execution is determined by the availability of operand data; and in the functional language model an instruction is only executed if there is a need for its result. This thesis introduces a graph-theoretic formalism in which these three computation models are unified.

A condensed graph, or CG, is a mathematical structure which is recursively defined to be either an atomic node or a node representing the condensation of a graph of CGs. Condensed graphs can be organized into arbitrarily complex condensation sequences — representing the same information at many levels of abstraction. These sequences are used in the CG model of computation to represent both algorithms and structured data.

Computation in the CG model is a process of CG association. We identify two association types and two processes (streaming and grafting) for transforming graphs from one type to the other. A static association is formed when a CG is allied with the input port of a node during the construction of the condensation sequence. In contrast, a dynamic association occurs when the CG is linked with the input port of a node is created as the result of an instruction execution.

The CG execution model is based on the bringing together of complementary elements of a computation triple. This triple is composed of a set of operands, a function and an output environment. Triple elements are bound to CGs and so are brought together by CG association. A set of reduction rules is defined which dictates how the triple element bindings take place. Thus, CGs containing two or three elements are created by associating two or more CGs together. A complete CG, a term which is synonymous with an instruction, is formed when all three elements of the computation triple are bound to a node.

In the general CG model, all elements of the triple drive the computation by being free to take part in dynamic CG associations. In the special cases where the computation is driven by one triple element, the resulting execution is either availability-driven, coercion-driven or control-driven — depending on whether that element is the set of operands, the output environment or the function. Thus, the CG model unifies the availability-driven, coercion-driven and control-driven models of computing into one single, uniform formalism.

A machine model for the construction and execution of computation triples is presented and it is used to successfully address some of the disadvantages and limitations associated
with each of the individual computing models.
Samenvatting

Binnen de informatica zijn in de loop van de tijd verschillende modellen van berekeningen ontwikkeld. Het oudst is het imperatieve model. Hierin wordt de berekening gestuurd door de instructies in het programma. Nieuwer zijn het dataflow model en het functioneel programmeren, waarin de berekening respectievelijk wordt gestuurd door de aanwezigheid van gegevens en door de behoefte aan resultaten. Het proefschrift introduceert een graaf-theoretisch formalisme waarin deze drie berekeningsmodellen worden vereenigd.

Een condensed graph (CG) is een graaf waarin iedere knoop of atomair is of opnieuw een CG representeert; een CG die gecondenseerd is tot een enkele knoop. Met condensed graphs kunnen willekeurig complexe condensatierijen worden gebouwd, waarin dezelfde informatie op verschillende abstractive niveaus wordt gerepresenteerd. Dergelijke condensatierijen worden in het CG-model zowel gebruikt om algoritmen als om datastructuren vast te leggen.

Berekeningen in het CG-model vinden plaats door middel van associatie. We kennen twee soorten van associatie, alsnog twee processen om grafen te transformeren. We spreken van een statische associatie wanneer, bij het bouwen van een condensatierij, een CG wordt geplaatst op een input van een knoop. Een dynamische associatie treedt op wanneer de plaatsing pas optreedt tijdens de uitvoering van het programma.

In het berekeningsmodel wordt iedere CG voorgesteld door een berekenings-drietal. Het model is gebaseerd op het samenbrengen van de elementen van deze drietalen. Zo'n drietal bestaat uit een verzameling argumenten, een functie en een output-omgeving. Door middel van associatie worden de elementen van de drietalen ingevuld. Er zijn reductierules die bepalen hoe de associatie van CGs aan drietal-elementen plaatsvindt. Zodra alle drie elementen van een drietal zijn ingevuld is de CG een uitvoerbare instructie geworden. De volgorde waarin de elementen van de berekenings-drietalen beschikbaar komen bepaalt welk type berekeningsmodel we van doen hebben.

In het algemene CG-model sturen de drie elementen van de drietalen gezamenlijk de berekening, doordat ze alle drie dynamisch met de CG geassocieerd kunnen worden. In het speciale geval dat de berekening slechts door één element wordt gestuurd, is de berekening — afhankelijk van of dit element de verzameling argumenten, de functie of de output-omgeving is — availability-driven, coercion-driven of control-driven. Op deze manier vereenvoudigt het CG-model deze verschillende berekeningswijzen tot een uniform formalisme.

De unificering die het CG-model bewerkstelligt geeft een aantal nieuwe voordelen. Deze worden zo goed mogelijk uitgebad in een computer architectuur, die wordt
voorgesteld. Voor ieder van de berekeningsmodellen kent deze architectuur ook beperkingen en nadelen. Deze worden zowel kwalitatief als kwantitatief onderzocht en er worden voorstellen gedaan die de nadelen succesvol vermijden.
References


REFERENCES


REFERENCES


Index

CG
- Atomic, 25
- Complete, 37
  - Simple Data, 77
- Condensed Vertex, 25
- Constants, 33
- Definition of, 25
- Dynamic Association, 33
- Incomplete, 37
- Language, 54
- Mobile, 37
- Ports of, 31
- Static Association, 33
- Stationary, 37
- evap Function, 23, 32
- E Node, 52
  - Firing, 80
  - Mutual Reduction, 43
  - Strictness, 39
- E Vertex, 21
- X Node, 52
  - AF Deallocations, 89
  - Deconstruction, 53
  - Mutual Reduction, 43
  - Ports, 31, 39
  - Strictness, 39
  - Subgraph Constant Evaluation, 41
- X Vertex, 21
  - branch Node, 29
  - fetch Node, 79
  - Condensed node operand, 80
  - filter Node, 29, 52, 54, 80
    - Strictness, 39
  - spec Node, 70
- Abstract Data Types, 33
- Activation Frame

Also see AF, 51
AF, 71, 89, 91, 95
X node execution, 52
filter node execution, 52
Ancestor, 70
Deallocation, 70
Discarding, 73
Identification, 71
in V-graph, 52
Memory Management Policy, 53, 57, 89
Reuse, 74
Algorithm
- CG constants, 37
- Definition of, 29
- Hierarchical Composition, 33
- Hierarchical Decomposition, 32
- Stepwise Refinement, 32
Ancillary Processors
- Also see AP, 49
AP
- Function-strict, 50
- Kinds of, 50
Arcs
- Global Input, 8
- Global Output, 8
Atomic
- CG, 25
  - Function, 42
  - Nodes, 33, 38
  - Operand, 43
Atoms, 3
Availability-driven, 1, 2, 6, 9, 30, 34, 35, 39, 45, 49, 60, 65, 72, 76, 77, 86, 95
Backtracking, 42, 66, 74, 94
Definition of, 35
Graph
Acyclic, 13, 29
Ancestor, 84, 89, 91
Cyclic, 17, 18, 29
Linear Subgraph, 72
Linear Subgraph, 91
Graph Reduction, 4
Graph-re-writing, 53
H-graph, 32, 39
Horizontal Graph
Also see H-graph, 24
I-structure, 79
IG(D,P)
Meaning of, 19
IG(D, P^0)
Meaning of, 21
InDeg(v, D), 14
Induced Partition Subgraph, 19
Input Environment, 20
Output Environment, 20
Induced Subgraph
Definition of, 15
Receiver, 19
Transmitter, 19
Input Environment, 14
Induced Partition Subgraph, 20
Instruction
Definition of, 33
Token, 50
Karp, 6
Kirkhoff, 13
Lazy Evaluation, 6
Load balancing, 10, 97
Locality, 7, 10, 11, 76, 93, 97
In the CG machine, 63, 71
Manchester Machine, 10, 94
Matching Bottleneck, 9
Miller, 6
MIT, 95
Multi-ring, 97
Mutable CGs, 34
Mutual Reduction, 42, 80
Net, 13
Node, 31
Constructor, 84, 91
Fireable, 38
Fired, 38
Grafted, 35
Reactivation, 37
Reducible, 37
Stemmed, 35
Completely, 36
Partial, 36
Store, 79
Non-Strict
Firing Rule, 8, 9, 30
Function, 4
Normal form, 4
Object-Oriented, 33, 96
OutDeg(v, D), 14
Output Environment, 14
In result token, 50
Induced Partition Subgraph, 20
Partition
DAG Enforcing, 18
DAG Preserving, 18
Graph-trivial, 17
of a set, 15
Point-Trivial, 17
Subgraph
Seeded, 21
Population, 80
by TM instruction evaluation, 51
In Mutual Reduction, 42
Ports, 31
In-degree, 31
Input, 31
of a CG, 31
of x Node, 31
Of e Node, 81
Out-degree, 31
Output, 31
Pre-order walk, 65
### INDEX

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVM</td>
<td>97</td>
</tr>
<tr>
<td>Receiver</td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td>20</td>
</tr>
<tr>
<td>Redux</td>
<td>4</td>
</tr>
<tr>
<td>Reduction</td>
<td></td>
</tr>
<tr>
<td>Applicative-order</td>
<td>4</td>
</tr>
<tr>
<td>Mutual</td>
<td>42</td>
</tr>
<tr>
<td>Node</td>
<td>37</td>
</tr>
<tr>
<td>Normal-order</td>
<td>4</td>
</tr>
<tr>
<td>Strict</td>
<td>38, 95</td>
</tr>
<tr>
<td>Referential Transparency</td>
<td>5</td>
</tr>
<tr>
<td>Relation</td>
<td>13</td>
</tr>
<tr>
<td>Computing with</td>
<td>96</td>
</tr>
<tr>
<td>Irreflexive</td>
<td>14</td>
</tr>
<tr>
<td>Remaining Packet Garbage</td>
<td>9, 30, 76, 94</td>
</tr>
<tr>
<td>Result tokens</td>
<td>50</td>
</tr>
<tr>
<td>Reverse pre-order walk</td>
<td>65</td>
</tr>
<tr>
<td>Rodriguez</td>
<td>6</td>
</tr>
<tr>
<td>Safety</td>
<td>30</td>
</tr>
<tr>
<td>Scheduling</td>
<td>71–74</td>
</tr>
<tr>
<td>Hardware</td>
<td>97</td>
</tr>
<tr>
<td>Side-Effect</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>90</td>
</tr>
<tr>
<td>Side-Effects</td>
<td>1, 5, 10, 11, 77</td>
</tr>
<tr>
<td>Speculation</td>
<td>11, 95</td>
</tr>
<tr>
<td>Speculative</td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td>46</td>
</tr>
<tr>
<td>Parallelism</td>
<td>30</td>
</tr>
<tr>
<td>Subcomputation</td>
<td>57</td>
</tr>
<tr>
<td>State</td>
<td>90</td>
</tr>
<tr>
<td>Transition Semantics</td>
<td>3</td>
</tr>
<tr>
<td>Static Association</td>
<td>33</td>
</tr>
<tr>
<td>in CG language</td>
<td>54</td>
</tr>
<tr>
<td>Stemming Process</td>
<td></td>
</tr>
<tr>
<td>Definition of</td>
<td>35</td>
</tr>
<tr>
<td>Store</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>83</td>
</tr>
<tr>
<td>Infinite</td>
<td>85</td>
</tr>
<tr>
<td>Lazy</td>
<td>79</td>
</tr>
<tr>
<td>Mutable</td>
<td>91, 94</td>
</tr>
<tr>
<td>Not Shared</td>
<td>84</td>
</tr>
<tr>
<td>Shared</td>
<td>84</td>
</tr>
<tr>
<td>Store Node</td>
<td>79</td>
</tr>
<tr>
<td>Strict</td>
<td></td>
</tr>
<tr>
<td>Firing Rule</td>
<td>8</td>
</tr>
<tr>
<td>Function</td>
<td>4</td>
</tr>
<tr>
<td>Function-Strict</td>
<td>38</td>
</tr>
<tr>
<td>Reduction-Strict</td>
<td>38</td>
</tr>
<tr>
<td>String Reduction</td>
<td>4</td>
</tr>
<tr>
<td>Structured data</td>
<td>7, 77</td>
</tr>
<tr>
<td>Sub-Definition</td>
<td>40</td>
</tr>
<tr>
<td>Deconstruction</td>
<td>42</td>
</tr>
<tr>
<td>Sharing</td>
<td>42</td>
</tr>
<tr>
<td>Sub-definition</td>
<td></td>
</tr>
<tr>
<td>grafting in the TM</td>
<td>53</td>
</tr>
<tr>
<td>Subgraph</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>34, 41</td>
</tr>
<tr>
<td>Anonymous</td>
<td>54</td>
</tr>
<tr>
<td>Sub-Definition</td>
<td>40</td>
</tr>
<tr>
<td>Unconnected</td>
<td>81</td>
</tr>
<tr>
<td>Tagged Data Space</td>
<td>93</td>
</tr>
<tr>
<td>Tail-recursive computation</td>
<td>72</td>
</tr>
<tr>
<td>TCP</td>
<td>93</td>
</tr>
<tr>
<td>Actions</td>
<td>50</td>
</tr>
<tr>
<td>Deconstruction</td>
<td>53</td>
</tr>
<tr>
<td>Memory Deallocation</td>
<td>52</td>
</tr>
<tr>
<td>Subgraph Constant Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>To populate a definition</td>
<td>52</td>
</tr>
<tr>
<td>Triple List Building</td>
<td>63, 65, 72</td>
</tr>
<tr>
<td>Throttle</td>
<td></td>
</tr>
<tr>
<td>Coarse-Grain</td>
<td>95</td>
</tr>
<tr>
<td>Memory</td>
<td>62, 63</td>
</tr>
<tr>
<td>Threshold</td>
<td>63</td>
</tr>
<tr>
<td>Triple List</td>
<td>62, 63</td>
</tr>
<tr>
<td>Threshold</td>
<td>66, 73</td>
</tr>
<tr>
<td>Throttling</td>
<td>9, 62</td>
</tr>
<tr>
<td>TM</td>
<td>85, 90</td>
</tr>
<tr>
<td>Grafting in</td>
<td>53</td>
</tr>
<tr>
<td>Handling sequential graphs</td>
<td>72</td>
</tr>
<tr>
<td>Instructions</td>
<td>49</td>
</tr>
<tr>
<td>Transmitter</td>
<td></td>
</tr>
<tr>
<td>Global</td>
<td>20</td>
</tr>
<tr>
<td>Triple</td>
<td>33, 37, 50, 66, 74, 76, 78–80, 90, 91, 97</td>
</tr>
<tr>
<td>Triple Constructor Process</td>
<td></td>
</tr>
<tr>
<td>Also see TCP</td>
<td>50, 85</td>
</tr>
<tr>
<td>Triple List</td>
<td>50, 97</td>
</tr>
</tbody>
</table>
Triple Manager
   Also see TM, 49

V-Graph, 50, 51, 69, 72, 73
   Condensed Node Execution, 51
   Memory, 51
   Memory Management Policy, 53
   Vertex Set, 24
   Vertical Cross-Section, 72

V-graph, 24, 38, 39
   AF in, 52
   Memory, 50
   Pre-order walk, 65
   Reverse pre-order walk, 65

Veen, Arthur, 8

Vertex
   Input Environment, 14
   Environment, 14
   In-degree, 14
   Internal, 14
   Out-degree, 14
   Output Environment, 14

Verteex, Receiver, 14

Vertex, Transmitter, 14

Vertical Graph
   Also see V-graph, 24

Von Neumann, 3

Zero-Arity Function, 42, 77
About the Author

John Morrison was born in Cork City, Ireland, on the 9th July 1961 and entered University College Cork in 1979. He received his B.Sc. degree in Computer Science in 1983 and his M.Sc. degree by research in 1987 with a thesis entitled "An Investigation into Virtual Time and Time Warp".

From 1986 to 1991 he was employed as a research scientist by Philips Research Laboratories (Philips Natuurkundig Laboratorium) in Eindhoven. There he took part in a number of large European funded, ESPRIT, projects. Among them were ESPRIT 415a (1984-1989), and ESPRIT 2427 (1989-1991). He was also a member of the PRISMA project (1986-1990), which was sponsored by the Dutch 'stimuleringsprojectteam Informatieonderzoek'.

In September 1991 he joined the staff of the Computer Science Department at University College Cork, Ireland.

Contact address:
Department of Computer Science,
University College,
Cork,
Ireland.
email: J.Morrison@cs.ucc.ie
Stellingen

behorende bij het proefschrift

Condensed Graphs:
Unifying
Availability-Driven, Coercion-Driven
and Control-Driven Computing

van

John P. Morrison

Technische Universiteit Eindhoven
30 oktober 1996
1. The Condensed Graph's representation, and processing, of structured data is indistinguishable for its representation, and processing, of function graphs.

2. Form a Source-Oriented partition $\mathcal{P} = P_1, P_2, \ldots, P_n$ of a DAG $D$ by:
   
   (a) Initially letting $\mathcal{P}$ be the point-trivial partition of $D$.
   
   (b) Adding a second element to every singleton $P_i$ if, and only if, that element is the only successor of $P_i$ in $D$.
   
   (c) Form a new subset $P_{ij} = P_i \cup P_j$ (if $P_i$ and $P_j$ have an element in common) and replace $P_i$ and $P_j$ by $P_{ij}$ in $\mathcal{P}$. Repeat this "unioning procedure" until all the elements of $\mathcal{P}$ are again pair-wise disjoint.

The Source-Oriented partition has the following properties:

- It is independent of the order in which points are chosen for its construction.
- Its induced partition subgraphs constitute connected, converging chains with a single, unique receiver.
- Condensation with respect to this partition is DAG preserving.
- Repetitive condensation with respect to this partition results in a linear condensed graph.

3. In Jefferson's Time Warp mechanism [1] it is claimed that states can be saved less frequently than after every event. Such a strategy, however, requires a more complicated state restoration strategy than articulated there. The more infrequent the state saving process, the more redundant work is precipitated. Furthermore, under certain circumstances the Time Warp Mechanism can become live-locked [2]. This live-lock can be avoided by either (a) saving the state of a process whenever it sends a message or (b) by not transmitting any message or antimessage recalculated during the redundant-work phase.

4. In the debugging phase of a parallel program, deterministic playback can be achieved by testing the program under a modified Time Warp operating system [2].

5. In certain circumstances, including when the size of the parameter data to a remote procedure call is large, it may be more efficient to communicate the procedure code to the calling site rather than to communicate the parameters to the site of the procedure.

6. The use of graphical user-interface management systems enables interfaces to be designed quickly and easily by non-specialists. These systems have done much to popularize the production of impressive, professional-looking software. However, with such tools an incremental design approach is very attractive and can lead the unsuspecting designer into the construction of flawed Knowledge.
7. A spectrum of object models can be identified for the implementation of software objects [3]. These range from passive to active, from private to public, and combinations thereof. These models greatly reduce the size of the executable code and lead to faster executions. Many can be uncovered by compiler analysis or can be indicated by an annotation mechanism [4].

8. In ray-tracing, the number of object-feeler intersection checks for either bounding volume, or voxel, hierarchies in scenes of more than one object can always be reduced by employing a hierarchical graph structure in conjunction with these hierarchies [5].

9. Accessorizing golf carts with satellite up-links is evidence that technology has become far more sophisticated than the applications for which it is employed.

10. Students work harder when they set their own deadlines. When told that a particular task should take so long, they automatically assume that that time applies to the weakest among them. Thus, the duration of their effort tends to be inversely proportional to size of their ego.

11. On a sea of serendipity there's many a notion to navigate.


